Sound and Reusable Components for Abstract Interpretation

SVEN KEIDEL and SEBASTIAN ERDWEG, JGU Mainz, Germany

Abstract interpretation is a methodology for defining sound static analysis. Yet, building sound static analyses for modern programming languages is difficult, because these static analyses need to combine sophisticated abstractions for values, environments, stores, etc. However, static analyses often tightly couple these abstractions in the implementation, which not only complicates the implementation, but also makes it hard to decide which parts of the analyses can be proven sound independently from each other. Furthermore, this coupling makes it hard to combine soundness lemmas for parts of the analysis to a soundness proof of the complete analysis.

To solve this problem, we propose to construct static analyses modularly from *reusable analysis components*. Each analysis component encapsulates a single analysis concern and can be proven sound independently from the analysis where it is used. We base the design of our analysis components on *arrow transformers*, which allows us to compose analysis components. This composition preserves soundness, which guarantees that a static analysis is sound, if all its analysis components are sound. This means that analysis developers do not have to worry about soundness as long as they reuse sound analysis components. To evaluate our approach, we developed a library of 13 reusable analysis components in Haskell. We use these components to define a k-CFA analysis for PCF and an interval and reaching definition analysis for a While language.

CCS Concepts: • **Software and its engineering → Automated static analysis**; • **Theory of computation** → *Proof theory*.

Additional Key Words and Phrases: Abstract Interpretation, Static Analysis, Soundness

ACM Reference Format:
Sven Keidel and Sebastian Erdweg. 2019. Sound and Reusable Components for Abstract Interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 176 (October 2019), 28 pages. https://doi.org/10.1145/3360602

1 INTRODUCTION

Abstract interpretation [Cousot and Cousot 1979] is a methodology for defining sound static analysis. A static analysis is sound if it predicts, at compile time, all relevant dynamic behavior of a program. For example, if a sound static nullness analysis claims a variable is not null, then this variable may not store a null pointer in any execution of the program. Analysis soundness is important whenever a developer or optimizing compiler acts on the analysis result [Knoop and Rüthing 1999]. For example, when a developer or compiler omits a null check, only a sound nullness analysis can provide the required guarantee that this check is indeed redundant.

Building sound static analyses for modern programming languages is difficult. Analysis developers must provide abstractions for all values (e.g., integers, strings, objects) as well as for all effects (e.g., environments, stores, exceptions) supported by the analyzed language. The combination of these abstractions forms the essence of a static analysis. However, a static analysis often closely couples different abstractions, which makes it harder to replace them. This coupling also complicates a soundness proof, as it is not clear which parts of the analysis can be proven sound independently

Authors' address: Sven Keidel; Sebastian Erdweg, JGU Mainz, Germany.

2475-1421/2019/10-ART176
https://doi.org/10.1145/3360602

and which parts have to be proven together. Furthermore, the coupling makes it hard to establish an end-to-end soundness proof, from soundness lemmas for each part of the analysis.

In this paper, we propose *analysis components* as modular building blocks for static analyses. An analysis component is governed by an interface $\mathcal{I}$ that describes which concern of the analyzed language the component implements. For example, an analysis component for stores will enlist read and write operations in its interface $\mathcal{I}$. The crucial feature of analysis components is that they can be proven sound individually, and the soundness of the complete static analysis follows by construction. To this end, each analysis provides both the canonical concrete semantics C and an abstract semantics $\widehat{C}$ for the operations enlisted in the interface. An analysis component is sound if for each operation in $\mathcal{I}$, the abstract semantics $\widehat{C}$ approximates the concrete semantics C. Analysis developers can use such analysis components as building blocks to construct sound static analyses.

In our approach, analysis developers define a static analysis as an interpreter against the interfaces of analysis components. We call such an interpreter a *generic interpreter* because it is not specific to the concrete or abstract semantics stipulated by the analysis components. Indeed, we can instantiate the same generic interpreter to obtain a range of alternative language semantics by selecting compatible components:

- We obtain a concrete interpreter using the canonical concrete semantics C of the components.
- We obtain an abstract interpreter using the abstract semantics $\widehat{C}$ of the components.

A key theoretical result of this work is that the instantiated abstract interpreter is guaranteed to soundly approximate the instantiated concrete interpreter if the used analysis components are sound. That is, analysis developers do not need to worry about soundness as long as they combine sound analysis components.

As consequence of our design, the same analysis component can be reused across analyses and across languages without change. For example, when researchers discover a new abstraction for stores, they can cast it as an analysis component implementing the `Store` interface and prove the component sound. Afterwards, any existing analysis that uses a `Store` component can easily be upgraded to use the new abstraction, without needing to revisit the soundness of the analysis. Moreover, many analysis components like `Store` are actually language-independent and can be reused across languages. Indeed, most language-specific behavior is captured by the generic interpreter. Thus, to target a new language, an analysis developer can reuse existing analysis components and only has to develop a generic interpreter for the new language.

We demonstrate that our design of analysis components is feasible by developing the component-based analysis framework in Haskell. In our framework, we represent analysis components as a pair of arrow transformers, a generalization of monad transformers. We can compose these arrow transformers and use them to instantiate the generic interpreter, thus obtaining executable concrete and abstract interpreters. We extend the arrow-based theory on compositional soundness proofs for abstract interpreters by Keidel et al. [2018] to allow reasoning about isolated arrow transformers. This forms the basis of our new theory about horizontal and vertical composability of analysis components, and the proof obligations entailed thereby.

We evaluate our design by creating the open-source library *Sturdy* of 13 sound analysis components in Haskell. We demonstrate the applicability of our analysis components by using them to define well-known analyses: A k-CFA analysis [Shivers 1991] for PCF as well as an interval analysis [Nielson et al. 1999] for a WHILE language. We were able to define both analyses modularly by describing generic interpreters and analysis components separately. We then changed the WHILE analysis in two different ways to study the impact on the analysis definition and soundness proof. First, we changed the analysis to additionally compute reaching definitions [Nielson et al. 1999] rather than intervals only. Second, we changed the WHILE language to add exception handling.

In both cases, changes were confined to a single analysis component and the generic interpreter, whereas the rest of the analysis definition and soundness proofs remained stable.

In summary, we make the following contributions:

- We propose an approach for the modular construction of static analyses from reusable analysis components, which are based on arrow transformers (Section 2).
- We define a soundness proposition for analysis components and demonstrate how they can be shown sound in isolation (Section 3).
- We develop a theory that explains the horizontal and vertical composition of analysis components and when their soundness is preserved (Section 4).
- We prove that a static analysis based on analysis components is sound, if all its analysis components are sound (Section 5).
- We provide an open-source library of reusable analysis components in Haskell (Section 6).
- We evaluate the applicability and reusability of our components by defining a k-CFA analysis, an interval analysis, and a reaching definitions analysis (Section 7).

2 ANALYSIS COMPONENTS BY EXAMPLE

Static analyses mix language concerns, which convolutes their implementation and soundness proof. In this section, we first illustrate the problems that arise when analyses mix concerns, before sketching our solution of analysis components.

2.1 Problem Statement

A static analysis is sound if it correctly approximates the concrete semantics. Analysis soundness has been specifically well-studied for abstract interpreters, which need to approximate the concrete interpreter. Unfortunately, the soundness criteria for abstract interpreters requires reasoning about the whole interpreter definition. As we show here, such non-modular reasoning quickly becomes unwieldy, even for simple languages.

For example, consider the following concrete interpreter `run` and abstract interpreter $\widehat{\texttt{run}}$ for a simple While language implemented in Haskell. We only show the case for assignments `Assign`.

```
data Expr = ...
data Statement = Assign Var Expr | If Expr [Statement] [Statement] | While Expr [Statement]

run :: Map Var Addr₁ → Map Addr Val₂ → [Statement] → Maybe₃ (Map Addr Val₂)
run env₁ store₂ (Assign var expr : rest) = case₃ eval env₁ store₂ expr of
  Just₃ val → case lookup₁ var env₁ of
    Just addr → run env₁ (insert₂ addr val store₂) rest
    Nothing → let addr = alloc env₁ var
      in run (insert₁ var addr env₁) (insert₂ addr val store₂) rest
  Nothing₃ → Nothing₃

run̂ :: Map Var Addr₁ → Map̂ Addr Val̂₂ → Int₄ → [Statement] → Maybe̲₃ (Map̂ Addr Val̂₂)
run̂ _    _      fuel₄  _          | fuel ≤ 0₄ = JustOrNothing₃ ⊤₂
run̂ env₁ store₂ fuel₄ (Assign var expr : ss) = case₃ eval̂ env₁ store₂ expr of
  Just₃ val → case lookup₁ var env₁ of
    Just addr → run̂ env₁ (insertWith₂ (⊔) addr val store₂) (fuel-1₄) ss
    Nothing → let addr = alloc̲ env₁ store₂ var val
      in run̂ (insert₁ var addr env₁) (insertWith₂ (⊔) addr val store₂) (fuel-1₄) ss
  Nothing₃ → Nothing₃
  JustOrNothing₃ val → Nothing₃ ⊔ (... same code as for Just₃ val)
```

The concrete interpreter `run` takes an environment (mapping variables to addresses) and a store (mapping addresses to values), and yields a possibly updated store if the execution does not fail. The interpreter code itself is standard, but we color-coded and enumerated the parts of the code that relate to different concerns: $\texttt{environment}_1$, $\texttt{store}_2$, and $\texttt{failure}_3$.

The abstract interpreter $\widehat{\texttt{run}}$ handles a fourth concern: $\texttt{termination}_4$. In this simple example, we use a fuel counter that we decrease on every recursive call, and we top out when no fuel is left. For the $\texttt{environment}_1$ concern, $\widehat{\texttt{run}}$ uses the same representation and operations as `run`, but addresses may now be shared between variables. For the $\texttt{store}_2$ concern, $\widehat{\texttt{run}}$ uses a representation that maps addresses to an abstract value domain $\widehat{\texttt{Val}}$, and it uses `insertWith` to join values in the store. Finally, for the $\texttt{failure}_3$ concern, $\widehat{\texttt{run}}$ uses a representation with a third alternative `JustOrNothing`.

Even though the analysis $\widehat{\texttt{run}}$ is fairly simple, it highlights two key challenges when developing sound static analyses:

Modular Implementation The code of $\widehat{\texttt{run}}$ fails to separate concerns and mixes them with language-specific code. That is, all concerns are directly addressed in the analysis code and there is high coupling. This entails the standard problems [Parnas 1972]: It becomes hard to update the code of one concern without affecting other concerns.
Ideally, we would like to implement each concern separately and independent of $\widehat{\texttt{run}}$ as a reusable component. That is, we would like to hide the implementation of each concern behind an interface and only use that interface in $\widehat{\texttt{run}}$. We could then instantiate $\widehat{\texttt{run}}$ by selecting and composing appropriate components. This would allows us, for example, to exchange the implementation for stores without having to think about environments or failures. This would also make it easier to adapt the analysis when the analyzed language changes. Many existing analysis frameworks separate concerns to some extent (e.g., call graph construction and transfer functions), but in a way that precludes addressing the second challenge.

Modular Soundness Proof The entanglement of concerns in $\widehat{\texttt{run}}$ also complicates the soundness proof significantly. In order to show that $\widehat{\texttt{run}}$ soundly approximates `run`, we have to reason about all concerns at once. Moreover, a change to any of the concerns potentially invalidates the entire soundness proof. Essentially, the problems from the implementation are reflected in the soundness proof: It becomes hard to update the code of one concern without affecting the other ones.
Ideally, we would like to prove the soundness of each component separately and independent of $\widehat{\texttt{run}}$. That is, we would like to find a soundness proposition that we can prove separately for each component, and only use that proposition in the soundness proof of $\widehat{\texttt{run}}$. We can then obtain a provably sound analysis by instantiating $\widehat{\texttt{run}}$ with appropriate components, as long as each component satisfies the soundness proposition. One of the key questions is how such a soundness proposition may look like, and how sound components can be composed to yield sound compound components.

In the remainder of this section, we will discuss two designs of components for modularly defined and sound static analyses. The first component design is simple yet fails to address our challenges, illustrating why a good component design is difficult to come by. We resolve these issues in our second component design, which is based on arrow transformers.

2.2 A First Attempt to Design Analysis Components

In this subsection, we propose a *preliminary* design for analysis components that addresses parts of the two challenges of the previous subsection. In this preliminary design, an analysis component consists of four parts, as we illustrate in Figure 1: An interface describing the operations of the component, a concrete and an abstract instance of the interface to define the concrete and abstract

Interface

```
class ExceptOps exc e where
  throw :: e → exc e x
  catch :: exc e y → (e → y) → y
```

Soundness Proof

$\texttt{throw} \mathrel{\dot{\sqsubseteq}} \widehat{\texttt{throw}}$
$\texttt{catch} \mathrel{\dot{\sqsubseteq}} \widehat{\texttt{catch}}$

Concrete Instance

```
data Except e x =
  Success x | Fail e

instance ExceptOps Except e where
  throw e = Fail e
  catch exc h = case exc of
    Success x → x
    Fail e    → h e
```

Abstract Instance

```
data Except^ e x =
  Success^ x | Fail^ e | SuccessOrFail^ x e

instance ExceptOps Except^ e where
  throw^ e = Fail^ e
  catch^ exc h = case exc of
    Success^ x          → x
    Fail^ e             → h e
    SuccessOrFail^ x e  → x ⊔ h e
```

Fig. 1. Preliminary design of an analysis component for exceptions. We write $\texttt{f} \mathrel{\dot{\sqsubseteq}} \widehat{\texttt{f}}$ as a short-hand to say that $\widehat{\texttt{f}}$ soundly approximates f.

semantics, and a proof that the abstract semantics soundly approximates the concrete semantics for each operation.

We illustrate this design in Figure 1 for a component providing exception handling. The interface is parameterized by an exception type `exc e x`, which describes a computation that throws an exception `e` or terminates successfully with `x`. The `catch` operation takes a computation `exc e y` and extracts the value `y` or handles the exception with `(e → y)`. The concrete instance of the component use data type `Except` as exception type and implements the operations in a standard way. The abstract instance uses error type $\widehat{\texttt{Except}}$ that has an extra case $\widehat{\texttt{SuccessOrFail}}$, representing a computation that succeeded or failed. For $\widehat{\texttt{SuccessOrFail}}$, the abstract $\widehat{\texttt{catch}}$ joins ($\sqcup$) the outcomes of the success and fail cases. Finally, the component contains a soundness proof for $\widehat{\texttt{throw}}$ and $\widehat{\texttt{catch}}$ (we skip the details for now). This preliminary design of analysis components addresses parts of our design challenges, but not all of them:

Modular Implementation We succeeded in encapsulating analysis concerns in components, and components can be exchanged with other components implementing the same interface. However, our components do not compose. For example, consider the composition of the exception component from above with a component for stores. The problem is that the exception component describes computations of the form `e → y` (see the type of `catch`), where the store component describes computations of type `(store,x) → (store,y)`. To add stores, we would need to change the type of the `catch` operation to thread a store:

```
catch :: (store,exc e y) → (e → (store,y)) → (store,y)
```

This is not modular because the interface for the exception component has changed in an incompatible way and we cannot reuse previous implementations. To resolve this, we need to make the exception component parametric in the shape of computations so that it can accommodate effects (like store passing) imposed by other components.

Modular Soundness Proof We succeeded in making the soundness of each analysis component separately provable. For example, we can show that $\widehat{\texttt{catch}}$ soundly approximates `catch` given a standard Galois connection between `Except` and $\widehat{\texttt{Except}}$. But as long as the composition of components requires changes to the interface or instances to accommodate new effects, previous soundness proofs become void. The question is what happens when we follow our plan of making components parametric in the shape of computations. This will require us to make the soundness proofs parametric, too, meaning we need to proof soundness independent of effects imposed by other components.

In the following subsection, we refine our first design to address both challenges.

Interface

```
class ArrowExcept e c where
  throw :: c e x
  catch :: c x y → c (x,e) y → c x y
```

Soundness Proof

$\forall c \leftrightarrows \widehat{c}$,
$\text{throw}_c \sqsubseteq \widehat{\text{throw}_{\widehat{c}}}$
$\text{catch}_c \sqsubseteq \widehat{\text{catch}_{\widehat{c}}}$

Concrete Instance

```
data Except e x =
  Success x | Fail e
type ExceptT e c x y = c x (Except e y)

instance ArrowExcept e (ExceptT e c)
  throw = proc e → returnA ≺ Fail e
  catch f h = proc x → do
    exc ← f ≺ x
    case exc of
      Success x → returnA ≺ x
      Fail e    → h ≺ (x,e)
```

Abstract Instance

```
data Except^ e x =
  Success^ x | Fail^ e | SuccessOrFail^ x e
type ExceptT^ e c x y = c x (Except^ e y)

instance ArrowExcept e (ExceptT^ e c)
  throw^ = proc e → returnA ≺ Fail^ e
  catch^ f h = proc x → do
    exc ← f ≺ x
    case exc of
      Success^ y          → returnA ≺ y
      Fail^ e             → h ≺ (x,e)
      SuccessOrFail^ y e  → (returnA ≺ y)
                          ⊔ (h ≺ (x,e))
```

Fig. 2. An arrow-based analysis component for exceptions.

2.3 Arrow-Based Analysis Components

In the previous subsection, we presented a preliminary design for analysis components that supported separation of concerns but failed to support component composition. To make analysis components composable, we abstract over the effects imposed by other components using a higher-order type parameter `c`, which we add to each interface as illustrated in Figure 2 for exceptions. The type parameter `c` has kind $* \to * \to *$ and describes computations, that is, `c x y` is a computation with input `x` and output `y`. In the literature, this design is known as *arrows* [Hughes 2000]

Arrows abstract over effects of computations and are a generalization of monads. For example, we can define an arrow `Arr` as `Arr x y = (Store,x) → Except e (Store,y)`, which represents a computation that threads a store and may yield an exception. But, importantly, we can define parametric arrow computations without specifying the exact arrow type. This is similar to monads, which provide `return` and `bind` operations for defining parametric monadic computations. The set of operations for arrows is somewhat larger, but in this paper we will hide the details using the `proc`-notation [Paterson 2001] that is similar to monadic `do`-notation. For example, the implementation of `catch` in the concrete instance of Figure 2 uses `proc x → ...` to introduce an arrow computation that binds the input to `x`. Arrow statement `exc ← f ≺ x` runs `f` on input `x` and binds the result to variable `exc`. Function `returnA` has type `(c x x)` and embeds its input as an arrow output.

Keidel et al. [2018] have previously explored the usage of arrows in the definition of abstract interpreters. They showed that it is possible to define an arrow type for the concrete domain and a separate arrow type for the abstract domain, such that a single generic interpreter can be instantiated to yield the concrete and abstract semantics, respectively. They also showed that this design allows compositional soundness proofs, where each operation of the arrows can be verified independently and the soundness of the instantiated interpreters follows by construction. However, Keidel et al. fail our goal: Their arrows only separate concrete from abstract domain but fail to separate concerns like exceptions and stores—they did not consider *analysis components*.

Inspired by their work, we use arrows in the interface of our analysis components. However, components can only be composable if their implementations permit effects from other components. To this end, we define the concrete and abstract instances of our analysis components

using *arrow transformers*. An arrow transformer wraps an arrow type to impose additional effects. For example, the concrete instance in Figure 2 uses arrow transformer `ExceptT e c`, which adds exceptions of type `e` to the output of a computation `c`. We do the same in the abstract instance using arrow transformer $\widehat{\texttt{ExceptT}}$ `e c`. Using arrow transformers, the implementation of the concrete and abstract operations is parametric in the underlying arrow except for the locally added effect (here: the propagation and representation of exceptions). From now on, we use the notation $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowExcept}}$ to refer to analysis components.

The revised design based on arrow-transformers addresses both of our design goals, where our preliminary design fell short:

Modular Implementation Each component encapsulates a single analysis concern and is exchangeable with other components implementing the same interface. However, in contrast to our preliminary design, analysis components based on arrow transformers are composable. For example, we can compose the exception component $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowExcept}}$ with a store component $\langle \texttt{StoreT}, \widehat{\texttt{StoreT}} \rangle_{\texttt{ArrowStore}}$ to obtain a component which combines both effects:

$$\begin{aligned}&\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}} \rangle_{\texttt{ArrowExcept}} \circ \langle \texttt{StoreT}, \widehat{\texttt{StoreT}} \rangle_{\texttt{ArrowStore}} \\ &\quad = \langle \texttt{ExceptT} \circ \texttt{StoreT}, \widehat{\texttt{ExceptT}} \circ \widehat{\texttt{StoreT}} \rangle_{\texttt{ArrowExcept+ArrowStore}}\end{aligned}$$

Specifically, the composition of arrow transformers stacks their effects (`ExceptT e (StoreT c)`), while the composition of interfaces combines all operations in a new interface. We do not need to change the implementation of either component. Like monad transformers [Liang et al. 1995], the composition of arrow transformers requires a lifting of the inner arrow transformer to the outermost transformer. We show in our evaluation that these liftings are mostly boilerplate and can be derived automatically.

Modular Soundness Proof Like in the preliminary design, we can prove soundness of an arrow-based component by proving each operation of the interface sound. However, since we use arrow transformers, components are parametric in the effects of other components and the soundness proof must be parametric as well. That is, we must show that $\widehat{\texttt{throw}}_{\widehat{\texttt{c}}}$ is sound with respect to $\texttt{throw}_{\texttt{c}}$ for any related arrows $\widehat{\texttt{c}}$ and `c`. We found that such generic soundness proofs are feasible and we provide a large library of provably sound analysis components in Section 6. Sound analysis components following our design are freely composable and their composition always remains sound. We provide the formal results in the upcoming sections.

2.4 Instantiating Concrete and Abstract Interpreters

Using the analysis components described in the previous subsection, we can refactor the concrete and abstract interpreter of Section 2.1. First, we extract a generic interpreter as proposed by Keidel et al. [2018] to capture the similarities of the concrete and abstract interpreter. In contrast to Keidel et al., we parameterize the generic interpreter using the interfaces of analysis components:

```
runGeneric :: (ArrowEnv String addr c, ArrowStore addr val c, ArrowExcept e c, ArrowFix c)
    ⇒ c [Statement] ()
runGeneric = fix $ λrun' → proc stmts → case stmts of
  Assign var expr : rest → do
    val ← eval ≺ expr
    addr ← lookup id alloc ≺ var
    write ≺ (addr,val)
    local run' ≺ (var,addr,rest)
  ...
```

The second step of our refactoring is to compose analysis components implementing all required interfaces of the generic interpreter:

$$\langle \mathsf{EnvT}, \widehat{\mathsf{EnvT}} \rangle_{\mathtt{ArrowEnv}} \circ \langle \mathsf{StoreT}, \widehat{\mathsf{StoreT}} \rangle_{\mathtt{ArrowStore}} \circ \langle \mathsf{ExceptT}, \widehat{\mathsf{ExceptT}} \rangle_{\mathtt{ArrowExcept}} \circ \langle \mathtt{Fix}, \widehat{\mathtt{Fix}} \rangle_{\mathtt{ArrowFix}}$$

The order in which the analysis components are composed matters. For example, in the order above, we obtain a language semantics in which store updates in a `try` block are reset whenever an exception occurs. In contrast, if we swap the order of the store and exception component, we obtain a language semantics in which store updates persist whenever an exception occurs.

The third and final step is to obtain the original concrete and abstract interpreters from Section 2.1 by instantiating the generic interpreter. The concrete slice of the composed analysis component yields the concrete interpreter, the abstract slice yields the abstract interpreter. In Haskell it suffices to specify the desired interpreter type and let the implicit type-class inference select the correct component instances.

```
run :: EnvT (StoreT (ExceptT Fix)) [Statement] ()
run = runGeneric

run^ :: EnvT^ (StoreT^ (ExceptT^ Fix^)) [Statement] ()
run^ = runGeneric
```

As we show in the following sections, we obtain that an abstract interpreter $\widehat{\mathtt{run}}$ soundly approximates a concrete interpreter `run` if they are instances of the same generic interpreter and the fully-composed analysis component is sound. The fully-composed analysis component is sound if all individual analysis components are sound. Thus, analysis soundness follows directly from using sound analysis components.

3 ANALYSIS COMPONENTS AND THEIR SOUNDNESS

To be able to rely on the results of an analysis, the analysis has to be proven sound. In this section, we describe our analysis components formally and how to prove them sound.

To set the stage, let us first recall the definition of soundness [Cousot 1999]: A concrete function $f : A \to B$ is sound with respect to an abstract function $\widehat{f} : \widehat{A} \to \widehat{B}$, if all behavior of f is overapproximated by $\widehat{f}$. More formally, let $\alpha_A : \mathcal{P}A \leftrightarrows \widehat{A} : \gamma_A$ and $\alpha_B : \mathcal{P}B \leftrightarrows \widehat{B} : \gamma_B$ be Galois connections between concrete and abstract domains, then

$$f \text{ is sound w.r.t. } \widehat{f} \quad \text{iff} \quad \forall X \in \mathcal{P}A.\ \alpha_B\{f(x) \mid x \in X\} \sqsubseteq \widehat{f}(\alpha_A(X)).$$

Here the powersets $\mathcal{P}A$ and $\mathcal{P}B$ describe properties of the concrete domain. Given such a property $X \in \mathcal{P}A$ about the inputs of f, the set $\{f(x) \mid x \in X\}$ describes the strongest post-condition of f for the pre-condition X.

This soundness proposition is relative to the Galois connection (α_A, γ_A) and (α_B, γ_B). These Galois connections describe how concrete properties $\mathcal{P}A$ and $\mathcal{P}B$ correspond to abstract values $\widehat{A}$ and $\widehat{B}$. Choosing a Galois connection and an ordering of the abstract domains $\widehat{A}$ and $\widehat{B}$ is part of the analysis design and different Galois connections provide different soundness guarantees. In the following subsection, we describe how to construct Galois connections for analysis components.

3.1 Galois Connections between Analysis Components

Proving soundness requires a Galois connection that relates a concrete domain A to an abstract domain $\widehat{A}$. A Galois connection [Ore 1944] between two preorders A and $\widehat{A}$ consists of a pair of monotone functions (α, γ), where $\alpha : A \to \widehat{A}$ is called the *abstraction function* and $\gamma : \widehat{A} \to A$ is called the *concretization function*, such that

$$\forall x \in A, \widehat{x} \in \widehat{A}.\ \alpha(x) \sqsubseteq_{\widehat{A}} \widehat{x} \text{ iff } x \sqsubseteq_A \gamma(\widehat{x}).$$

Our analysis components consist of operations over a pair of arrow transformers. To relate these operations in a soundness proof, we need to define a Galois connection between these arrow transformers. However, we first describe the shape of Galois connections for regular arrows, because this will guide the design of Galois connections for arrow transformers.

In the following, we use the notation $\mathcal{G}(A, \widehat{A})$ to denote the set of all Galois connections between the types A and $\widehat{A}$. An arrow C is a function that constructs the type of a computation $C(A, B)$, whose input type is A and output type is B. Analogously, a Galois connection between two arrows [Keidel et al. 2018] C and $\widehat{C}$ is a function that takes a Galois connection between the inputs $\mathcal{G}(\mathcal{P}A, \widehat{A})$ and outputs $\mathcal{G}(\mathcal{P}B, \widehat{B})$ and constructs Galois connection between the arrow types:

$$\forall A, \widehat{A}, B, \widehat{B}.\quad \mathcal{G}(\mathcal{P}A, \widehat{A}) \times \mathcal{G}(\mathcal{P}B, \widehat{B}) \to \mathcal{G}(C(A, B), \widehat{C}(\widehat{A}, \widehat{B})).$$

Since our analysis components consist of arrow transformers, we need to generalize the construction further. A Galois connection between arrow transformers T and $\widehat{T}$ maps a Galois connection between the underlying arrow types C and $\widehat{C}$ to a Galois connection between the transformed arrow types $T(C)$ and $\widehat{T}(\widehat{C})$.

Definition 3.1. A Galois connection for an analysis component $\langle T, \widehat{T}\rangle$ is a Galois connection between the two arrow transformers T and $\widehat{T}$. It has the following type:

$$\begin{aligned}\forall C, \widehat{C}.\, [\forall A, \widehat{A}, B, \widehat{B}.\, \mathcal{G}(\mathcal{P}A, \widehat{A}) \times \mathcal{G}(\mathcal{P}B, \widehat{B}) \to \mathcal{G}(C(A, B), \widehat{C}(\widehat{A}, \widehat{B}))] \\ \to [\forall A, \widehat{A}, B, \widehat{B}.\, \mathcal{G}(\mathcal{P}A, \widehat{A}) \times \mathcal{G}(\mathcal{P}B, \widehat{B}) \to \mathcal{G}(T(C)(A, B), \widehat{T}(\widehat{C})(\widehat{A}, \widehat{B}))]\,.\end{aligned}$$

That is, given a Galois connection between the underlying arrow types C and $\widehat{C}$, the function constructs a Galois connection between the arrows $T(C)$ and $\widehat{T}(\widehat{C})$.

The design of an analysis component crucially depends on the choice of Galois connection. The Galois connection dictates how the component's abstract arrow transformer needs to approximate the concrete transformer. Moreover, the Galois connection is not uniquely determined; different Galois connections provide different soundness guarantees. Therefore, the developer of an analysis component also has to specify the corresponding Galois connection in accordance with Definition 3.1.

Example 3.2. For example, a Galois connection between two exception arrow transformers $\mathtt{ExceptT}_E(C)(A, B) = C(A, \mathtt{Error}\ E\ B)$ and $\widehat{\mathtt{ExceptT}}_{\widehat{E}}(C)(A, B) = C(A, \widehat{\mathtt{Error}}\ \widehat{E}\ B)$ has the type

$$\begin{aligned}\forall C, \widehat{C}.\, [\forall A, \widehat{A}, B, \widehat{B}.\, \mathcal{G}(\mathcal{P}A, \widehat{A}) \times \mathcal{G}(\mathcal{P}B, \widehat{B}) \to \mathcal{G}(C(A, B), \widehat{C}(\widehat{A}, \widehat{B}))] \\ \to [\forall A, \widehat{A}, B, \widehat{B}.\, \mathcal{G}(\mathcal{P}A, \widehat{A}) \times \mathcal{G}(\mathcal{P}B, \widehat{B}) \to \mathcal{G}(\mathtt{Except}_E(C)(A, B), \mathtt{Except}_{\widehat{E}}(\widehat{C})(\widehat{A}, \widehat{B}))]\,.\end{aligned}$$

To construct this Galois connection, we extend the Galois connections for domain and codomain to include the extra data of the exception arrow transformer. From a Galois connection (α_E, γ_E) between the exception types and a Galois connection (α_B, γ_B) between the codomains, we can easily derive Galois connections $(\alpha_{\mathtt{Error}}, \gamma_{\mathtt{Error}}) \in \mathcal{G}(\mathcal{P}(\mathtt{Error}\ E\ B), \widehat{\mathtt{Error}}\ \widehat{E}\ \widehat{B})$ for the codomain of the exception transformers. Then from the Galois connection (α_C, γ_C) between the underlying arrows and (α_A, γ_A) between the domains, we construct the Galois connection of the underlying exception transformer types:

$$\alpha_{\mathtt{ExceptT}} = \alpha_C((\alpha_A, \gamma_A), (\alpha_{\mathtt{Error}}, \gamma_{\mathtt{Error}})) \qquad \gamma_{\mathtt{ExceptT}} = \gamma_C((\alpha_A, \gamma_A), (\alpha_{\mathtt{Error}}, \gamma_{\mathtt{Error}}))$$

Importantly, all Galois connections of analysis components have the same type shown in Definition 3.1. This allows us to compose these Galois connections with regular function composition. This becomes important, when we compose analysis components, which we discuss in Section 4. With Galois connections between arrow transformers, we can develop the soundness proposition of analysis components.

3.2 Soundness of Analysis Components

In this subsection, we describe how to prove soundness of analysis components, and what soundness means exactly. In particular, we develop a theory for analysis components and their soundness proofs that allows us to express soundness of arrow operations of arbitrary arity and type. To this end, we first describe our analysis components more formally.

An analysis component consists of a type class describing the interface of the component and two instances for two arrow transformers. Type classes and their instances can be described by algebras for a functor [Hamana and Fiore 2011]. The functor describes the codomain of each operation of the type class. For arrow type classes, this functor has type $\mathbf{Set}^{\mathcal{U}\times\mathcal{U}} \to \mathbf{Set}^{\mathcal{U}\times\mathcal{U}}$, i.e., it maps arrow types to arrow types. For example, we can describe the type class `ArrowExcept` in Figure 2 with the functor

$$\mathtt{ArrowExcept}_E(C)(X,Y) = [X \equiv E] + [C(X,Y) \times C(X \times E, Y)]\,.$$

The first operand of the coproduct describes the type of `throw` and the second operand the arguments of `catch`. An algebra over a functor F is a function of type $\forall X, Y.\, F(C)(X,Y) \to C(X,Y)$. This function combines all operations of the type class. In case of `ArrowExcept`, the algebra combines a computation $C(E,Y)$ for `throw` and a function $C(X,Y) \times C(X \times E, Y) \to C(X,Y)$ for `catch`. In addition, the functor is parameterized by other arguments of the type class. For example, the functor $\mathtt{ArrowExcept}_E$ is parameterized by the type of exceptions E.

With this theory, we define our analysis components formally as follows:

Definition 3.3 (Analysis Component). An analysis component $\langle f, \widehat{f}\rangle : \langle T, \widehat{T}\rangle_F$ is a pair of algebras $\langle f, \widehat{f}\rangle$ over a functor $F : \mathbf{Set}^{\mathcal{U}\times\mathcal{U}} \to \mathbf{Set}^{\mathcal{U}\times\mathcal{U}}$, where $\langle f, \widehat{f}\rangle$ is a pair of functions $f_C : F(T(C))(X,Y) \to T(C)(X,Y)$ and $\widehat{f}_{\widehat{C}} : F(\widehat{T}(\widehat{C}))(X,Y) \to \widehat{T}(\widehat{C})(X,Y)$.

In this definition, F defines the interface of the analysis components, f and $\widehat{f}$ implement the interface for arrow transformers T and $\widehat{T}$.

This formal definition of analysis components allows us to define their soundness proposition precisely:

Definition 3.4 (Soundness of Analysis Components). Given an analysis component $\langle f, \widehat{f}\rangle : \langle T, \widehat{T}\rangle_F$ and a Galois connection for the arrow transformers of this analysis component, then the analysis component $\langle T, \widehat{T}\rangle_F$ is sound iff all operations of F are pair-wise sound in f and $\widehat{f}$ according to the Galois connection. More formally, let $F = F_1 + \ldots + F_n$ be the functor representing a type class, $f = f_1 \ldots f_n$ be the algebra representing the concrete operations, and $\widehat{f} = \widehat{f}_1 \ldots \widehat{f}_n$ be the algebra representing the abstract operations. Then f is sound w.r.t. $\widehat{f}$ iff

$$\forall x, \widehat{x}. \quad \alpha_{F(T)}(x) \sqsubseteq \widehat{x} \implies \alpha_T(f_i(x)) \sqsubseteq \widehat{f}_i(\widehat{x}) \quad \text{for all } 1 \le i \le n.$$

In other words, an analysis component is sound if each operation preserves soundness of their arguments. For example, in the case of the `ArrowExcept` component in Figure 2 we have to prove the following two lemmas.

$$\alpha(\mathtt{throw}) \sqsubseteq \widehat{\mathtt{throw}}$$
$$\forall f, \widehat{f}, g, \widehat{g}. \quad \alpha(f,g) \sqsubseteq (\widehat{f}, \widehat{g}) \implies \alpha(\mathtt{catch}(f,g)) \sqsubseteq \widehat{\mathtt{catch}}(\widehat{f}, \widehat{g})$$

That is, we prove that `throw` is sound w.r.t. $\widehat{\mathtt{throw}}$ and $\mathtt{catch}(f,g)$ w.r.t. $\widehat{\mathtt{catch}}(\widehat{f}, \widehat{g})$ given sound continuations $f, \widehat{f}, g, \widehat{g}$. In contrast to Keidel et al. [2018], these soundness lemmas for the `ArrowExcept` component are reusable because of the following reasons.

- The operations are defined over arrow transformers $\mathtt{Except}_E(C)$ and $\mathtt{Except}_{\widehat{E}}(\widehat{C})$ and the proofs are universal in the underlying arrows C and $\widehat{C}$, which allows us to swap out the underlying arrows when we compose this component.

- The proofs are universal in the exception types E and $\widehat{E}$, which allows us to use this component and proofs in languages with different exception types.

But how do we actually prove these lemmas if we do not know the underlying arrows C and $\widehat{C}$? We need to establish a base-line, which allows us to reason about generic arrows in these soundness proofs. Fortunately, arrows already provide a basic reasoning tool-kit: the algebraic arrow laws [Hughes 2000]. To illustrate how such a proof works, we include proofs in the supplementary material accompanying this paper. These proofs show that it is feasible to reason about soundness of arrow operations over arrow transformers without knowing the underlying arrows C and $\widehat{C}$. The only assumptions we had to make about the arrow operations of C and $\widehat{C}$, was they are sound, monotone and obey the arrow laws in Appendix A. We demonstrate in the following section how these assumptions are preserved under composition of components.

To summarize, in this section, we developed a generic theory to prove soundness of analysis components once and for all. These soundness proofs are reusable, because they are specific to arrow transformers rather than specific to monolithic arrows. In the following section, we explain a different way to define sound analysis components from existing analysis components.

4 SOUND COMPOSITION OF ANALYSIS COMPONENTS

In Sections 2.2 and 2.3 we showed that we need to *compose* analysis components to combine their effects and to explain how their effects interact. In this section, we describe three different ways for composing analysis components and prove them sound.

4.1 Horizontal Composition

The simplest way of composition occurs when the arrow transformers of a component implement multiple interfaces $\langle T, \widehat{T}\rangle_F$ and $\langle T, \widehat{T}\rangle_G$. For example, the transformers `ExceptT` and $\widehat{\texttt{ExceptT}}$ defined in Figure 2 do not support a `finally f g` operation that executes `g` no matter if `f` succeeds or fails. We capture this operation in a new interface:

```
class ArrowFinally c where finally :: c x y → c x () → c x y
```

We implement this operation in another component $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}}\rangle_{\texttt{ArrowFinally}}$ with the same arrow transformers. We horizontally compose $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}}\rangle_{\texttt{ArrowExcept}}$ with the component $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}}\rangle_{\texttt{ArrowFinally}}$ to obtain the functionality of both components in a new component $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}}\rangle_{\texttt{ArrowExcept+ArrowFinally}}$. More formally:

Definition 4.1 (Horizontal Composition). The horizontal composition $\langle T, \widehat{T}\rangle_F \oplus \langle T, \widehat{T}\rangle_G$ of two analysis components $\langle f, \widehat{f}\rangle : \langle T, \widehat{T}\rangle_F$ and $\langle g, \widehat{g}\rangle : \langle T, \widehat{T}\rangle_G$ is defined as $\langle f + g, \widehat{f} + \widehat{g}\rangle : \langle T, \widehat{T}\rangle_{F+G}$.

Furthermore, this composition preserves soundness of components:

Theorem 4.2 (Horizontal composition preserves soundness). *Given sound analysis components $\langle T, \widehat{T}\rangle_F$ and $\langle T, \widehat{T}\rangle_G$, their horizontal composition $\langle T, \widehat{T}\rangle_F \oplus \langle T, \widehat{T}\rangle_G$ is sound.*

Proof. Follows directly by Theorem 3.4, because we can separately prove soundness of each operation in the interface F and in G. □

To summarize, horizontal composition allows us to compose components with the same arrow transformers that implement different interfaces.

4.2 Component Lifting

In general, analysis components use different arrow transformers to implement different interfaces $\langle T, \widehat{T}\rangle_F$ and $\langle U, \widehat{U}\rangle_G$. We detail how to compose such components using *vertical composition* in

the next subsection. Vertical composition means that we stack one component on top of the other, effectively wrapping the nested component. Here, we discuss an important preliminary for vertical composition, namely the lifting of operations of the nested component through the wrapping.

To compose components vertically, we need to compose their arrow transformers: $T \circ U = \forall C.\, T(U(C))$. Similar to *monad transformers* [Liang et al. 1995, Section 8], to make the operations of U available for the composed arrow transformers $T \circ U$, we need to *lift* them through T. The reason is that we cannot interact with the inner arrow transformer U directly; all interaction with U has to go through T.

For example, to make the `ArrowExcept` operations defined by `ExceptT` available in `StoreT∘ExceptT`, we need to lift `throw` and `catch` through `StoreT`. This lifting explains how store passing interacts with exception handling. To this end, we have to implement a lifting instances that explains how and when `StoreT` provides `ArrowExcept` operations:

```
instance ArrowExcept e (StoreT (ExceptT e c)) where
  throw = Store (proc (_,e) → throw ≺ e)
  catch (Store f) (Store g) = Store $ catch f (proc ((s,x),e) → g ≺ (s,(x,e)))
```

This instance allows `Store` to provide `ArrowExcept` operations whenever `Store` is applied to `Except`. The lifting then delegates to the operations of the nested `Except` transformer.

But this lifting is not reusable, because it is coupled to the composition `StoreT ∘ ExceptT`. If we want to replace one of the transformers or if there is another transformer in between `StoreT` and `ExceptT`, the lifting fails to work. Therefore, we generalize the lifting definition to precisely capture when `StoreT` can provide `ArrowExcept` operations:

```
instance ArrowExcept e c ⇒ ArrowExcept e (StoreT c)
```

That is, `StoreT` provides `ArrowExcept` operations whenever the underlying arrow `c` provides `ArrowExcept` operations. The implementation of the operations stays the same. This lifting is more reusable because it is neither coupled to the arrow transformer `Except` nor its position in the transformer stack.

Formally, a lifting of operations in $\langle U, \widehat{U}\rangle_F$ through the transformers $\langle T, \widehat{T}\rangle$ corresponds to a pair of functions $\langle \delta_U, \widehat{\delta_{\widehat{U}}}\rangle : \langle U, \widehat{U}\rangle_F \to \langle T \circ U, \widehat{T} \circ \widehat{U}\rangle_F$, where δ_U lifts the concrete part of the component and $\widehat{\delta_{\widehat{U}}}$ the abstract part of the component. As discussed above, to make this lifting reusable, δ_U needs to be parametric in U and $\widehat{\delta_{\widehat{U}}}$ parametric in $\widehat{U}$.

A lifting of components is sound if the functions $\langle \delta, \widehat{\delta}\rangle$ preserve soundness:

Definition 4.3 (Soundness of Component Liftings). A lifting $\langle \delta, \widehat{\delta}\rangle : \langle U, \widehat{U}\rangle_F \to \langle T \circ U, \widehat{T} \circ \widehat{U}\rangle_F$ is sound iff the component $\langle \delta(f), \widehat{\delta}(\widehat{f})\rangle : \langle T \circ U, \widehat{T} \circ \widehat{U}\rangle_F$ is sound for all sound components $\langle f, \widehat{f}\rangle : \langle U, \widehat{U}\rangle_F$.

In general, each lifting has to be shown sound separately. In particular, because liftings are not unique for a pair of arrow transformers, we cannot formulate a generic soundness theorem. However, in many cases we obtain a proof with less or no effort, which we discuss in the following.

Reusable liftings. As described above, we can make liftings reusable by abstracting over the underlying arrow and only specifying minimal requirements. We use this technique extensively to limit the number of lifting instances and soundness arguments needed.

Generic liftings. First-order operations of type `c x y` often can be lifted with a generic `lift` operation:

```
class ArrowLift t c where
  lift :: c x y → t c x y
```

For example, the `throw` operation that throws an exception can be lifted through the `Store` arrow transformer with this generic `lift` operation:

```
instance ArrowExcept e c ⇒ ArrowExcept e (StoreT c) where
  throw = lift throw
```

It suffices to show the soundness of the generic `lift` operation to ensure all its use cases are sound.

Derivable liftings. Often concrete and abstract arrow transformer are implemented with the same arrow transformer. For example, the `StoreT` and $\widehat{\texttt{StoreT}}$ arrow transformers are both implemented with the `StateT` arrow transformer:

```
newtype StateT s c x y = StateT (c (s,x) (s,y))
newtype StoreT c = StoreT (StateT Store c x y)
newtype StoreT^ c = StoreT^ (StateT Store c x y)
```

In this case, a lifting for the `StoreT` and $\widehat{\texttt{StoreT}}$ arrow transformers can be derived automatically by Haskell [Gibbons 2010] from the lifting defined on the underlying `StateT` arrow transformer.

```
deriving instance ArrowExcept e (StoreT c)
deriving instance ArrowExcept e (StoreT^ c)
```

Furthermore, the liftings for both arrow transformers share the same implementation and all differences of concrete and abstract store type are universally quantified. Therefore, soundness of this component lifting follows as a free theorem of parametricity [Keidel et al. 2018, Theorem 5].

We evaluate how many of these liftings fall into either of these categories in Section 7.

To summarize, to compose two analysis components with differing arrow transformers, we need to *lift* the operations of the inner arrow transformers through the outer arrow transformers. Such a lifting is sound if it preserves soundness of the underlying component. In general, soundness of these liftings needs to be proven manually, however, often we obtain a soundness proof with less or no effort if we can reuse the same lifting operation or share the implementation of the lifting. Equipped with component lifting, we can support the vertical composition of analysis components.

4.3 Vertical Composition of Analysis Components

In the remainder of this section, we discuss how to combine independent analysis components $\langle T, \widehat{T}\rangle_F$ and $\langle U, \widehat{U}\rangle_G$ using vertical composition. Our goal is to obtain a new analysis component that implements both interfaces F and G based on the functionality of all involved arrow transformers. The key idea of vertical composition is to first lift one component and then to use horizontal lifting on the result.

For example, to obtain an analysis for store passing and exception handling, we compose the components $\langle \texttt{StoreT}, \widehat{\texttt{StoreT}}\rangle_{\texttt{ArrowStore}}$ and $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}}\rangle_{\texttt{ArrowExcept}}$. The order in which we compose these components matters. For example, the order $\texttt{StoreT} \circ \texttt{ExceptT}$ determines that store updates are reset while the order $\texttt{ExceptT} \circ \texttt{StoreT}$ determines that store updates propagate when an exception occurs.

The composition $\langle \texttt{StoreT}, \widehat{\texttt{StoreT}}\rangle_{\texttt{ArrowStore}} \circ \langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}}\rangle_{\texttt{ArrowExcept}}$ of these two components requires a combination of techniques we presented in the previous two subsections:

(1) We lift the operations of $\langle \texttt{ExceptT}, \widehat{\texttt{ExceptT}}\rangle_{\texttt{ArrowExcept}}$ through $\langle \texttt{StoreT}, \widehat{\texttt{StoreT}}\rangle$ to obtain a component $\langle \texttt{StoreT} \circ \texttt{ExceptT}, \widehat{\texttt{StoreT}} \circ \widehat{\texttt{ExceptT}}\rangle_{\texttt{ArrowExcept}}$.
(2) We specialize the generic arrow transformer types of $\langle \texttt{StoreT}, \widehat{\texttt{StoreT}}\rangle_{\texttt{ArrowStore}}$ to obtain a component $\langle \texttt{StoreT} \circ \texttt{ExceptT}, \widehat{\texttt{StoreT}} \circ \widehat{\texttt{ExceptT}}\rangle_{\texttt{ArrowStore}}$.

(3) Finally, we horizontally compose the components $\langle \mathsf{StoreT} \circ \mathsf{ExceptT}, \widehat{\mathsf{StoreT}} \circ \widehat{\mathsf{ExceptT}} \rangle_{\mathsf{ArrowExcept}}$ and $\langle \mathsf{StoreT} \circ \mathsf{ExceptT}, \widehat{\mathsf{StoreT}} \circ \widehat{\mathsf{ExceptT}} \rangle_{\mathsf{ArrowStore}}$ to obtain a component with the operations of both interfaces: $\langle \mathsf{StoreT} \circ \mathsf{ExceptT}, \widehat{\mathsf{StoreT}} \circ \widehat{\mathsf{ExceptT}} \rangle_{\mathsf{ArrowStore+ArrowExcept}}$.

The lifting in this composition is *glue code* which describes how the two components interact.

More formally, we define vertical composition of analysis components with glue code as follows:

Definition 4.4 (Vertical Composition). The vertical composition $\langle T, \widehat{T} \rangle_F \circ_\Delta \langle U, \widehat{U} \rangle_G$ of two analysis components $\langle f, \widehat{f} \rangle : \langle T, \widehat{T} \rangle_F$ and $\langle g, \widehat{g} \rangle : \langle U, \widehat{U} \rangle_G$ and a lifting $\Delta = \langle \delta, \widehat{\delta} \rangle : \langle U, \widehat{U} \rangle_G \to \langle T \circ U, \widehat{T} \circ \widehat{U} \rangle_G$ is defined as

$$\langle T, \widehat{T} \rangle_F \circ_\Delta \langle U, \widehat{U} \rangle_G : \langle T \circ U, \widehat{T} \circ \widehat{U} \rangle_{F+G}$$
$$\langle f, \widehat{f} \rangle \circ_\Delta \langle g, \widehat{g} \rangle = \langle f, \widehat{f} \rangle \oplus (\Delta \langle g, \widehat{g} \rangle) = \langle f + \delta(g), \widehat{f} + \widehat{\delta}(\widehat{g}) \rangle.$$

That is, we lift the operations of $\langle U, \widehat{U} \rangle_G$ through $\langle T, \widehat{T} \rangle$ and horizontally compose the resulting components. This brings us to the main soundness theorem for the composition of analysis components.

Theorem 4.5 (Vertical composition preserves soundness). *Given sound analysis components $\langle T, \widehat{T} \rangle_F$ and $\langle U, \widehat{U} \rangle_G$ and a sound lifting $\Delta : \langle U, \widehat{U} \rangle_G \to \langle T \circ U, \widehat{T} \circ \widehat{U} \rangle_G$, then the vertical composition $\langle T, \widehat{T} \rangle_F \circ_\Delta \langle U, \widehat{U} \rangle_G$ is sound.*

Proof. The lifted component $\langle T \circ U, \widehat{T} \circ \widehat{U} \rangle_G$ is sound because the lifting Δ preserves soundness (Theorem 4.3) and its input $\langle U, \widehat{U} \rangle_G$ is sound. Furthermore, the specialized component $\langle T \circ U, \widehat{T} \circ \widehat{U} \rangle_F$ is sound because $\langle T, \widehat{T} \rangle_F$ is parametric in the underlying arrow. Then by Theorem 4.2 the horizontal composition $\langle T \circ U, \widehat{T} \circ \widehat{U} \rangle_F \oplus \langle T \circ U, \widehat{T} \circ \widehat{U} \rangle_G$ is sound and hence $\langle T, \widehat{T} \rangle_F \circ_\Delta \langle U, \widehat{U} \rangle_G$ is sound. □

To summarize, in this section we discussed how to soundly compose analysis components to obtain a complete analyses. The composition of two components with differing arrow transformers and different interfaces requires some glue code that explains how the effects of these components interact. As for the components themselves, the definition and soundness proof of glue code can be reused, facilitating the easy construction of provably sound static analyzers.

5 SOUNDNESS OF COMPONENT-BASED STATIC ANALYSES

The focus in the paper so far has been on analysis components themselves. However, analysis components alone do not describe complete static analyses. In this section, we describe how to use analysis components to define complete static analyses. Finally, we prove that any static analysis, that is based on sound analysis components, is sound.

To use analysis components to describe a static analysis, we need to describe the semantics of the analyzed language with an arrow-based *generic interpreter* [Keidel et al. 2018] that captures the similarities of concrete and abstract semantics. For example, Listing 2 and Listing 1 in Section 7 show the generic interpreters for PCF and a While language. A generic interpreter does not describe the concrete semantics nor a particular abstract semantics. Instead it is a template of the language semantics, that we need to instantiate with suitable arrow instances to obtain the concrete semantics or a particular abstract semantics

To instantiate a generic interpreter with an analysis component, we first compose an analysis component $\langle \mathsf{ConT}, \widehat{\mathsf{AbsT}} \rangle_{\mathtt{I}}$ which matches the interface $\mathtt{I}$ of the generic interpreter. However, we cannot instantiate the generic interpreter with $\langle \mathsf{ConT}, \widehat{\mathsf{AbsT}} \rangle$ directly because the generic interpreter expects *arrows*, where the analysis component $\langle \mathsf{ConT}, \widehat{\mathsf{AbsT}} \rangle$ consists of *arrow transformers*. To obtain a pair of arrow instances, we apply the analysis component $\langle \mathsf{ConT}, \widehat{\mathsf{AbsT}} \rangle$ to a pair of base

arrows $\langle \mathcal{P}\_ \to \mathcal{P}\_,\ \_ \to \_ \rangle$. From this application we get the collecting semantics [Cousot 1999] $\mathsf{ConT}\ (\mathcal{P}\_ \to \mathcal{P}\_)$ of the concrete interpreter and the abstract function space $\widehat{\mathsf{AbsT}}\ (\to)$ of the abstract interpreter. More importantly, the abstract interpreter $\mathsf{run}_{\widehat{\mathsf{AbsT}}\ (\to)}$ soundly approximates the concrete collecting semantics $\mathsf{run}_{\mathsf{ConT}\ (\mathcal{P}\_ \to \mathcal{P}\_)}$, which we prove below.

In fact, *any* generic interpreter instantiated with *any* sound analysis component with a compatible interface is sound, which leads us to our main soundness theorem:

Theorem 5.1 (Soundness of component-based analyses). *Let* $eval_C : \mathcal{I}(C) \Rightarrow C(X, Y)$ *be a generic interpreter with an arrow-based interface* $\mathcal{I}$*. Furthermore, let* $\langle ConT, \widehat{AbsT} \rangle_{\mathcal{I}}$ *be a sound analysis component. Then the abstract semantics* $eval_{\widehat{AbsT}(\to)}$ *is sound with respect to the concrete collecting semantics* $eval_{ConT(\mathcal{P}\_ \to \mathcal{P}\_)}$.

Proof. The soundness lemma of the analysis component $\langle ConT, \widehat{AbsT} \rangle_{\mathcal{I}}$ (Definition 3.4) guarantees that the pair of arrow instances $ConT\ (\mathcal{P}\_ \to \mathcal{P}\_)$ and $\widehat{AbsT}\ (\to)$ are sound. Furthermore, the main soundness theorem for arrow-based generic interpreters [Keidel et al. 2018, Theorem 3] guarantees that the generic interpreter $eval_{\widehat{AbsT}(\to)}$ is sound w.r.t. $eval_{ConT(\mathcal{P}\_ \to \mathcal{P}\_)}$. □

Note that the arrows $\mathcal{P}\_ \to \mathcal{P}\_$ and $(\to)$ implement the `Arrow` and `ArrowChoice` type classes. This requires one extra sound lifting of these operations through the $ConT$ and $\widehat{AbsT}$ arrow transformers. Theorem 5.1 can be easily extended to account for this lifting, which we omitted for a cleaner presentation.

To summarize, in this section we have shown how to define a complete static analysis based on analysis component and how to prove it sound. This requires a generic interpreter for the analyzed language, which captures the similarities of concrete and abstract interpreter. We proved that this generic interpreter instantiated with a sound analysis component is sound.

6 STURDY: A LIBRARY OF SOUND AND REUSABLE ANALYSIS COMPONENTS

We developed the *Sturdy* library of 13 sound and reusable arrow-based analysis components in Haskell.[1] We use some of these components to implement two static analyses in Section 7, to demonstrate the reusability of these components. In this section, we briefly describe selected components and then discuss measures to counter their performance overhead.

6.1 Analysis Components

Single Transformer Components A good source for components are single arrow transformers that are used both for the concrete and the abstract interpreter. For example, the arrow transformer `ReaderT r c` adds data of type `r` to the input of the arrow computation `c`. It can be easily turned into an analysis component $\langle \mathsf{ReaderT}, \mathsf{ReaderT} \rangle_{\mathsf{ArrowReader}}$ that adds data to both the concrete and abstract interpreter. Furthermore, because the concrete and abstract implementation of the `ArrowReader` operations is exactly the same, only differing in the type of data `r`, and hence can be proved sound trivially with a soundness theorem for parametricity [Keidel et al. 2018, Theorem 5].

This way we defined 5 analysis components for reading and writing state, for constant data, and for continuation-passing style.

$\langle \mathsf{ReaderT}, \mathsf{ReaderT} \rangle_{\mathsf{ArrowReader}}$ $\langle \mathsf{StateT}, \mathsf{StateT} \rangle_{\mathsf{ArrowState}}$ $\langle \mathsf{WriterT}, \mathsf{WriterT} \rangle_{\mathsf{ArrowWriter}}$
$\langle \mathsf{ConstT}, \mathsf{ConstT} \rangle_{\mathsf{ArrowConst}}$ $\langle \mathsf{ContT}, \mathsf{ContT} \rangle_{\mathsf{ArrowCont}}$

[1] https://gitlab.rlp.net/plmz/sturdy/

These arrow transformers are well-known and we borrowed their definition from the arrow transformer libraries `arrows` and `atl` on Hackage.[2] Furthermore, some of these arrow transformers appeared in form of monad transformers in [Darais et al. 2017, 2015], which we took inspiration from.

Environment Components To implement environment components, we created the type class `ArrowEnv` with an operation `getEnv` to fetch an environment, `localEnv` to set a new environment in a local context, `extendEnv` to extend the given environment with a new binding and `lookup` to look up a binding in the current environment:

```
class ArrowEnv var val env c where
 getEnv :: c () env                 localEnv :: c x y → c (env,x) y
 extendEnv :: c (var,val,env) env   lookup :: c (val,x) y → c x y → c (var,x) y
```

Based on this interface, we created two components for environments. The first component $\langle \mathsf{EnvT}, \widehat{\mathsf{EnvT}} \rangle_{\mathsf{ArrowEnv}}$ implements the standard abstraction for environments [Cousot 1999], i.e., a mapping from variables to abstract values.

The second component $\langle \mathsf{EnvT}, \widehat{\mathsf{BoundedEnvT}} \rangle_{\mathsf{ArrowEnv}}$ implements a finite abstraction for environments for languages with closures [Shivers 1991]. In this component abstract environments consist of a pair of mappings $(\mathtt{Map\ Var}\ \widehat{\mathtt{Addr}}, \mathtt{\ Map}\ \widehat{\mathtt{Addr}}\ \widehat{\mathtt{Val}})$ from variables to abstract addresses to values. By limiting the amount of abstract addresses, the abstract domain of environments and abstract closures becomes finite.

All arrow transformers of the environment components are implemented with the `ReaderT` arrow transformer. This gives us the soundness proof for `getEnv` and `localEnv` proofs for free [Keidel et al. 2018] because they are implemented with the same `ArrowReader` operations.

Store Components To implement store components, we created the type class `ArrowStore` with operations to read from and write to a store:

```
class ArrowStore var val c where
  read :: c (val,x) y → c x y → c (var,x) y
  write :: c (var,val) ()
```

Based on this interface we defined a store component $\langle \mathsf{StoreT}, \widehat{\mathsf{StoreT}} \rangle_{\mathsf{ArrowStore}}$, which implements a path-insensitive store abstraction. The abstract store is a mapping from variables to abstract values. Each binding indicates if the binding *must* be present in the store or *may* not be present in the store. When we read a *may*-binding, the operation $\widehat{\mathtt{read}}\ \mathtt{f\ g}$ joins results of the success and failure continuations `f` and `g`.

Furthermore, we implemented a component $\langle \mathsf{ReachingDefsT}, \widehat{\mathsf{ReachingDefsT}} \rangle_{\mathsf{ArrowStore}}$ for tracking reaching definitions [Nielson et al. 1999], which uses the store interface. This component calculates which variable definitions reach a certain program point without being overwritten. We implemented this analysis as a lifting of the store operations, by recording the label of the current assignment alongside the value in the store in the abstract run. After the analysis has run, we read out the store at each program point from the fixpoint cache to obtain the reaching definition information.

Failure and Exception Components For failure and exceptions, we created two components $\langle \mathsf{FailureT}, \widehat{\mathsf{FailureT}} \rangle_{\mathsf{ArrowFail}}$ and $\langle \mathsf{ExceptT}, \widehat{\mathsf{ExceptT}} \rangle_{\mathsf{ArrowExcept}}$ that employ two different abstractions for error.

The abstract $\widehat{\mathtt{FailureT}}$ transformer wraps the output with an error type in which $\mathtt{Success\ x} \sqsubseteq \mathtt{Fail\ e}$. With this ordering, erroneous branches of computation overwrite successful branches

[2] http://hackage.haskell.org/

of computation: `Fail e` $\sqcup$ `Success x = Fail e`. This abstraction is useful when we want to propagate information about failures in programs.

The abstract $\widehat{\texttt{ExceptT}}$ transformer wraps the output with an error type (Figure 2) in which `Success x` $\sqsubseteq$ `SuccessOrFail x e` $\sqsupseteq$ `Fail e`. This abstraction is more precise than the error type of `FailureT`, because the case `Success` describes computation that must succeed and cannot fail.

Fixpoint Components To implement fixpoint components, we created a type class with a `fix` operation that calculates the fixpoint over an arrow computation:

```
class ArrowFix x y c where
  fix :: (c x y → c x y) → c x y
```

Based on this interface we implemented a fixpoint component $\langle \texttt{Fix}, \widehat{\texttt{Fix}} \rangle_{\texttt{ArrowFix}}$, whose concrete `fix` operation calculates the standard fixpoint `fix f = f (fix f)`. The abstract `fix` operation implements a parallel/sequential fixpoint algorithm [Darais et al. 2017]. We parameterized this fixpoint algorithm by a widening operator [Cousot and Cousot 1992] for the codomain `y` that ensures that the fixpoint iteration terminates and a second operator on the domain `x` that joins recursive calls to avoid infinitely deep chains of recursive calls. The fixpoint component $\langle \texttt{Fix}, \widehat{\texttt{Fix}} \rangle$ is the only component which does *not* consist of arrow transformers and hence has to be placed at the bottom of the component stack. This ensures that the function `f` we fix over is a pure function and no side effects interfere when we iterate on `f` multiple times.

Additionally, the abstract $\widehat{\texttt{fix}}$ operation detects computations which potentially do not terminate. Non-terminating computations are usually represented with the bottom element $\bot$ of the abstract domain for values and add a lot of boilerplate to the abstract interpreter. Instead, we capture the bottom element with a component $\langle \texttt{TerminatingT}, \widehat{\texttt{TerminatingT}} \rangle$ that wraps the output with a `Maybe`-like type:

```
data Terminating a = NonTerminating | Terminating a
```

This transformer allows us to remove the bottom element from the abstract domain of values and the boilerplate of propagating bottom values.

Based on the same `ArrowFix` interface we implemented a component $\langle \texttt{ContourT}, \widehat{\texttt{ContourT}} \rangle$ that tracks the *call context* of the abstract interpreter. This call context consists of a list of recursive calls to the abstract interpreter and is useful, for example, in a k-CFA analysis [Shivers 1991]. We implement this component as a lifting of the `fix` operation:

```
fix f = ContourT $ proc (δ,x) → fix (runContourT ⌊x : δ⌋_k ∘ f ∘ ContourT) ↢ x
```

On each recursive call we push the argument `x` of the abstract interpreter onto the current call string δ and limit it size to at most k.

6.2 Reducing the Performance Overhead of Analysis Components

Every analysis component adds some overhead to the runtime of the analysis. We identified two main sources for this performance overhead: Inefficient arrow code and dynamic dispatch. In the rest of this section, we explain how we addressed these issues to reduce the performance overhead of analysis components in our library.

The first issue was an inefficient pattern of arrow code that occured frequently in the implementation of arrow transformers: The composition of a pure function `f` with an effectful arrow computation `g` as in `g ∘ arr f`. The problem with this pattern is that the composition operation "∘" does not know that `arr f` is a pure computation and hence has to consider all possible effects of both operations. For example, if the arrow type supports exceptions, the composition operation "∘" has to check if an exception occured in `arr f`, even though `arr f` cannot cause an exception. Fortunately, we can eliminate this inefficient pattern by using a type class which captures the

Transformer	No Opts.	Profunctor		Inline		Prof. + Inline	
`ConstT`	261 μs	233 μs	(1)	5 μs	(56)	5 μs	(56)
`ReaderT`	907 μs	874 μs	(1)	8 μs	(119)	8 μs	(117)
`StateT`	435 μs	433 μs	(1)	13 μs	(33)	13 μs	(33)
`WriterT`	1506 μs	1490 μs	(1)	13 μs	(118)	13 μs	(119)
`ErrorT`	664 μs	664 μs	(1)	14 μs	(48)	14 μs	(47)
`ExceptT`	827 μs	804 μs	(1)	15 μs	(56)	15 μs	(55)
`TerminatingT`	515 μs	502 μs	(1)	14 μs	(38)	14 μs	(37)
Stack	209408 μs	18085 μs	(12)	27 μs	(7730)	26 μs	(7978)

Fig. 3. Benchmark results for individual arrow transformers without optimizations and with the profunctor and inlining optimization. Each column shows the average runtime in microseconds and in parentheses the speed up compared to the unoptimized version. The bottom row shows benchmark results for an arrow transformer stack which combines all transformers above.

composition of pure functions with effectful computations. The `Profunctor` type class[3] defines an operation `dimap` which pre- and post-composes two pure functions with an effectful computation:

```
dimap :: (x → x') → (y → y') → c x' y → c x y'
```

The implementation of `dimap` for arrows is more efficient than effectful composition, because it can exploit that the functions it composes with are pure. For example, in contrast to `g ∘ arr f`, the operation `dimap f id g` does not have to check that `f` cause an exceptions.

As a second source of performance overhead we identified the dynamic dispatch of type class methods. Without any optimization options the GHC Haskell compiler, converts type classes to dictonaries (records of functions) [Hall et al. 1996]. Calling functions from these dictonaries entails a dynamic dispatch, which causes a performance overhead. We counter this issue by annotating the arrow type class methods with `INLINE`, such that GHC retains a copy of the source code of the type class methods. Furthermore, we annotated the complete type of the generic interpreter in the file where we compose an analysis. This allows GHC to specialize the definition of the generic interpreter, inline arrow operations and eliminate any form of dynamic dispatch. Furthermore, because all arrow operations are inlined, GHC can optimize some redundent pre- and post-processing in arrow transformer liftings.

We evaluated how these two optimizations affect the performance of arrow transformers with a benchmark.[4] The benchmark instantiates an arrow-based concrete interpreter with different arrow transformers and measures the runtime for an example program. Figure 3 shows the runtimes for each transformer in microseconds with and without optimizations. The results show that the profunctor optimization does not significantly improve the performance of individual transformers, but it does improve the performance by 12x if multiple transformers are combined into a stack. The inlining optimization improves the performance of individual transformers and the transformer stack by several orders of magnitude. Lastly, combining the profunctor and inlining optimization does not significantly improve the performance over just the inlining optimization.

To summarize, in this section we presented a library of analysis components for different analysis concerns. Furthermore, we described two techniques that we used to reduce the performance overhead of arrow transformers. In Section 7, we will use some of these components to implement static analyses, to demonstrate their reusability.

[3] http://hackage.haskell.org/package/profunctors
[4] https://gitlab.rlp.net/plmz/sturdy/blob/benchmark/lib/bench/ArrowTransformerBench.hs

7 EXPERIMENTAL EVALUATION AND CASE STUDIES

In this paper, we presented an approach to reduce the effort of defining and proving soundness of static analyses with the help of sound and reusable analysis components. To evaluate our approach, we answer the following research questions:

(RQ1) Modular implementation: Are analysis components reusable and do they compose?
(RQ2) Modular soundness proofs: Are analysis components separately verifiable and do their soundness proofs compose?
(RQ3) Liftings: Is the effort of implementing liftings and proving their soundness acceptable?

To answer these research questions, we conducted two experiments. The first experiment starts with an interval analysis for the While language. We explore how analysis components support modular analysis development and soundness proofs by building a reaching definitions analysis on top of the interval analysis. We then challenge our approach by extending the While language with exceptions and observe how the interval and reaching definitions analyses change.

In our second experiment we build a control-flow analysis (k-CFA) for PCF. The analysis predicts the control flow of calls to first-class function values, which is not statically decidable. The goal of this experiment is to test if our approach is specific to some languages or analyses.

Across both experiments we were able to answer our research questions affirmatively. In particular, the implementation and soundness proofs of most liftings comes for free.

7.1 Experiment 1: Analyses for a While Language

We implement an interval analysis for a statically-scoped While language. This interval analysis will serve as a base-line as we extend this analysis in two different ways to study the impact of these changes. First, we add a reaching definition [Nielson et al. 1999] analysis to the interval analysis. Second, we extend the While language with exceptions.

7.1.1 Interval Analysis for the While Language. We start by defining an arrow-based generic interpreter for our While language (Listing 1 in Appendix B). The interpreter is basically a more complete version of the one we presented in Section 2.4. To implement an interval analysis for this language, we need to instantiate the generic interpreter with a "super-component" that implements all required interfaces. Our first research question **RQ1** asks if we can separate concerns in the implementation of this super-component. Indeed, we were able to implement each interface in a separate component and to compose the super-component from these using the techniques described in Section 4.

We display the involved components and their composition in Figure 4, which introduces a novel notation we devised for this paper. Each box $\boxed{\text{n}}$ in the table indicates a separate analysis component $\langle Row_L, Row_R \rangle_{Col}$. The column label indicates the component's interface; the left and right row labels indicate the used concrete and abstract arrow transformers; the boxed number indicates how many operations of the interface had to be implemented. For example, box $\boxed{4}$ in Figure 4 indicates an analysis component $\langle \mathsf{EnvT}, \widehat{\mathsf{EnvT}} \rangle_{\mathsf{ArrowEnv}}$ that implements 4 operations.

Our notation in Figure 4 also displays how components are composed. Components that appear on the same row compose horizontally without any extra effort. Components that appear on different rows require vertical composition based on one or more liftings. We display liftings as upward arrows, yet distinguish two kinds: A straight arrow ↑ represents a lifting whose implementation and soundness proof was trivial because the lifting was (i) reusable, (ii) generic, or (iii) derivable (Section 4.2). In contrast, a squiggly arrow ⇝ represents a lifting that required a non-trivial implementation and soundness proof. Regarding the research questions, we conclude the following:

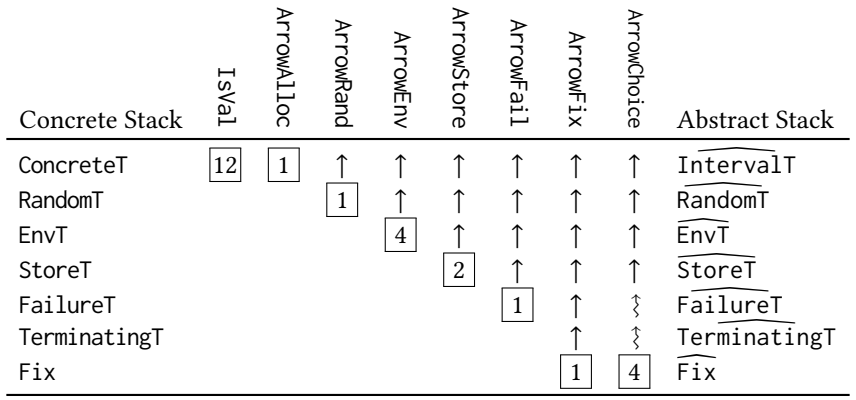

Fig. 4. Interval analysis of the WHILE language: Boxes n represent components, straight arrows ↑ represent trivial liftings, squiggly arrows ⇝ represent non-trivial liftings.

(RQ1) Modular implementation: We successfully separated concerns of the analysis into 7 analysis components: 6 reusable analysis components from our library and 1 language-specific analysis component ⟨ConcreteT, IntervalT⟩ for values, conditionals, and allocation.

(RQ2) Modular soundness proofs: We successfully decomposed the soundness proof into soundness lemmas about the individual components: Each soundness lemma proves a single concern of the analysis, while it is independent of other concerns. For example, the soundness proofs of the conditional `if_` in `IsVal` is independent of the store and fixpoint cache, even though these are threaded through the branches of the conditional. This is possible because the `IsVal` component is parametric in the underlying arrow `c`, which contains the store and fixpoint cache after composition.

(RQ3) Liftings: We required 22 liftings to compose all 7 analysis components. Of these liftings, 20 liftings are trivial. Only 2 liftings required an explicit implementation and soundness proof. We conclude that the effort for liftings is modest and acceptable.

7.1.2 Reaching Definitions Analysis for the WHILE Language. We want to refine our previous interval analysis to also keep track of reaching definitions [Nielson et al. 1999]. A definition (here: assignment) reaches another statement if there is at least one control-flow path where the assigned variable was not reassigned in between. Since the language syntax remains unchanged, no change occurs to the generic interpreter `run` or its required interfaces. The challenge is this: Can we reuse the implementation and soundness proofs of all previously used components unchanged?

(RQ1) Modular implementation: We encapsulate the concern of reaching definitions in its own analysis component ⟨ReachingDefsT, ReachingDefsT⟩$_{\texttt{ArrowStore}}$ as described in Section 6. Technically, the reaching definitions component piggybacks on another component implementing the `ArrowStore` interface, but stores additional data in the abstract run. In the concrete run, reaching definitions has no effect and uses the identity transformer. Figure 5 shows how the new component (gray background) neatly integrates with the existing components; no changes to other components were necessary.

(RQ2) Modular soundness proofs: We only had to prove soundness of the reaching definitions component, while all other soundness lemmas remain valid. Except for the reaching definitions component, there is no additional proof effort.

(RQ3) Liftings: Since the reaching definitions was realized as a non-trivial lifting, we additionally obtain 2 such liftings in our final composition. None of the other liftings were (or could have been) influenced. Thus, we retain that the lifting effort is modest and acceptable.

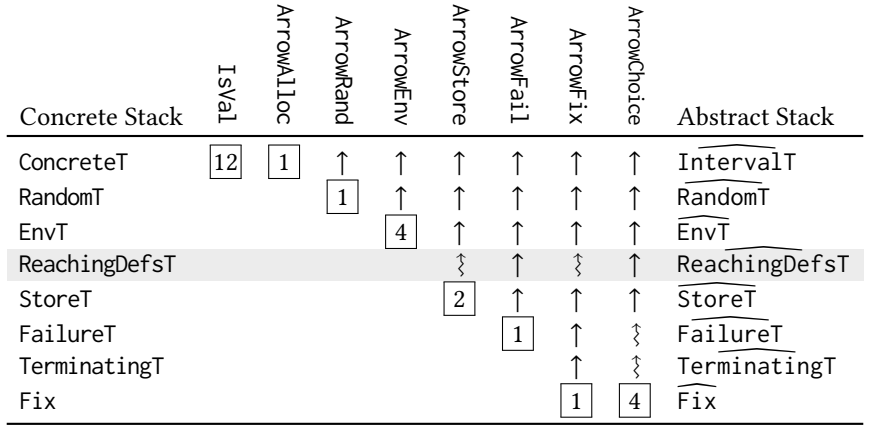

Fig. 5. Reaching definitions analysis of the While language

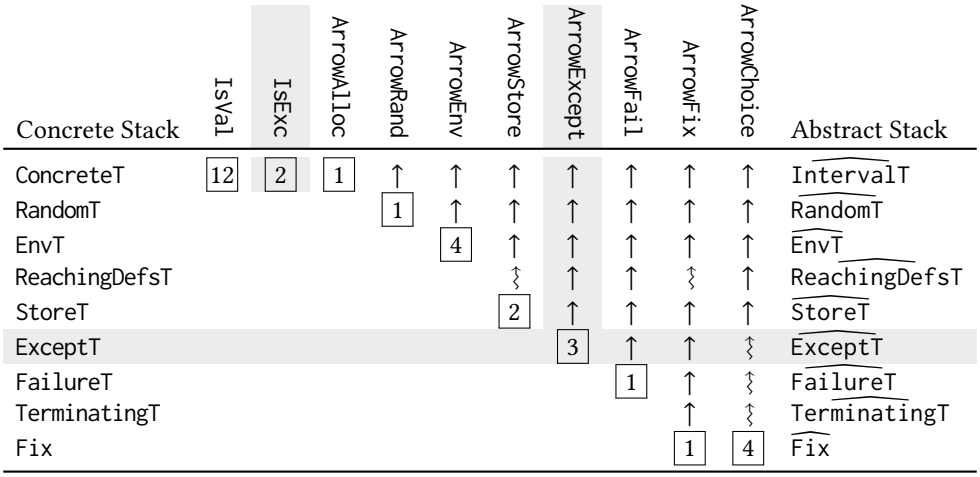

Fig. 6. Reaching definitions and interval analysis of the While language with exceptions.

7.1.3 Extending the While Language with Exceptions. Finally, we study the effort to update an analysis when a language evolves. In particular, we add exceptions to the While language and observe how this affects the interval and reaching definitions analyses.

This time we have to change the generic interpreter, because we are adding new syntax:

```
data Expr = ... | Throw ExceptName Expr
data Stmt = ... | TryCatch Stmt ExceptName String Stmt Label | Finally Stmt Stmt Label
```

The generic interpreter implements these new expressions and statements with operations of the `ArrowExcept` interface, that we now depend on. The rest of the generic interpreter stays the same. The challenge is this: Can we reuse our analysis components unchanged given that exception handling has a cross-cutting effect on the control flow of the language?

(RQ1) Modular implementation: We encapsulate the core functionality of exception handling in the language-specific interface `IsExc` and the reusable exception analysis component (Section 6), which provides the operations `throw`, `catch`, and `finally`. But, since `ArrowExcept` is a new interface, we also need to add liftings for its operations through the other components used,

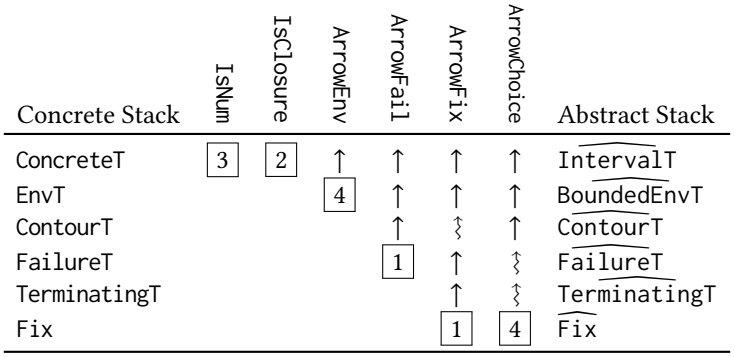

Fig. 7. k-CFA analysis of PCF: Components $\boxed{\text{n}}$ and liftings $\uparrow$/ $\wr$.

such that `throw`, `catch`, and `finally` are available after full composition. Figure 6 shows how the new component (row with gray background) and the new liftings (column with gray background) neatly integrate with the existing components. No other changes unrelated to exceptions were necessary.

(RQ2) Modular soundness proofs: We only had to prove soundness lemmas for the exceptions component and for the liftings of exception operations. All other soundness lemmas remain valid.

(RQ3) Liftings: Our extension requires 8 new liftings, of which only 1 lifting was non-trivial and required an explicit soundness proof. In total, we now have 34 liftings of which 29 are trivial and 5 are non-trivial.

To summarize, we extended the interval analysis with a reaching definitions analysis and added exceptions to our While language. In both cases, our design allowed us to capture the extension as a separate analysis component, while reusing all other analysis components unchanged. We were also able to reuse all previous soundness lemmas unchanged. For the reaching definitions analysis, we only needed to prove the new component sound. For exception handling, we additionally had to prove a few new liftings sound. However, the vast majority of liftings (85%) has a trivial implementation and soundness proof that is reusable, generic, or derivable.

7.2 Experiment 2: Control-Flow Analysis for PCF

To confirm that the succesful application of analysis components was not particular to the analysis or language of the first experiment, we define a k-CFA analysis [Shivers 1991] for PCF [Plotkin 1977]. PCF is a language with first-class functions and numbers and the main purpose of the k-CFA is to approximate which function values may be called at any function application.

To implement this analysis, we first need to describe the semantics of PCF with an arrow-based generic interpreter that captures the similarities between concrete and abstract semantics. Our case study builds on an existing generic interpreter for PCF and an existing k-CFA analysis [Keidel et al. 2018]. The goal of our case study is to modularize this analysis by using analysis components. As first step, we refactor the generic interpreter to depend on individual interfaces, each encapsulating a different concern (Listing 2 in Appendix B). Except for the use of arrows, the generic interpreter is fairly standard and requires no further explanation.

k-CFA imposes different challenges than those encountered in the first experiment. In particular, environments are embedded in to closure values and therefore must be abstracted to a finite domain if we want our analysis to terminate [Horn and Might 2010]. Let us revisit our research questions to see if they are affected by this.

(RQ1) Modular implementation: Again we succeeded in decomposing the analysis into several independent analysis components as shown in Figure 7. Each analysis component encapsulates a single concern, which simplifies its implementation. Furthermore, the composition cleanly combines the analysis components as they need to work in concert, and the liftings explain how different components interact. For example, the environment component asks the contour component for the current call context to allocate new addresses. However, the environment component is not tightly coupled to the contour component as these components communicate through an interface and are combined with component composition.

(RQ2) Modular soundness proofs: Again we were able to prove each analysis component sound independently and composition preserves soundness. We included the soundness proofs of the analysis components for the environments, exceptions, fixpoints, and values in the supplementary material accompanying this paper. Because of the separation of concerns, each soundness lemma can be verified independently, which makes it easier to prove compared to a monolithic proof. For example, when proving environment operations sound, we do not have to reason about failure or fixpoint caches. Similarly, the proof of the value operations also became easier because the operations are independent of effects in the language.

(RQ3) Liftings: The composition of analysis components for this analysis requires in total 14 liftings. Of these 14 liftings, we were able to derive the soundness proof of 11 liftings automatically using the techniques of Section 4.2. Only 3 liftings required an explicit implementation and soundness proof: The `Arrow`/`ArrowChoice` liftings of the failure and termination component and the `ArrowFix` lifting of the contour component.

To summarize, we modularized the implementation and soundness proof of a k-CFA analysis for PCF using analysis components. Each analysis component encapsulates a single concern, which simplifies the implementation and soundness proof and increases its reusability. Furthermore, the composition of these analysis components required 14 liftings of which 11 could be derived and proven sound automatically and only 3 required an explicit implementation and soundness proof. Analysis components appear to be applicable to a wider range of languages and analyses.

8 RELATED WORK

Proving soundness of static analyzers for real-world languages is a difficult endeavor. Some dynamic language features such as Java's reflection [Smaragdakis et al. 2015] or JavaScript's dynamic evaluation [Meawad et al. 2012] complicate static analysis and its soundness proof. As a consequence, static analyzers [Flanagan et al. 2002] and bug finders [Rutar et al. 2004] often either unsoundly approximate these language features or ignore them all together [Jourdan et al. 2015]. Unsound analyses still provide valuable information about program behavior, however, this information might not be reliable. For static analyzers that have been proven sound, the proof effort is significant. For example, *Verasco* [Jourdan et al. 2015] is a static analyzer for C, whose soundness has been formally verified in the proof assistant Coq. The implementation of the abstract interpreter consists of 17k lines of Coq code, as do the proof scripts (17k LOC). This shows that a soundness proof of a static analysis for a real-world language requires significant effort and expertise. In this work, we aim to reduce the effort and complexity of soundness proofs by separating analysis concerns with analysis components. We hope that with our technique the soundness proof of static analyses for real-world languages becomes more approachable.

Sergey et al. [2013] showed that analysis aspects such as context-sensitivity, polyvariance and flow-sensitivity can be captured by monads. A *monadic abstract interpreter* has the benefit, that it allows to change these analysis aspects by changing the underlying monad, while the rest of the analysis definition stays the same. This is possible because the monadic abstract interpreter

abstracts over the underlying monad with interfaces, which are similar to the interfaces of our analysis components. However, Sergey et al. did not develop a theory to prove monadic abstract interpreters sound. In this work, we demonstrate that arrows, a generalization of monads, are also capable of capturing different analysis aspects. We improve upon the work of Sergey et al. [2013] by developing a theory that simplifies and reduces the effort of proving soundness static analyses.

In this work, we describe abstract interpreters with arrows instead of monads. The benefit of using arrows is that they form an algebra and hence provide the reasoning principle of structural induction over arrow expressions [Keidel et al. 2018]. This induction principle decomposes the soundness proof of arrow-based abstract interpreters into smaller soundness lemmas of the arrow operations. We use this induction principle in Theorem 5.1 in Section 5 to prove soundness of generic interpreters instantiated with analysis components. In contrast, monadic expressions do not support an induction principle and hence a generic interpreter based on monads requires a manual soundness proof. We are not aware of any inherent disadvantages of arrows over monads, however, one important difference between arrows and monads is that arrows also capture the input of computations. In particular, higher-order arrow operations need to pass arguments to inner computations explicitly. For example, in `lookup (proc var ⟶ alloc ≺ var) ≺ var` the argument of `alloc` need to be passed in as argument to `lookup`. This explicit argument passing can be cumbersome and sometimes might not be possible if the higher-order arrow operation does not pass along the argument to the inner computation. In contrast, monads lift this restriction and arguments can be passed freely into higher-order monad operations.

As discussed in the introduction, we build on the theory of *compositional soundness proofs of abstract interpreters* by Keidel et al. [2018]. We improve upon this work by composing the arrow instances of the concrete and abstract interpreter from modular and reusable components based on *arrow transformer*. Our work simplifies the implementation and soundness proof of arrow instances, because existing analysis functionality can be reused and does not need to be reinvented. Furthermore, the implementation and soundness proof of our analysis components themselves is simpler compared to monolithic arrows, because each component captures only a single concern of an analysis instead of mixing them. Moreover, we retain the benefit of arrow-based abstract interpreters, namely, analysis creators do not need to reason about the code of the generic interpreter.

The idea of composing static analyses from modular components has also been explored by Darais et al. [2015]. The authors also propose to share code between concrete and abstract interpreter. But the code of the generic interpreter is parameterized by a monad instead of by an arrow. To recover the concrete and abstract interpreter, the generic interpreter is instantiated with two monads composed from monad transformers. These monad transformers capture reusable analysis functionality and are called *Galois Transformers*. A short-coming of this approach is that monads are missing a reasoning principles to compose a soundness proof and hence a generic interpreter based on monads still requires an explicit soundness proof. We improve upon this work by describing static analyses with analysis components based on arrow transformers. The benefit of using arrows is that arrows provide the reasoning principle of structural induction, which makes the soundness proof of static analyses compositional [Keidel et al. 2018]. This means that when we instantiate an arrow-based generic interpreter with sound analysis components, the resulting abstract interpreter is sound and no extra reasoning about the generic interpreter is required.

Defining abstract interpreters from components has been revisited recently by Darais et al. [2017]. Core of their approach is a definitional interpreter (generic interpreter) that is parameterized by a monad. Instantiating the interpreter with monads composed of monad transformers, yields the concrete and abstract semantics. The authors describe several analysis components, such as a fixpoint algorithm for big-step semantics, a trace collecting or dead code collecting semantics. We

took inspiration of these components, especially of the fixpoint algorithm. We improve upon this work by describing a theory that modularizes the soundness proof of analysis components. This means we can prove analysis components sound independently and the composition of analysis components preserves soundness.

There are several techniques to compose different abstract domains to improve the precision of the analysis such as reduced products [Cousot and Cousot 1979] and cofibered products [Venet 1996]. For example, the reduced product $\mathcal{P}(\mathbb{Z}) \leftrightarrows \texttt{Interval} \times \texttt{Parity}$ combines the abstract domains of intervals to rule out interval bounds with a wrong parity. While techniques such as reduced products compose different abstractions for data (e.g. values, environments, stores), in contrast, our paper describes a technique to modularly define and modularly prove sound the semantics for different cross-cutting aspects of the analysis (e.g. values, exceptions, mutable state, fixpoint computations). In the future, we want to explore if we can combine the technique of this paper and the techniques for composing abstract domains, by creating a component that combines two value components and computes their reduced product.

Madsen and Lhoták [2018] proposed an approach that reduces the soundness proof burden of static analyses. The approach uses SMT solvers to prove soundness of operations over some abstract domains automatically. Annotations in the code aid the SMT solver in the proving process. These annotations contain mathematical properties, such as monotonicity, required to prove soundness. The authors evaluate their approach by proving soundness of value operations over abstract domains for booleans, strings and integers. However, the authors have not explored if their verification technique scales to a soundness proof of a complete static analysis. Compared to our work, we currently do not use proof automation to prove soundness of our analysis components. However, our technique guarantees that a complete static analysis is sound if all its analysis components are sound. In the future, we want to explore how we can incorporate proof automation to simplify the soundness proof of analysis components.

9 CONCLUSION

We propose a novel approach to constructing static analyses modularly from reusable analysis components. Each analysis component covers one aspect of the analyzed language, can be proven sound independently, and their composition preserves soundness. Our analysis components consist of pairs of arrow transformers, for which we develop a Galois connection and soundness proposition. We use analysis components to instantiate arrow-based generic interpreters [Keidel et al. 2018] to obtain complete sound static analyses. A key result of our work is that a static analysis based on analysis components is sound, if all their analysis components are sound. We demonstrate the applicability and usefulness of our approach by creating a library of 13 reusable analysis components that allow us to define a k-CFA analysis for PCF and an interval and reaching definition analysis for a WHILE language.

A ARROW LAWS

The following algebraic arrow laws [Hughes 2000], allow generic reasoning about the arrows.

$$\begin{aligned}
\mathsf{arr}\ \mathsf{id} &= \mathsf{id} \\
\mathsf{arr}\ (f \ggg g) &= \mathsf{arr}\ f \ggg \mathsf{arr}\ g \\
\mathsf{first}\ (\mathsf{arr}\ f) &= \mathsf{arr}\ (\mathsf{first}\ f) \\
\mathsf{first}\ (f \ggg g) &= \mathsf{first}\ f \ggg \mathsf{first}\ g \\
\mathsf{first}\ f \ggg \mathsf{arr}\ \mathsf{fst} &= \mathsf{arr}\ \mathsf{fst} \ggg f \\
\mathsf{first}\ f \ggg \mathsf{arr}\ (\mathsf{id} *** g) &= \mathsf{arr}\ (\mathsf{id} *** g) \ggg \mathsf{first}\ f \\
\mathsf{first}\ (\mathsf{first}\ f) \ggg \mathsf{arr}\ \mathsf{assoc}_\times &= \mathsf{arr}\ \mathsf{assoc}_\times \ggg \mathsf{first}\ f \\
\mathsf{left}\ (\mathsf{arr}\ f) &= \mathsf{arr}\ (\mathsf{left}\ f) \\
\mathsf{left}(f \ggg g) &= \mathsf{left}\ f \ggg \mathsf{left}\ g \\
f \ggg \mathsf{arr}\ \mathsf{Left} &= \mathsf{arr}\ \mathsf{Left} \ggg \mathsf{left}\ f \\
\mathsf{left}\ f \ggg \mathsf{arr}\ (\mathsf{id} +++ g) &= \mathsf{arr}\ (\mathsf{id} +++ g) \ggg \mathsf{left}\ f \\
\mathsf{left}\ (\mathsf{left}\ f) \ggg \mathsf{arr}\ \mathsf{assoc}_+ &= \mathsf{arr}\ \mathsf{assoc}_+ \ggg \mathsf{left}\ f
\end{aligned}$$

where

$$\mathsf{assoc}_\times\ (a,(b,c)) = ((a,b),c) \qquad \mathsf{assoc}_+\ e = \begin{cases} \mathsf{Left}\ x & e = \mathsf{Left}\ (\mathsf{Left}\ x) \\ \mathsf{Right}\ (\mathsf{Left}\ y) & e = \mathsf{Left}\ (\mathsf{Right}\ y) \\ \mathsf{Right}\ (\mathsf{Right}\ z) & e = \mathsf{Right}\ z \end{cases}$$

B GENERIC INTERPRETERS FOR THE WHILE LANGUAGE AND PCF

```
data Expr = ...
data Statement = Assign String Expr Label | If Expr Statement Statement Label
  | While Expr Statement Label | Begin [Statement] Label

run :: (IsVal v c, ArrowAlloc (Var,v,Label) addr c, ArrowRand v c,
  ArrowEnv Var addr env c, ArrowStore addr v c, ArrowFail e c,
  ArrowFix [Statement] () c, ArrowChoice c) ⇒ c [Statement] ()
run = fix $ λrun' → proc stmts → case stmts of
  Assign x e l:ss → do
    v ← eval ≺ e
    addr ← lookup (proc (addr,_) → returnA ≺ addr) alloc ≺ (x,(x,v,l))
    write ≺ (addr,v)
    extendEnv' run' ≺ (x, addr, ss)
  If cond s1 s2 _:ss → do
    b ← eval ≺ cond
    if_ run' run' ≺ (b,([s1],[s2]))
    run' ≺ ss
  While cond body l:ss →
    run' ≺ If cond (Begin [body,While cond body l] l) (Begin [] l) l : ss
  Begin ss _:ss' → do
    run' ≺ ss; run' ≺ ss'
  [] → returnA ≺ ()
```

Listing 1. Generic interpreter for statements of the While language.

```
data Expr = Var Text | Lam String Expr
  | App Expr Expr | Y Expr | Zero | Succ Expr
  | Pred Expr | IfZero Expr Expr Expr

class IsNum v c where
  succ, pred :: c v v
  zero :: c () v
  if_ :: c x z → c y z
      → c (v, (x, y)) z

class IsClosure v env c where
  closure :: c (Expr, env) v
  applyClosure :: c ((Expr,env),v) v
                → c (v, v) v

applyClosure' ev = applyClosure $
  proc ((e,env),arg) → case e of
    Lam x body → do
      env' ← extendEnv ≺ (x,arg,env)
      localEnv ev ≺ (env', body)
    Y e' → do
      fun' ← localEnv ev ≺ (env, Y e')
      applyClosure' ev ≺ (fun',arg)
    _ → fail ≺ show e
```

```
eval :: (IsNum v c, IsClosure v env c,
 ArrowChoice c, ArrowFix Expr v c,
 ArrowEnv Text v env c,ArrowFail String c)
  ⇒ c Expr v
eval = fix $ λev → proc e → case e of
  Var x → lookup' ≺ x
  Lam x e1 → do
    env ← getEnv ≺ ()
    closure ≺ (Lam x e1, env)
  App e1 e2 → do
    fun ← ev ≺ e1
    arg ← ev ≺ e2
    applyClosure' ev ≺ (fun, arg)
  Zero → zero ≺ ()
  Succ e1 → do
    v ← ev ≺ e1; succ ≺ v
  Pred e1 → do
    v ← ev ≺ e1; pred ≺ v
  IfZero e1 e2 e3 → do
    v1 ← ev ≺ e1
    if_ ev ev ≺ (v1, (e2, e3))
  Y e1 → do
    fun ← ev ≺ e1
    env ← getEnv ≺ ()
    arg ← closure ≺ (Y e1, env)
    applyClosure' ev ≺ (fun, arg)
```

Listing 2. Generic interpreter for PCF and its language specific interface.

ACKNOWLEDGEMENTS

This research was supported by DFG grant "Evolute". We thank Arjen Rouvoet and Peter Mosses who provided helpful feedback.

REFERENCES

P. Cousot. 1999. The Calculational Design of a Generic Abstract Interpreter. In *Calculational System Design*, M. Broy and R. Steinbrüggen (Eds.). NATO ASI Series F. IOS Press, Amsterdam.

Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*. ACM, 269–282.

Patrick Cousot and Radhia Cousot. 1992. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings*. 269–295.

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *PACMPL* 1, ICFP (2017), 12:1–12:25.

David Darais, Matthew Might, and David Van Horn. 2015. Galois transformers and modular abstract interpreters: reusable metatheory for program analysis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 552–571.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 234–245.

Jeremy Gibbons (Ed.). 2010. *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*. ACM.

Cordelia V Hall, Kevin Hammond, Simon L Peyton Jones, and Philip L Wadler. 1996. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18, 2 (1996), 109–138.

Makoto Hamana and Marcelo P. Fiore. 2011. A foundation for GADTs and inductive families: dependent polynomial functor approach. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 59–70.

David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. 51–62.

John Hughes. 2000. Generalising monads to arrows. *Sci. Comput. Program.* 37, 1-3 (2000), 67–111.

Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 247–259.

Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional Soundness Proofs of Abstract Interpreters. *PACMPL* ICFP (2018).

Jens Knoop and Oliver Rüthing. 1999. Optimization Under the Perspective of Soundness, Completeness, and Reusability. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*. 288–315.

Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. 333–343.

Magnus Madsen and Ondrej Lhoták. 2018. Safe and Sound Program Analysis with Flix. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*.

Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. 2012. Eval Begone!: Semi-automated Removal of Eval from Javascript Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 607–620.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer.

Oystein Ore. 1944. Galois connexions. *Trans. Amer. Math. Soc.* 55, 3 (1944), 493–513.

David Lorge Parnas. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Communication of the ACM* 15, 12 (1972), 1053–1058.

Ross Paterson. 2001. A New Notation for Arrows. In *Proceedings of International Conference on Functional Programming (ICFP)*. ACM, 229–240.

Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theor. Comput. Sci.* 5, 3 (1977), 223–255.

Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. In *15th International Symposium on Software Reliability Engineering (ISSRE 2004), 2-5 November 2004, Saint-Malo, Bretagne, France*. 245–256.

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic Abstract Interpreters. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 12.

Olin Shivers. 1991. *Control-flow analysis of higher-order languages*. Ph.D. Dissertation. Carnegie Mellon University.

Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*. 485–503.

Arnaud Venet. 1996. Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*. 366–382.