Compositional Soundness Proofs of Abstract Interpreters

SVEN KEIDEL, CASPER BACH POULSEN, and SEBASTIAN ERDWEG, Delft University of Technology, The Netherlands

Abstract interpretation is a technique for developing static analyses. Yet, proving abstract interpreters sound is challenging for interesting analyses, because of the high *proof complexity* and *proof effort*. To reduce complexity and effort, we propose a framework for abstract interpreters that makes their soundness proof compositional. Key to our approach is to capture the similarities between concrete and abstract interpreters in a single shared interpreter, parameterized over an arrow-based interface. In our framework, a soundness proof is reduced to proving reusable soundness lemmas over the concrete and abstract instances of this interface; the soundness of the overall interpreters follows from a generic theorem.

To further reduce proof effort, we explore the relationship between soundness and parametricity. Parametricity not only provides us with useful guidelines for how to design non-leaky interfaces for shared interpreters, but also provides us soundness of shared pure functions as *free theorems*. We implemented our framework in Haskell and developed a k-CFA analysis for PCF and a tree-shape analysis for Stratego. We were able to prove both analyses sound compositionally with manageable complexity and effort, compared to a conventional soundness proof.

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

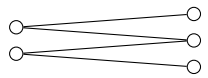
Additional Key Words and Phrases: Abstract Interpretation, Soundness

ACM Reference Format:
Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional Soundness Proofs of Abstract Interpreters. 1, 1 (September 2018), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Abstract interpretation [Cousot and Cousot 1979] is an approach to static analysis with soundness at its heart: An abstract interpreter must approximate the behavior of a program as prescribed by a concrete interpreter. This soundness proposition can guide the design of abstract interpreters [Cousot 1999] and prescribes what needs to be proven about the analysis. Unfortunately, it is far less clear *how* to prove an abstract interpreter sound and, in particular, how to decompose the soundness proof into proof obligations of manageable size. Yet, compositional soundness proofs are crucial when developing verified abstract interpreters for real-world languages to reduce *proof complexity* and *proof effort*.

What makes the decomposition of the soundness proof difficult is that concrete and abstract interpreters are often misaligned, such that a case of one interpreter relates to multiple cases of the other interpreter (see figure). For example, a language construct `IfZero` that checks if a given number is zero has two outcomes in the concrete interpreter (is zero, is not zero) but three outcomes in an interval analysis (is zero, contains zero, does not contain zero). Such misalignment between concrete and abstract interpreter prevents a piece-wise decomposition of the soundness proof. Conversely, when concrete and abstract interpreter functions share the same structure we could decompose the proof along that structure.

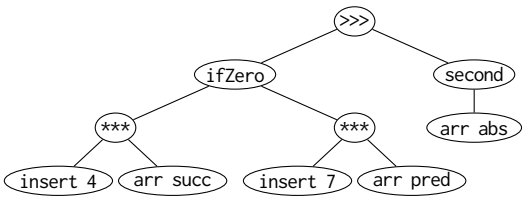


Authors' address: Sven Keidel; Casper Bach Poulsen; Sebastian Erdweg, Delft University of Technology, The Netherlands.

We present a novel framework for defining abstract interpreters such that their soundness proofs become compositional. Our key contributions are that (i) we can abstract from the difference between concrete and abstract interpreters such that (ii) the soundness proof for the shared parts is fully compositional and (iii) follows automatically from the soundness of the non-shared parts. Indeed, most concrete and abstract interpreter are very similar and only differ in a few places where the interpreters operate on the concrete or abstract domain (e.g., addition of numbers vs intervals). We propose to make these similarities explicit in a *shared parameterized interpreter* function, abstracting from the interpretations of primitive operations on the respective domain. We realize this abstraction using Haskell's arrows [Hughes 2000], a generalization of monads. Instantiating the shared interpreter with arrow instances for the concrete and abstract domain fixes the respective language semantics. For an abstract interpreter factorized in this way, we obtain the following benefits when proving soundness:

(1) We can decompose the soundness proof into soundness lemmas about the operations of the concrete and abstract arrow instances. Each soundness lemma is context free, i.e., independent from where the operation is used in the shared interpreter. This narrows the scope of the lemmas and makes them reusable.
(2) Arrows restrict the meta-language of shared interpreters, which solely consists of arrow expressions. Because arrows are a first-order language, we can use structural induction over arrow expressions to obtain a generic soundness proof for *any* shared interpreter composed of sound arrow operations.

For example, consider the following abstract syntax tree of a shared arrow expression. On the right, we list the soundness lemmas required to prove concrete and abstract instances of the shared expression sound. We write $e \dot{\sqsubseteq} \widehat{e}$ to mean that e is soundly approximated by $\widehat{e}$:



$$f \dot{\sqsubseteq} \widehat{f} \wedge g \dot{\sqsubseteq} \widehat{g} \implies (f \mathbin{***} g) \dot{\sqsubseteq} (\widehat{f \mathbin{***} g})$$
$$f \dot{\sqsubseteq} \widehat{f} \wedge g \dot{\sqsubseteq} \widehat{g} \implies (f \mathbin{>>>} g) \dot{\sqsubseteq} (\widehat{f \mathbin{>>>} g})$$
$$f \dot{\sqsubseteq} \widehat{f} \implies \mathtt{second}\ f \dot{\sqsubseteq} \widehat{\mathtt{second}}\ \widehat{f}$$
$$f \dot{\sqsubseteq} \widehat{f} \wedge g \dot{\sqsubseteq} \widehat{g} \implies \mathtt{ifZero}\ f\ g \dot{\sqsubseteq} \widehat{\mathtt{ifZero}}\ \widehat{f}\ \widehat{g}$$
$$\mathtt{insert}\ n \dot{\sqsubseteq} \widehat{\mathtt{insert}}\ n \qquad \mathtt{arr}\ \mathtt{succ} \dot{\sqsubseteq} \widehat{\mathtt{arr}}\ \mathtt{succ}$$
$$\mathtt{arr}\ \mathtt{pred} \dot{\sqsubseteq} \widehat{\mathtt{arr}}\ \mathtt{pred} \qquad \mathtt{arr}\ \mathtt{abs} \dot{\sqsubseteq} \widehat{\mathtt{arr}}\ \mathtt{abs}$$

Functions >>>, ***, and second are language-independent arrow operations, arr is language-independent and embeds pure functions into arrow computations, and ifZero and insert are language-specific operations. The concrete and abstract arrow instances define implementations for all arrow operations; we denote abstract implementations with a *hat* $\widehat{***}$ to distinguish them from concrete definitions ***. With that, we formulate a context-free soundness lemma for each arrow operation. For example, the lemma of *** is context-free in that it proves soundness of the operation *for all* sound subexpressions $f, \widehat{f}$ and $g, \widehat{g}$. This allows us to reuse the same lemma for every occurrence of *** in the shared expression. Soundness of the shared expression now follows by structural induction on arrow expressions: Given all leaves are sound and all intermediate nodes preserve soundness, the composed expression is sound. This way, we have effectively decomposed the soundness proof into smaller lemmas that can be proved independently and that can be composed to reason about full abstract interpreters. We assert this result as a generic meta-theorem, stating that any arrow expression is sound if the arrow operations it uses are sound.

We also show that in meta-languages with *parametricity* [Reynolds 1983], the soundness of shared code follows as a *free theorem* [Wadler 1989], given the interface does not leak details of the abstract interpreter into shared code. Based on this observation, we extract guidelines for the

interface design to be used in the shared interpreter. In particular, following our guidelines, we get soundness of pure functions embedded with `arr` for free, which reduces the number of lemmas required for our example from 8 to 5. Lastly, parametricity allows us to generalize our framework to abstract interpreters that share code over interfaces other than arrows.

To evaluate our approach, we implemented a k-CFA analysis for PCF and developed a tree-shape analysis for *Stratego* [Visser et al. 1998], a dynamic language for program transformations used in practice and featuring dynamic scoping of pattern-bound variables, higher-order functions, and generic tree traversals. For both analyses, we extract a shared parameterized interpreter and prove it sound compositionally, thus demonstrating the applicability of our approach. We show that, for the k-CFA analysis, the soundness proof can be decomposed into 16 independently provable lemmas and for the tree-shape analysis into 27 lemmas. We reflect on our soundness proofs and explain why it has a reduced complexity and effort compared to conventional soundness proofs.

In summary, we make the following contributions:

- We describe a new approach for organizing abstract interpreters by sharing code with the concrete interpreter over an interface based on arrows.
- We show that the soundness proof of such abstract interpreters can be conducted compositionally, based on soundness lemmas of the arrow operations.
- We prove a generic meta-theorem showing that *any* shared interpreter is sound if it is composed of sound arrow operations. Thus, the soundness proofs of our abstract interpreters are not only compositional, but proofs about the shared parts actually follow for free.
- We apply parametricity to develop guidelines for the interface design, to obtain soundness of embedded pure functions for free, and to generalize our approach to interfaces other than arrows.
- We demonstrate the applicability of our approach through two case studies and show that our approach reduces the effort and complexity of soundness proofs.

2 WHY AND HOW TO MAKE SOUNDNESS PROOFS COMPOSITIONAL

In this section, we first discuss the *complexity* and *effort* of soundness proofs of conventional abstract interpreters. Then, we describe informally how we can make soundness proofs compositional and how this reduces proof complexity and effort.

2.1 Conventional Abstract Interpreters

To illustrate the difficulties of soundness proofs of conventional abstract interpreters, we construct an abstract interpreter for a small example language in Haskell. Expressions in our example language are either variables, integer literals, additions, or conditionals:

```
data Expr = Var String | Lit Int | Add Expr Expr
          | IfZero Expr Expr Expr
```

We would like to implement an abstract interpreter for this language that predicts the numbers a program evaluates to as an interval. For example, consider the following program:

```
IfZero (Var "x") (Lit 2) (Lit 5),
```

This program evaluates to 2 if `x` is bound to `0` and to 5 otherwise. In order to be sound, the abstract interpreter must approximate all possible results of this program. That is, if the interval for `x` may contain `0`, the most precise approximation of this program in the domain of intervals is $[2, 5]$.

We define a conventional concrete interpreter `eval` and a conventional abstract interpreter $\widehat{\texttt{eval}}$ for this language in Figure 1. The definition of the concrete interpreter is standard, hence, we only explain how the abstract interpreter differs. In case of an addition, the abstract interpreter adds

```
type Val = Int
type Env = Map String Val

eval :: Expr -> Env -> Maybe Val
eval e env = case e of
  Var x -> lookup x env
  Lit n -> return n
  Add e1 e2 -> do
    v1 <- eval e1 env
    v2 <- eval e2 env
    return (v1 + v2)
  IfZero e1 e2 e3 -> do
    v <- eval e1 env
    if v == 0
    then eval e2 env
    else eval e3 env
```

```
type Val^ = (Int,Int)
type Env^ = Map String Val^

eval^ :: Expr -> Env^ -> Maybe^ Val^
eval^ e env = case e of
  Var x -> lookup^ x env
  Lit n -> return (n,n)
  Add e1 e2 -> do
    (i1,j1) <- eval^ e1 env
    (i2,j2) <- eval^ e2 env
    return (i1+i2, j1+j2)
  IfZero e1 e2 e3 -> do
    (i1,j1) <- eval^ e1 env
    if i1 == 0 && j1 == 0
    then eval^ e2 env
    else if j1 < 0 || 0 < i1
      then eval^ e3 env
      else eval^ e2 env ⊔ eval^ e3 env
```

Fig. 1. Conventional design of a concrete (left) and abstract interpreter (right) for our example language.

the interval bounds. In case of `IfZero`, as described in the introduction, the abstract interpreter distinguishes three cases for the interval resulting from evaluating `e1`: the interval contains zero only, does not contain zero, or contains zero and other values. If the interval contains zero only, we evaluate `e2`; if the interval does not contain zero, we evaluate `e3`. But if the interval contains zero and other values, we evaluate both `e2` and `e3` and join their results using the least upper bound operation $\sqcup$.

The abstract interpreter appears to correctly approximate the concrete interpreter's behavior. But what exactly do we have to prove to verify the soundness of $\widehat{\texttt{eval}}$? We prove the following soundness proposition for the *collecting semantics* [Cousot 1999] of `eval`:

$$\forall\, e \in \mathsf{Expr}.\, \forall\, X \subseteq \mathsf{Env}. \quad \alpha_V(\{\texttt{eval}\ e\ \rho \mid \rho \in X\}) \sqsubseteq \widehat{\texttt{eval}}\ e\ \alpha_E(X)$$

Here, α_V and α_E are abstraction functions of Galois connections [Cousot and Cousot 1979] for values and environments of the interpreters:

$$\alpha_V : \mathcal{P}(\mathsf{Val}) \leftrightarrows \widehat{\mathsf{Val}} : \gamma_V \qquad \alpha_E : \mathcal{P}(\mathsf{Env}) \leftrightarrows \widehat{\mathsf{Env}} : \gamma_E$$

$$\alpha_V(X) = (\min X, \max X) \qquad \alpha_E(X) = \bigsqcup_{\rho \in X} [\, x \mapsto \alpha_V(\rho(x)) \mid x \in \mathrm{dom}(\rho)]$$

The soundness proposition quantifies over sets of environments X, which represent properties of the program's free variables. For example, $X = \{\rho \mid \rho \in \mathsf{Env} \wedge \forall\, (x \mapsto v) \in \rho.\ \mathrm{even}(v)\}$ describes environments that map variables to even numbers. The soundness proposition states that, for any e, all concrete evaluations of e under environments ρ satisfying X must be predicted by a single abstract evaluation of e under the single abstract environment $\alpha_E(X)$ representing property X.

To prove this soundness proposition for our example, we proceed by structural induction over the expressions of our language. The soundness proof for `Var`, `Lit` and `Add` is easy, because the interpreters align and we only need to reason about the Galois connection α_V. The case `IfZero` $e_1\ e_2\ e_3$ is slightly more involved: We perform a case distinction on the result of $\widehat{\texttt{eval}}\ e_1\ \alpha_E(X)$, because the result prescribes which branch of `IfZero` will be analyzed.

- In case $\widehat{\texttt{eval}}\ e_1\ \alpha_E(X) = \texttt{Just}\ (0,0)$, the first branch e_2 will be analyzed. From the induction hypothesis for e_1, we learn that $\alpha_V\{\ \texttt{eval}\ e_1\ \rho \mid \rho \in X\} \sqsubseteq \widehat{\texttt{eval}}\ e_1\ \alpha_E(X) = \texttt{Just}\ (0,0)$. Since $\gamma_V(\texttt{Just}\ (0,0)) = \{0\}$, the concrete interpretation $\texttt{eval}\ e_1$ must also result in 0 and the concrete interpreter evaluates the the first branch e_2. This lets us conclude:
$$\begin{aligned}\alpha(\{\ \texttt{eval}\ (\texttt{IfZero}\ e_1\ e_2\ e_3)\ \rho \mid \rho \in X\}) &\sqsubseteq \alpha(\{\ \texttt{eval}\ e_2\ \rho \mid \rho \in X\}) \\ &\sqsubseteq \widehat{\texttt{eval}}\ e_2\ \alpha_E(X) \sqsubseteq \widehat{\texttt{eval}}\ (\texttt{IfZero}\ e_1\ e_2\ e_3)\ \alpha_E(X).\end{aligned}$$
- The case for intervals without 0 is analogous to the previous case.
- The last case is more involved because $\widehat{\texttt{eval}}\ e_1\ \alpha_E(X)$ contains zero and other numbers. In this case, we have to reason about multiple outcomes of behavior of the concrete interpreter. Independent of the result of e_1, the concrete interpreter will either evaluate the first or second branch of `IfZero` and hence:
$$\{\texttt{eval}\ (\texttt{IfZero}\ e_1\ e_2\ e_3)\ \rho \mid \rho \in X\}) \subseteq \{\texttt{eval}\ e_2\ \rho \mid \rho \in X\} \cup \{\texttt{eval}\ e_3\ \rho \mid \rho \in X\}$$
This lets us conclude:
$$\begin{aligned}&\alpha_V(\{\texttt{eval}\ (\texttt{IfZero}\ e_1\ e_2\ e_3)\ \rho \mid \rho \in X\}) \\ &\quad\sqsubseteq \alpha_V(\{\texttt{eval}\ e_2\ \rho \mid \rho \in X\} \cup \{\texttt{eval}\ e_3\ \rho \mid \rho \in X\}) \\ &\quad\sqsubseteq \alpha_V(\{\texttt{eval}\ e_2\ \rho \mid \rho \in X\}) \sqcup \alpha_V(\{\texttt{eval}\ e_3\ \rho \mid \rho \in X\}) \\ &\quad\sqsubseteq \widehat{\texttt{eval}}\ e_2\ \alpha_E(X) \sqcup \widehat{\texttt{eval}}\ e_3\ \alpha_E(X) = \widehat{\texttt{eval}}\ (\texttt{IfZero}\ e_1\ e_2\ e_3)\ \alpha_E(X).\end{aligned}$$

With this, we have proved soundness for a very simple static analysis of a very simple programming language. And already the proof was not trivial: For every case in the abstract interpreter, we had to establish which cases of the concrete interpreter are relevant and then establish that the abstract interpreter subsumes them all. The complexity and effort of such proofs quickly grows as language features become more complex. For example, consider another language construct `TryZero e1 e2 e3` of our example language whose concrete semantics is like `IfZero e1 e2 e3` except the evaluation defaults to `e3` if the evaluation of `e1` fails:

```
eval e env = case e of
  TryZero e1 e2 e3 -> case eval e1 env of
    Just v | v == 0    -> ① eval e2 env
           | otherwise -> ② eval e3 env
    Nothing            -> ③ eval e3 env
```

When defining an abstract interpreter for `TryZero`, we need to be careful about how we handle failed executions. In particular, we often do not know whether a computation definitely succeeds or fails. To be precise, we use type $\widehat{\texttt{Maybe}}$ to represent potential failure (`JustNothing`) alongside definite success (`Just`) and definite failure (`Nothing`). Based on this type, we can implement `TryZero` in the abstract interpreter as shown in Figure 2.

In the soundness proof for `TryZero`, we have to relate 7 cases of the abstract interpreter to 3 cases of the concrete interpreter as indicated by the diagram on the right. Compared to `IfZero`, the soundness proof for `TryZero` is worse in two ways:

- We have to relate a single case of the abstract interpreter to up to 3 cases of the concrete interpreter at once. The more cases we need to relate, the higher the *proof complexity*.
- We have to prove 7 cases of the abstract interpreter sound. The more cases we need to prove, the higher the *proof effort*.

These problems are already apparent in the soundness proof for our example language. For precise abstract interpreters of real-world languages, proof complexity and proof effort quickly make a soundness proof infeasible. However, it is exactly for analyses of such languages that we need

```
eval^ :: Expr -> Env^ -> Maybe Val^
eval^ e env = case e of
  TryZero e1 e2 e3 -> case eval^ e1 env of
    Just (i,j)
      | (i,j) == (0,0) -> (A) eval^ e2 env
      | j < 0 || 0 < i -> (B) eval^ e3 env
      | otherwise      -> (C) eval^ e2 env ⊔ eval^ e3 env
    Nothing            -> (D) eval^ e3 env
    JustNothing (i1,j1)
      | (i,j) == (0,0) -> (E) eval^ e2 env ⊔ eval^ e3 env
      | j < 0 || 0 < i -> (F) eval^ e3 env
      | otherwise      -> (G) eval^ e2 env ⊔ eval^ e3 env
```

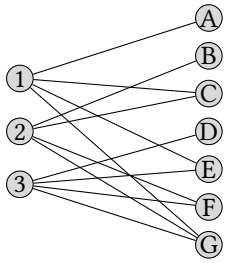


Fig. 2. Abstract interpretation of TryZero and how its cases relate to the concrete interpreter.

soundness proofs to ensure all corner cases are covered. Therefore, the question this paper aims to answer is: How can we make soundness proofs of abstract interpreters simpler and more systematic, such that soundness proofs of abstract interpreters for real-world languages become feasible?

2.2 Concrete and Abstract Interpreters using Arrows

This paper presents techniques that make soundness proofs of abstract interpreters compositional, thereby reducing proof complexity and proof effort. A key idea is to factorize the implementation of a concrete and abstract interpreter into a shared implementation based on Haskell arrows [Hughes 2000]. This factoring aligns the cases of the interpreters and exposes the structure along which a proof can be decomposed, namely the arrow operations used to define the shared interpreter. Because arrows are a first-order language and their code is not interleaved with computations of the meta-language, they induce an induction principle in the meta-language. By proving that every arrow operation preserves soundness of its arguments, the soundness of the entire shared interpreter directly follows from this induction principle. With this technique, we can decompose monolithic soundness proofs into smaller, reusable, and context free soundness lemmas about the arrow operations of the shared interpreter.

Note, that our technique requires to implement a concrete interpreter in the same meta-language as the abstract interpreter. This causes extra work if a reference semantics already exists and is implemented in a different meta-language. However, it simplifies the soundness proof as we do not have to conduct proofs across different meta-languages. In this subsection, we provide a brief introduction to arrows, and demonstrate how to use arrows to define a shared interpreter that corresponds to the concrete and abstract interpreters in the previous subsection. In the next subsection, we show how this shared interpreter enables a compositional soundness proof.

Arrows, like monads, support effectful computations that, for example, manipulate state, trigger exceptional control flow, or rely on non-determinism. Arrows generalize monads by internalizing the input type for a computation. For example, an arrow computation of type `(c x y)` expects a value of type `x` as input and yields a value of type `y`; `c` is the higher-order type constructor defining the arrow. In contrast, monadic computations have type `m y` and rely on meta-level bindings to implicitly provide inputs `x` through the lexical context in which the computation was defined. The pretty notation [Paterson 2001] for arrows reads similarly to `do`-notation for monads. The keyword `proc x` starts a new arrow computation with input `x`. The notation `y <- f -< x` represents an arrow computation `f`, which receives its input from the variable `x` and binds the result to the variable `y`. This notation desugars to operations of the `Arrow`, `ArrowChoice`, and user-defined type classes with language-specific operations. This desugaring translates sequential statements

`y <- f -< x; g -< y` into the sequential composition operator `f >>> g`. For arrow expressions where variables span multiple statements as in `y1 <- f -< x1; y2 <- g -< x2; h -< (y1,y2)` the notation desugars into the parallel composition operator `***` as in `(f *** g) >>> h`. Case expressions are translated to a pure function that destructs a sum type into an `Either` type, embedded into arrows with `arr :: (x -> y) -> c x y`, followed by the choice operator `|||` of the `ArrowChoice` type class that encodes the bodies of each case. For example, here is an example desugaring ($\rightsquigarrow$) of an arrow expression:

```
    proc e -> case e of { Var x -> f -< x; Lit y -> g -< y }
⇝   arr (\e -> case e of { Var x -> Left x; Lit y -> Right y }) >>> (f ||| g)
```

Appendix A contains an illustrative example that the reader may find helpful for understanding how the pretty notation for arrows desugars into arrow expressions. For the full details on how arrows desugar, we refer the reader to the work of Paterson [2001].

In Figure 3, we use arrows to describe a shared interpreter `eval'` that generalizes both `eval` and $\widehat{\texttt{eval}}$ from the previous subsection. To do so, we extract the operations that differ between the concrete and abstract interpreter into a type class `IsVal`. Each type class member of `IsVal` in Figure 3 represents a language-specific operation. `lookup` defines a variable lookup operation as an arrow from a string to the value type `v` that the type class is parameterized by. The `ifZero` operation is parameterized by two arrows as continuations and takes as argument a triple of a value and arguments `x` and `y` for the continuations. If the value in the triple is zero, the first continuation is invoked using `x`; otherwise, the second continuation is invoked using `y`. The `try` operation is parameterized by three arrows: one for computing a value (or raising an error); one for dispatching on the value resulting from invoking the first arrow if no error was raised; and one for the case where an error was raised. The `fix` operator of the type class `ArrowFix` (also Figure 3) computes the fixpoint of the shared interpreter. This allows concrete and abstract interpreter to employ different fixpoint strategies.

To define the concrete and abstract language semantics, we instantiate the shared interpreter with two different arrow instances. We do this in by defining two arrow types `Interp` and $\widehat{\texttt{Interp}}$ that define instances for the `Arrow`, `ArrowChoice`, `ArrowFix`, and `IsValue` type classes. In Figure 4, we show the arrow types, their instances for `IsValue`, and the top-level interpreters `eval` and $\widehat{\texttt{eval}}$ that instantiated the shared interpreter `eval'`. The shared interpreter completely desugars into operations of the arrow type classes implemented by `Interp` and $\widehat{\texttt{Interp}}$. Ultimately, the two instantiated interpreters have the same semantics as the interpreters of Section 2.1. Note that since the shared interpreter describes a parameterized semantics, we can define new alternative abstract domains by instantiating the shared interpreter with another arrow instance.

2.3 Compositional Soundness Proofs of Abstract Interpreters

The previous section described how to define concrete and abstract interpreters in a way that common code is shared between the two. This organization of concrete and abstract interpreter allows us to prove soundness of interpreters like `eval` and $\widehat{\texttt{eval}}$ in Figure 4 compositionally based on separate *soundness preservation lemmas* for each arrow operation. For our example, we prove the following soundness preservation lemmas, one for each operation of the `IsVal`, `Arrow`, `ArrowChoice`, and `ArrowFix` type classes. We use $f \mathrel{\dot\sqsubseteq} \widehat{f}$ as a compact notation for the soundness proposition.

- $\texttt{arr}\ f \mathrel{\dot\sqsubseteq} \widehat{\texttt{arr}}\ f$ for each pure function f in the shared interpreter,
- $\texttt{lit} \mathrel{\dot\sqsubseteq} \widehat{\texttt{lit}}$, $\texttt{add} \mathrel{\dot\sqsubseteq} \widehat{\texttt{add}}$, $\texttt{lookup} \mathrel{\dot\sqsubseteq} \widehat{\texttt{lookup}}$,
- if $f \mathrel{\dot\sqsubseteq} \widehat{f}$ and $g \mathrel{\dot\sqsubseteq} \widehat{g}$ then $\texttt{ifZero}\ f\ g \mathrel{\dot\sqsubseteq} \widehat{\texttt{ifZero}}\ \widehat{f}\ \widehat{g}$
- if $f \mathrel{\dot\sqsubseteq} \widehat{f}$ and $g \mathrel{\dot\sqsubseteq} \widehat{g}$ and $h \mathrel{\dot\sqsubseteq} \widehat{h}$ then $\texttt{try}\ f\ g\ h \mathrel{\dot\sqsubseteq} \widehat{\texttt{try}}\ \widehat{f}\ \widehat{g}\ \widehat{h}$

```
data Expr = Var String
          | Lit Int
          | Add Expr Expr
          | IfZero Expr Expr Expr
          | TryZero Expr Expr Expr

class ArrowFix x y c where
  fix :: (c x y -> c x y) -> c x y

class ArrowChoice c => IsVal v c where
  lookup :: c String v
  lit :: c Int v
  add :: c (v, v) v
  ifZero :: c x v -> c y v -> c (v,(x,y)) v
  try :: c x y -> c y v -> c x v -> c x v
```

```
eval' :: IsVal v c => c Expr v -> c Expr v
eval' ev = proc e -> case e of
  Var x -> lookup -< x
  Lit n -> lit -< n
  Add e1 e2 -> do
    v1 <- ev -< e1; v2 <- ev -< e2
    add -< (v1,v2)
  IfZero e1 e2 e3 -> do
    v <- ev -< e1
    ifZero ev ev -< (v,(e2,e3))
  TryZero e1 e2 e3 ->
    try (proc (e1,x) -> do
           v <- ev -< e1
           returnA -< (v,x))
        (ifZero ev ev)
        (proc (_,(_,e3)) -> ev -< e3)
        -< (e1,(e2,e3))
```

Fig. 3. Shared interpreter based on arrows.

```
type Interp a b = Env -> a -> Maybe b
instance IsVal Val Interp where
  lookup = \e x -> Map.lookup x e
  lit = arr id
  add = arr (\(x,y) -> x + y)

  ifZero f g = proc (v,(x,y)) ->
    if v == 0
    then f -< x
    else g -< y


  try f g h = \e x -> case f e x of
    Just y -> g e y
    Nothing -> h e x


eval :: Interp Expr Val
eval = fix eval'
```

```
type Interp̂ a b = Env̂ -> a -> Maybe̲ b
instance IsVal Val̂ Interp̂ where
  lookup̂ = \e x -> toMaybe̲ (Map.lookup e x)
  lit̂ = arr (\n -> (n,n))
  add̂ = arr (\((i1,j1),(i2,j2)) ->
              (i1+i2,j1+j2))
  ifZero̲ f g = proc ((i,j),(x,y)) ->
    if i == 0 && j == 0
    then f -< x
    else if j < 0 || 0 < i
         then g -< y
         else (f -< x) ⊔ (g -< y)
  trŷ f g h = \e x -> case f e x of
    Just y -> g e y
    Nothing -> h e x
    JustNothing y -> g e y ⊔ h e x

eval̂ :: Interp̂ Expr Val̂
eval̂ = fix eval'
```

Fig. 4. Arrow instances for the concrete interpreter (left) and the abstract interpreter (right).

- if $f \dot{\sqsubseteq} \widehat{f}$ and $g \dot{\sqsubseteq} \widehat{g}$ then $f >\!\!>\!\!> g \dot{\sqsubseteq} \widehat{f} \mathbin{\widehat{>\!\!>\!\!>}} \widehat{g}$
- if $f \dot{\sqsubseteq} \widehat{f}$ and $g \dot{\sqsubseteq} \widehat{g}$ then $f \mathbin{***} g \dot{\sqsubseteq} \widehat{f} \mathbin{\widehat{***}} \widehat{g}$
- if $f \dot{\sqsubseteq} \widehat{f}$ and $g \dot{\sqsubseteq} \widehat{g}$ then $f \mid\!\mid\!\mid g \dot{\sqsubseteq} \widehat{f} \mathbin{\widehat{\mid\!\mid\!\mid}} \widehat{g}$
- if $\left[\forall x, \widehat{x}.\, x \dot{\sqsubseteq} \widehat{x} \Rightarrow f(x) \dot{\sqsubseteq} f(\widehat{x})\right]$ then $\mathtt{fix}\, f \dot{\sqsubseteq} \widehat{\mathtt{fix}}\, f$

Fig. 5. Soundness lemmas for the `try` operation (left) and for the `ifZero` operation (right).

The fixpoint combinator `fix` required a different soundness lemma, because it is the only higher-order construct compared to the otherwise first-order arrow language. To keep the rest of the shared arrow code first-order, we only allow one occurrence of `fix` at the very top-level of the interpreters.

For soundness of the interpreters $\texttt{eval} \mathrel{\dot{\sqsubseteq}} \widehat{\texttt{eval}}$, we first unfold the definition of `eval` and $\widehat{\texttt{eval}}$ which gives us $\texttt{fix}\ \texttt{eval}' \mathrel{\dot{\sqsubseteq}} \widehat{\texttt{fix}}\ \texttt{eval}'$. We then use the lemma for `fix`, which leaves us to prove $\texttt{eval}'\ x \mathrel{\dot{\sqsubseteq}} \texttt{eval}'\ \widehat{x}$ given $x \mathrel{\dot{\sqsubseteq}} \widehat{x}$. Because $\texttt{eval}'\ x$ and $\texttt{eval}'\ \widehat{x}$ refer to an arrow expression with the same structure, except for occurrences of x and $\widehat{x}$, we can use structural induction over the arrow expressions. The cases of this induction are always instances of the soundness lemmas for the arrow operations from above and the assumption $x \mathrel{\dot{\sqsubseteq}} \widehat{x}$. This proves that the top-level interpreters are sound.

But what impact does compositional soundness proofs have on proof complexity and proof effort? Let us compare the proof of `TryZero` to the non-compositional proof from the previous subsection. Before, we had to prove 7 cases of the abstract interpreter and relate them to up to 3 cases of the concrete interpreter. Now, `TryZero` is composed of `try` and `ifZero`. Their soundness lemmas are simpler and can be proved independently as illustrated in Figure 5. Moreover, the soundness lemmas are independent of their specific usage in the shared interpreter and can be reused whenever the shared interpreter makes use of `try` or `ifZero`. In particular, we reused the lemma for `ifZero` twice: Once for interpreting `IfZero` and once for interpreting `TryZero`.

To summarize, compared to conventional soundness proofs, in compositional soundness proofs we have to prove smaller lemmas that are context-free, which reduces the proof complexity. Furthermore, we have to prove less cases and lemmas are reused, which reduces the proof effort. In the next two sections we describe our framework more formally.

3 SOUNDNESS PROPOSITION FOR ARROWS

To construct compositional soundness proofs, we first need a soundness proposition $\dot{\sqsubseteq}$ that is applicable for all intermediate expressions of the interpreters. For example, the shared interpreter of Figure 3 uses the `ifZero` operator with return type `c (v,(Expr,Expr)) v`. This type is instantiated in the concrete interpreter with arrow type

```
Interp (Val,(Expr,Expr)) Val
```

and in the abstract interpreter with arrow type

$$\widehat{\texttt{Interp}}\ (\widehat{\texttt{Val}},(\texttt{Expr},\texttt{Expr}))\ \widehat{\texttt{Val}}.$$

To relate values of these two types in our soundness proposition, we need to define a Galois connection [Cousot and Cousot 1979] between these arrow types. However, in general, our shared interpreter makes use of arrows of many different types, many which of which only become apparent after arrow desugaring. For example, the composition operator `>>>` is used by the shared interpreter with various types ranging from `Val` and `Expr` to tuples, `Maybe`, `Either`, and combinations thereof. To relate all types with a Galois connections, we require a systematic way for constructing Galois connections and, based on that, soundness propositions.

3.1 Systematic Way for Constructing Galois Connections

A well-known technique for constructing Galois connections is described by Nielson et al. [1999, Lemma 4.23]. A Galois connection $\alpha : \mathcal{P}A \rightleftarrows \widehat{A} : \gamma$ can be defined by an embedding function $\iota : A \to \widehat{A}$, such that the abstraction function is given by $\alpha(X) = \bigsqcup\{\iota(x) \mid x \in X\}$. Then the concretization function exists and is uniquely determined by $\gamma(\widehat{x}) = \{x \mid \alpha(x) \sqsubseteq \widehat{x}\}$. In other words, we only need to define an embedding function and we obtain the Galois connection for free.

First, we define embedding functions for abstracted base types. For example, for an interval analysis, we can define an embedding function for numeric values $\iota : \texttt{Int} \to \texttt{Interval}$ by $\iota(n) = [n, n]$. Then the abstraction function sends the set $\{1, 3, 5\}$ to $\bigsqcup\{\iota(1), \iota(3), \iota(5)\} = \bigsqcup\{[1, 1], [3, 3], [5, 5]\} = [1, 5]$. Second, for compound types, we define the embedding function component-wise. For example, for products we define the embedding function $\iota_{(A,B)} : (A, B) \to (\widehat{A}, \widehat{B})$ by $\iota_{(A,B)}(a, b) = (\iota_A(a), \iota_B(b))$, given embeddings $\iota_A : A \to \widehat{A}$ and $\iota_B : B \to \widehat{B}$. This approach naturally extends to other compound data types we face in Haskell, such as lists `[a]`, `Maybe a`, `Either a b`, and so on. Note that data types in Haskell also have a coinductive interpretation, e.g., lists can be infinite. However, in this work we only consider inductive interpretations of datatypes.

However, the construction of Galois connections with embedding functions $\iota : A \to \widehat{A}$ places requirements on the concrete domain A and the abstract domain $\widehat{A}$. First, it assumes that both domains have a preorder $\sqsubset_A$ respectively $\sqsubset_{\widehat{A}}$. Second, it assumes that the abstract domain $\widehat{A}$ is finitely complete, that is, all elements x and y have a least upper bound $x \sqcup_{\widehat{A}} y$. While it is easy to define preorders for the types occurring in our interpreter, these orders often are not finitely complete. For example, type `Either Int String` has no least upper bound for `Left 5` and `Right "x"`. Fortunately, we can lift a non-completely ordered type X to a finitely complete ordered type $X^\top$. The lifting $X^\top$ adds a greatest element $\top$ to the type X, such that all incomparable elements now have a least upper bound:

$$x_1 \sqsubseteq_{X^\top} x_2 \text{ iff } x_2 = \top \vee x_1 \sqsubseteq_X x_2$$

For example, the lifted type $(\texttt{Either Int String})^\top$ has all least upper bounds, such as $(\texttt{Left 5}) \sqcup (\texttt{Right "x"}) = \top$.

Based on embedding functions ι_X, partial orders $\sqsubset_X$, and the lifting $X^\top$, we can systematically construct Galois connections for all types that occurring in our interpreters. What is left, is to define the soundness proposition for arrow types `Interp` and $\widehat{\texttt{Interp}}$.

3.2 Soundness Proposition for Arrows

It is not possible to give a general definition of a soundness proposition for arbitrary arrows, because arrows and their soundness propositions are analysis-specific. However, we can define a soundness proposition for specific classes of arrows. In this section, we define a soundness proposition for Kleisli arrows [Hughes 2000]. Kleisli arrows are functions $A \to M(B)$ parameterized by a monad M. It is well-known that monads are expressive enough to describe a wide range of effects in programming languages [Liang et al. 1995; Moggi 1991; Wadler 1995]. For example, we can describe the two interpreter arrows of section Section 2.2 as Kleisli arrows:

$$\texttt{Interp}(A, B) = A \to M(B) \qquad \widehat{\texttt{Interp}}(A, B) = A \to \widehat{M}(B)$$
$$M(B) = \texttt{Env} \to \texttt{Maybe}\ B \qquad \widehat{M}(B) = \widehat{\texttt{Env}} \to \widehat{\texttt{Maybe}}\ B$$

This way, Kleisli arrows and their soundness proposition serve as a good starting point to define analysis-specific soundness propositions.

We define the soundness proposition for Kleisli arrows for the *forward collecting semantics* [Cousot and Cousot 1992] of the concrete interpreter. The forward collecting semantics of a

function $f : A \to B$ describes the strongest post-condition $\{f(x) \mid x \in X\}$ of f under a pre-condition $X \subseteq A$ over the inputs of f. For example, the strongest post-condition for $f(x) = x + x$ for the pre-condition $\mathbb{N}$ is the set of even numbers. In our scenario, we describe the forward collecting semantics of $f : A \to B$ as a single function $\lambda X.\{f(x) \mid x \in X\}$ of type $\mathcal{P}(A) \to \mathcal{P}(B)$. Before we can define the soundness proposition for Kleisli arrows, we first need to define a Galois connection between the forward collecting semantics of the concrete Kleisli arrow and the $\top$-lifted abstract Kleisli arrow on the underlying function space [Nielson et al. 1999, page 253]:

$$\alpha_{A,M(B)} : (\mathcal{P}A \to \mathcal{P}(M(B))) \rightleftarrows \left(\widehat{A}^\top \to \widehat{M}(\widehat{B})^\top\right) : \gamma_{A,M(B)}$$
$$\alpha_{A,M(B)}(f) = \alpha_{M(B)} \circ f \circ \gamma_{\widehat{A}} \qquad \gamma_{A,M(B)}(\widehat{f}) = \gamma_{\widehat{M}(\widehat{B})} \circ \widehat{f} \circ \alpha_A$$

The Galois connection for Kleisli arrows uses Galois connections $\alpha_A : \mathcal{P}A \leftrightarrows \widehat{A}^\top : \gamma_{\widehat{A}}$ and $\alpha_{M(B)} : \mathcal{P}(M(B)) \leftrightarrows \widehat{M}(\widehat{B})^\top : \gamma_{\widehat{M}(\widehat{B})}$ constructed with the techniques described in Section 3.1. With the Galois connection between the concrete and abstract Kleisli arrows, we are ready to state soundness proposition for Kleisli arrows.

Definition 1 (Soundness proposition for Kleisli arrows)**.** Let `Interp` and $\widehat{\texttt{Interp}}$ be Kleisli arrows. Then, a computation $f \in \texttt{Interp}(A, B)$ is sound with respect to a computation $\widehat{f} \in \widehat{\texttt{Interp}}(A, B)$

$$f \mathrel{\dot{\sqsubseteq}} \widehat{f} \quad \text{iff} \quad \alpha_{A,M(B)}(\lambda X.\{f(x) \mid x \in X\})) \sqsubseteq \widehat{f}^\top$$

In this definition, $\widehat{f}^\top$ is the $\top$-lifting of function $\widehat{f}$:

$$\widehat{f}^\top(x) = \begin{cases} \top, & x = \top \\ \widehat{f}(x), & x \neq \top \end{cases}$$

This definition is well-defined for Kleisli arrows over any types A and B for which Galois connections α_A and $\alpha_{M(B)}$ exist. Given these Galois connections, we can use this soundness proposition for all parts of the interpreters, making it a key ingredient for constructing compositional soundness proofs.

4 COMPOSITIONAL SOUNDNESS FOR ARROW-BASED ABSTRACT INTERPRETERS

In this section, we present how our framework enables compositional soundness proofs and we prove that the composition always succeeds. Our framework is language-agnostic and can be used for any abstract interpreter satisfying the following two requirements:

- The concrete interpreter and abstract interpreter must share their implementation. That is, `eval = fix eval'` and $\widehat{\texttt{eval}} = \widehat{\texttt{fix}}$ `eval'` for some `eval'`.
- The shared interpreter `eval'` must be an arrow computation.

The first requirement enables compositional soundness proofs, because the proof can be decomposed along the structure of the shared code. The second requirement ensures that the recomposition of subproofs must succeed. Together, they provide a powerful framework where all shared code is sound by construction and users only have to prove soundness for the differing code: the concrete and abstract implementations of arrow operations.

Arrows induce an induction principle because arrow notation [Paterson 2001] (used throughout the examples in this paper) fully desugars to operations of the arrow type classes and the residual code does not contain any non-arrow constructs of the meta-language anymore. Furthermore, the arrow type classes can be described by an endofunctor F [Hamana and Fiore 2011] and the arrow instances as algebras of this endofunctor. The initial F-algebra induces the desired induction principle. For example, the initial F-algebra for the shared interpreter of Figure 3 is described by the following generalized algebraic datatype (GADT) that enumerates all arrow expressions that can be described over the `IsVal` type class:

```
data AExp :: C -> C -> Set where
  Lit :: AExp Int v
  Add :: AExp (v,v) v
  Lookup :: AExp String v
  IfZero :: AExp x v -> AExp y v -> AExp (v,(x,y)) v
  Try :: AExp x y -> AExp y v -> AExp x v -> AExp x v
  (>>>) :: AExp x y -> AExp y z -> AExp x z
  (***) :: AExp x y -> AExp u v -> AExp (x,u) (y,v)
  (|||) :: AExp x z -> AExp y z -> AExp (Either x y) z
  Arr_1 :: AExp A_1 B_1    ...    Arr_n :: AExp A_n B_n
```

The datatype contains one constructor for each operation of the `IsVal` type class and its superclasses `Arrow` and `ArrowChoice`. It does not contain an operation for the fixpoint combinator, which requires special treatment as we discuss later. Besides these arrow operations, the desugaring arrow computations also generates pure functions that are embedded into arrow computations using the `arr` operation. To avoid a higher-order constructor `Arr :: (a -> b) -> AExp a b`, we enumerate each of the pure functions as individual constructors $\texttt{Arr}_i$. The initial F-algebra `AExp` then induces the following induction principle for predicates P.

$$\frac{\begin{array}{c} P(\texttt{Lit}) \qquad P(\texttt{Add}) \qquad P(\texttt{Lookup}) \\ P(f_1) \wedge P(f_2) \implies P(\texttt{IfZero}\ f_1\ f_2) \\ P(f_1) \wedge P(f_2) \wedge P(f_3) \implies P(\texttt{Try}\ f_1\ f_2\ f_3) \\ P(f_1) \wedge P(f_2) \implies P(f_1 \texttt{>>>} f_2) \\ P(f_1) \wedge P(f_2) \implies P(f_1 \texttt{***} f_2) \\ P(f_1) \wedge P(f_2) \implies P(f_1 \texttt{+++} f_2) \\ P(\texttt{Arr}_1) \qquad \ldots \qquad P(\texttt{Arr}_n) \end{array}}{\forall \texttt{A}, \texttt{B} \in \mathbf{C}.\ \forall e \in \texttt{AExp A B}.\ P(e)}$$

This induction principle allows us to decompose soundness proofs because of the shared implementation. Specifically, we set

$$P(e) \text{ iff } e \mathrel{\dot{\sqsubseteq}} \widehat{e},$$

where e refers to the concrete instance of the arrow code and $\widehat{e}$ to the abstract instance, i.e., the respective F-algebra. With this predicate, the premises of the induction principle exactly correspond to the *soundness preservation lemmas* discussed in Section 2.3. For example:

$$f_1 \mathrel{\dot{\sqsubseteq}} \widehat{f_1} \wedge f_2 \mathrel{\dot{\sqsubseteq}} \widehat{f_2} \implies (f_1 \texttt{>>>} f_2) \mathrel{\dot{\sqsubseteq}} (\widehat{f_1} \widehat{\texttt{>>>}} \widehat{f_2})$$

Thus, the induction principle shows that *all* shared arrow code is sound if the soundness preservation lemmas hold. This is the essence of decomposing the soundness proof of an arrow-based abstract interpreter.

However, before we can state our main soundness theorem, we need to add support for fixpoint combinators. In Section 2.3, we applied concrete and abstract fixpoint combinators `fix` and $\widehat{\texttt{fix}}$ to the shared interpreter. Since fixpoint combinators are higher-order functions of the form

$$\texttt{Fix} : (\texttt{AExp A B} \to \texttt{AExp A B}) \to \texttt{AExp A B},$$

adding them to our GADT would break the induction principle, because the datatype would not be strictly positive [Coquand and Paulin-Mohring 1990]. Instead, we adapt the soundness proposition for fixpoint combinators by Cousot and Cousot [1992, Proposition 4.3]:

Definition 2. A fixpoint combinator `fix` is sound with respect to $\widehat{\texttt{fix}}$ iff $\texttt{fix}\ f \mathrel{\dot{\sqsubseteq}} \widehat{\texttt{fix}}\ f$ for all soundness preserving functions $f_C : C(A,B) \to C(A,B)$, that is:

$$\left[\forall x, \widehat{x}.\, x \mathrel{\dot{\sqsubseteq}} \widehat{x} \Rightarrow f(x) \mathrel{\dot{\sqsubseteq}} f(\widehat{x})\right] \implies \texttt{fix}\ f \mathrel{\dot{\sqsubseteq}} \widehat{\texttt{fix}}\ f.$$

Now we are ready to state our main soundness theorem:

Theorem 3 (Soundness of Abstract Interpreters based on Arrows). *For a given concrete interpreter* `eval : Interp`(A,B) *and abstract interpreter* $\widehat{\texttt{eval}} : \widehat{\texttt{Interp}}(A,B)$ *defined by* `eval = fix eval'` *and* $\widehat{\texttt{eval}} = \widehat{\texttt{fix}}\ \texttt{eval}'$ *with a shared implementation* $\texttt{eval}'_C : C(A,B) \to C(A,B)$ *(natural in C [Mac Lane 1978])*[1] *over a functor F with an initial algebra, soundness* $\texttt{eval} \mathrel{\dot{\sqsubseteq}} \widehat{\texttt{eval}}$ *follows from (i) soundness of the fixpoint combinators* `fix` *and* $\widehat{\texttt{fix}}$ *and (ii) the soundness preservation lemmas of F.*

Proof. From the soundness proposition of the fixpoint combinators, we know that $\texttt{eval} \mathrel{\dot{\sqsubseteq}} \widehat{\texttt{eval}}$ if $\texttt{eval}'(x) \mathrel{\dot{\sqsubseteq}} \texttt{eval}'(\widehat{x})$ for all $x \in \texttt{Interp}(A,B), \widehat{x} \in \widehat{\texttt{Interp}}(A,B)$ with $x \mathrel{\dot{\sqsubseteq}} \widehat{x}$. Because `eval'` is natural in the arrow type C, the arrow expressions $\texttt{eval}'(x)$ and $\texttt{eval}'(\widehat{x})$ have the same structure except for occurrences of x and $\widehat{x}$. Thus $\texttt{eval}'(x) \mathrel{\dot{\sqsubseteq}} \texttt{eval}'(\widehat{x})$ follows by structural induction, the soundness preservation lemmas, and the assumption $x \mathrel{\dot{\sqsubseteq}} \widehat{x}$. □

Thus, to prove an abstract interpreter based on arrows sound, it suffices to use a sound fixpoint combinator and to verify the soundness preservation lemmas. Since each soundness preservation lemma is concerned with a single arrow operation only, the soundness proof of the abstract interpreter decomposes into small, manageable proof obligations.

The naturality of $\texttt{eval'}_C$ in the arrow type C is crucial in this proof of Theorem 3. It ensures that the shared interpreter does not produce a structurally different arrow expression when instantiated with the concrete and abstract arrow types. Only if the structure of the interpreters is the same, we can apply the induction principle. In general, we can ensure this if the shared interpreter is parametric in the arrow type.

One shortcoming of our proof method, though, is the handling of the pure functions $\texttt{Arr}_1 \ldots \texttt{Arr}_n$ that the arrow desugaring generates. Proving soundness for each pure function is tedious and usually uninteresting. In the next section, we use parametricity [Reynolds 1983], a property of parametric polymorphism, to describe interface guidelines such that *all* pure functions are sound by a free theorem of parametricity.

5 INTERFACE DESIGN AND PARAMETRICITY

The main goal of this paper is to reason about soundness of the operations of the interpreters, rather than about composed code of the shared interpreter itself. The design of the interface influences how much reasoning about shared code is necessary, if any at all. In this section, we provide guidelines for how to design interfaces such that soundness of pure functions follows as a free theorem of parametricity.

To this end, let us revisit the interface for `IfZero` from Section 2.2:

```
ifZero :: c x v -> c y v -> c (v,(x,y)) v
```

Instead of providing two continuations that are called when the argument value is zero or not, we could have designed an operation `isZero`, that returns a Boolean value that represents its outcome:

```
data Bool^ = True | False | Top
isZero :: c v Bool^

eval' ev = proc e -> case e of
```

[1] $\texttt{eval'}_C$ is natural in C iff for all $f : C(A,B) \to D(A,B)$, $f \circ \texttt{eval}'_C = \texttt{eval}'_D \circ f$

```
IfZero e1 e2 e3 -> do
  b <- isZero <<< ev -< e1
  case b of
    True -> ev -< e2
    False -> ev -< e3
    Top -> (ev -< e2) ⊔ (ev -< e3)
```

The value `Top` is solely used by the abstract interpreter to express uncertainty about whether a value is zero. The concrete instance of `isZero` never returns `Top` because it is always certain if the value is zero. Although this definition describes an alternative but equivalent semantics, there are two problems:

(1) The shared interpreter now describes behavior that is specific to the abstract interpreter but not the concrete semantics. The interface of the shared interpreter *leaks* details of the abstract interpreter into shared code.
(2) Proving soundness of the instantiated shared interpreter requires reasoning about more code than just the arrow operations it is comprised of. In particular, we have to consider the entire case expression in the shared code to prove soundness. The interface design of `isZero` does not allow us to decompose the soundness proof.

But is there a metric that helps us identify interface operations that leak details of the abstract interpreter? The answer can be found in a property called *parametricity* [Reynolds 1983], a property of parametric polymorphism. The key idea of parametricity is that types can be interpreted as relations and terms in related environments yield related results [Wadler 1989].

To set the stage, we recall the definition of Reynolds' parametricity [Reynolds 1983] due to Ghani et al. [2015]. Well-typed System F programs e are identified by the typing judgment $\Gamma, \Delta \vdash e : \tau$, where τ is a type with type variables closed under Γ and Δ is the regular typing context. Parametricity describes two parallel interpretations for System F contexts, types and terms, that work in lock-step: An object interpretation $[\![T]\!]_o : \mathbf{Set}^{|\Gamma|} \to \mathbf{Set}$ that interprets types as sets and terms as functions, and a relational interpretation $[\![T]\!]_r : \mathbf{Rel}^{|\Gamma|}(A, B) \to \mathbf{Rel}([\![T]\!]_o A, [\![T]\!]_o B)$ that interprets types as relations and terms as relation preserving functions. Each interpretation takes extra arguments based on $|\Gamma|$, the number of type variables in the context Γ.

How these two interpretations interact is described by the following main theorem of parametricity [Reynolds 1983]:

Theorem 4 (Abstraction Theorem). *Let* $A, B \in \mathbf{Set}^{|\Gamma|}$, $R \in \mathbf{Rel}^{|\Gamma|}(A, B)$, $a \in [\![\Delta]\!]_o A$ *and* $b \in [\![\Delta]\!]_o B$. *For every term* $\Gamma, \Delta \vdash e : \tau$, *if* $(a, b) \in [\![\Delta]\!]_r R$, *then* $([\![e]\!]_o A\ a, [\![e]\!]_o B\ b) \in [\![\tau]\!]_r(R)$. □

More informally, if a and b are instances of the typing context Δ and are related by R, then a term e with context Δ applied to a and b are related by R. If we choose R to be the soundness proposition for arrow types, the abstraction theorem provides an alternative way to prove soundness of abstract interpreters with a shared implementation. We prove this as a theorem below. However, since arrows are higher-order types of kind $\star \to \star \to \star$, we in fact require the abstraction theorem for higher-order parametricity that holds for System F_ω [Atkey 2012]. The general idea of the abstraction theorem for first-order parametricity carries over to the one for higher-order parametricity. Therefore, we omit the definitions for higher-order parametricity for simplicity and brevity.

Theorem 5. *In System* F_ω, *soundness of abstract interpreters that share a common implementation with the concrete interpreter follows from the soundness lemmas for operations of its interface.*

Proof. First, we desugar the type class `IsValue` into a record that is passed in as a dictionary [Hall et al. 1996]. This allows us to type check `eval'` with the following judgement:

$$\{\mathtt{c} : * \to * \to *,\ \mathtt{v} : *\}, \{dict : \mathtt{IsValue\ c\ v}\} \vdash \mathtt{eval'} : \mathtt{c\ Expr\ v} \to \mathtt{c\ Expr\ v}$$

We now apply the abstraction theorem for higher-order parametricity as follows. The typing variable context has type variables for the arrow type `c` and value type `v`, hence, for A and B we choose the tuples $(\mathtt{Interp}, \mathtt{Val})$ and $(\widehat{\mathtt{Interp}}, \widehat{\mathtt{Val}})$ that instantiate the respective arrow and value type. Furthermore, for the relation R we have to define relations on arrows and values. For the relation on values, we choose $v \mathrel{\dot\sqsubseteq_{\mathtt{Val}}} \widehat{v}$ iff $\alpha_{\mathtt{Val}}(v) \sqsubseteq \widehat{v}$, where $\alpha_{\mathtt{Val}} : \mathcal{P}(\mathtt{Val}) \to \widehat{\mathtt{Val}}$ is the abstraction function for values. Because arrows are higher-kinded types, the relation on arrows is parameterized by relations R over the domain and Q over the codomain of the arrow. For the soundness relation on arrows, we choose

$$f \mathrel{\dot\sqsubseteq_{\mathtt{Interp}}} \widehat{f} \quad \text{iff} \quad (a, \widehat{a}) \in R \implies (f(a), \widehat{f}(\widehat{a})) \in Q \quad \text{for all } a \in A, \widehat{a} \in \widehat{A}.$$

If we instantiate R and Q with the relation $\alpha(x) \sqsubseteq \widehat{x}$, we obtain the original soundness proposition:

$$f \mathrel{\dot\sqsubseteq_{\mathtt{Interp}}} \widehat{f} \quad \text{iff} \quad \alpha_A(a) \sqsubseteq \widehat{a} \implies \alpha_{\widehat{A}}(f(a)) \sqsubseteq \widehat{f}(\widehat{a}) \quad \text{for all } a \in A, \widehat{a} \in \widehat{A}.$$

If we use the abstraction theorem with these definitions, we obtain the following rule.

$$\frac{\begin{array}{c} a \in [\![\mathtt{IsValue\ c\ v}]\!]_o(\mathtt{Interp}, \mathtt{Val}) \\ b \in [\![\mathtt{IsValue\ c\ v}]\!]_o(\widehat{\mathtt{Interp}}, \widehat{\mathtt{Val}}) \\ (a, b) \in [\![\mathtt{IsValue\ c\ v}]\!]_r(\dot\sqsubseteq_{\mathtt{Interp}}, \dot\sqsubseteq_{\mathtt{Val}}) \end{array}}{([\![\mathtt{eval'}]\!]_o(\mathtt{Interp}, \mathtt{Val})\ a, [\![\mathtt{eval'}]\!]_o(\widehat{\mathtt{Interp}}, \widehat{\mathtt{Val}})\ b) \in [\![\mathtt{c\ Expr\ v} \to \mathtt{c\ Expr\ v}]\!]_r(\dot\sqsubseteq_{\mathtt{Interp}}, \dot\sqsubseteq_{\mathtt{Val}})}$$

The rule says, given two instances a and b for the interface `IsValue` and a and b satisfy the soundness preservation lemmas of `IsValue`, then the shared interpreter `eval'` instantiated with the instance a is sound with respect to `eval'` instantiated with b. □

The main consequence of Theorem 5 is that we do not have to reason about soundness of shared code, since it follows as a free theorem from parametricity. In particular, this relieves us from having to prove soundness of individual pure functions in `arr`. Instead, we obtain a generic soundness lemma for the `arr` operation itself:

$$(\mathtt{arr}, \widehat{\mathtt{arr}}) \in [\![\forall \mathtt{x}, \mathtt{y}.\ (\mathtt{x} \to \mathtt{y}) \to \mathtt{c\ x\ y}]\!]_r(\dot\sqsubseteq_{\mathtt{Interp}}).$$

Because all pure functions f in the shared interpreter are shared code, this lemma guarantees $(\mathtt{arr}\ f \mathrel{\dot\sqsubseteq} \widehat{\mathtt{arr}}\ f)$.

Theorem 5 can also help us understand how to design the interface such that the each arrow operation is compositionally sound. When a soundness proof for an arrow operation fails, it usually fails with the approach based on parametricity as well as with the approach from Section 4. However, the approach based on parametricity can tell us why a proof failed. To this end, it is instructive to compare the soundness lemmas of Theorem 5 to the corresponding soundness lemmas of Theorem 3. For example, for the composition operator >>>, Theorem 5 requires

$$(>\!>\!>, \widehat{>\!>\!>}) \in [\![\forall \mathtt{x}, \mathtt{y}, \mathtt{z}.\ \mathtt{c\ x\ y} \to \mathtt{c\ y\ z} \to \mathtt{c\ x\ z}]\!]_r(\dot\sqsubseteq_{\mathtt{Interp}})$$

whereas Theorem 3 requires

$$f_1 \mathrel{\dot\sqsubseteq} \widehat{f}_1 \wedge f_2 \mathrel{\dot\sqsubseteq} \widehat{f}_2 \implies (f_1 >\!>\!> f_2) \mathrel{\dot\sqsubseteq} (\widehat{f}_1 \widehat{>\!>\!>} \widehat{f}_2).$$

The soundness lemmas have almost the same meaning, except that the orderings used in the former lemma are fixed by the relational interpretation $[\![-]\!]_r$ rather than chosen by us. This is an important

distinction because it restricts how we can design our interface, while still being able to prove soundness compositionally.

For example, let us revisit the flawed version of `isZero` introduced earlier in this section. Observe that we cannot prove $\texttt{ifZero} \dot{\not\sqsubseteq} \widehat{\texttt{ifZero}}$ using parametricity either:

$$(\texttt{isZero}, \widehat{\texttt{isZero}}) \notin [\![\texttt{c} \ \texttt{v} \ \widehat{\texttt{Bool}}]\!]_r(\dot{\sqsubseteq}_{\texttt{Interp}}, \dot{\sqsubseteq}_{\texttt{Val}})$$

The problem is that the ordering for $\widehat{\texttt{Bool}}$ is determined by its relational interpretation based on the underlying sum type:

$$[\![\widehat{\texttt{Bool}}]\!]_r = \{(\texttt{True}, \texttt{True}), (\texttt{False}, \texttt{False}), (\texttt{Top}, \texttt{Top})\}.$$

However, we require that `Top` is the greatest element to be able to prove the soundness lemma for `isZero`, which is not the case for this ordering. The underlying problem is that we exposed the type $\widehat{\texttt{Bool}}$ with non-standard ordering to the shared interpreter. This problem exists not only for $\widehat{\texttt{Bool}}$, but for all types with non-standard ordering, such as values, environments, etc.

These observations lead us to the following guideline for good interface design of shared interpreters, helping us to avoid leaking interfaces:

Guideline. An interface of a shared interpreter is good if its operations do not expose types with non-standard orderings. Instead, non-standard ordered types in the abstract interpreter must be hidden from the interface by using universal quantification.

To summarize, the abstraction theorem for meta-languages with parametricity provides an alternative way to prove soundness of abstract interpreters that share code. This drastically reduces the required effort of the soundness proof, because shared code is sound by a free theorem of parametricity. Furthermore, the abstraction theorem provides us with a useful guideline for how to design our interface. Finally, nothing in the proof of Theorem 5 is specific to arrows. In particular, we are not making use of the induction principle for arrows and use the abstraction theorem instead. This should allow us to apply Theorem 5 to abstract interpreters that share code with the concrete interpreter using an interface other than arrows. We have not explored this further so far.

6 CASE STUDIES

This paper presents a framework for compositional soundness proofs. In this section, we report on two case studies that we conducted to answer the following research questions:

(RQ1) Is our technique applicable to interesting languages and interesting static analyses?
(RQ2) Does our technique reduce the complexity and effort of soundness proofs?

The case studies involved constructing shared interpreters for Stratego and PCF, developing concrete and abstract arrow instances, and proving the instantiated interpreters sound. For Stratego, we developed a tree-shape analysis as abstract arrow instance; for PCF, we implemented an advanced control-flow analysis (k-CFA) as abstract arrow instance.[2]

6.1 Tree-Shape Analysis for Stratego

We developed a sound abstract interpreter for *Stratego* [Visser et al. 1998], a real-world language for the implementation of program transformations that operate on abstract syntax trees akin to s-expressions. Stratego is being used in various projects to define interpreters [Dolstra and Visser 2002], refactorings [de Jonge and Visser 2012], desugarings [Erdweg et al. 2011], and compilers

[2] All code of the case studies is open source and can be found at https://github.com/svenkeidel/sturdy/. The proofs can be found in the extended version of this paper at https://arxiv.org/.

[Avgustinov et al. 2007; Bagge and Kalleberg 2006; Economopoulos and Fischer 2011]. Furthermore, Stratego is used to compile programs of WebDSL [Visser 2007], a domain-specific web-programming language in which, for example, the website conf.researchr.org of ICFP and others is implemented [van Chastelet et al. 2015].

Stratego transformations operate on untyped terms using rewrite rules and strategies as illustrated by the following simple evaluator for arithmetic expressions:

```
rules
  reduce: Add(Succ(m), n) -> Succ(Add(m,n))
  reduce: Add(Zero(), n)  -> n

strategies
  main = downup(try(reduce))
```

The strategy `main` walks the expression tree down and up again, and tries to reduce each visited node using the rewriting rule `reduce`. Rule `reduce` consists of two alternatives `reduce: pat -> gen` that try to match `pat` and, if successful, generate `gen`. Stratego has many language features that make it a challenging language to statically analyze, including dynamic scoping of pattern-bound variables, higher-order functions, and generic tree traversals.

Stratego provides a rich set of abstractions for program transformations. These abstractions desugar into a core language for Stratego with just 12 constructs [Bravenboer et al. 2006; Visser et al. 1998]. We developed a shared interpreter based on arrows for this core language. For the interface of the shared interpreter, we identified 27 operations, of which 9 operations are language-independent and 18 operations are specific to Stratego. The language-specific operations consist of 10 operations for terms, 6 for term environments, and 2 for strategy environments.

We instantiated the shared interpreter with Kleisli arrows for the concrete and abstract domain. The concrete domain uses the usual interpretation of terms and environments. In the abstract domain, we approximate terms as a set of term patterns containing wildcards $*$. For example, the abstract term $\{\texttt{Zero}(), \texttt{Add}(*, *)\}$ represents the set of concrete terms containing `Zero()` and all terms with root `Add`. This way, our abstract arrow instance realizes a tree-shape analysis [Keidel and Erdweg 2017] that Stratego developers can use to predict the shape of trees a transformation will produce when run.

For the concrete instance of `ArrowFix`, we compute the usual least fixpoint. However, since the abstract domain of sets of term patterns is infinite, the least fixpoint is not computable for the abstract domain. Therefore, for the abstract instance of `ArrowFix`, we approximate the greatest fixpoint instead. Specifically, our fixpoint combinator keeps track of the recursive depth of the interpreter and yields $\top$ for recursive calls whose depth exceeds a certain threshold. This produces a finite approximation of the infinite set of terms that can be produced by a given program transformation. The precision of the abstract interpreter increases with more iterations.

We have verified the soundness of our tree-shape analysis by proving that abstract instantiations of the shared interpreter approximates the concrete instantiation. The soundness proof is completely compositional. We decomposed the proof into 27 soundness lemmas, one for each operation in the interface of the shared interpreter. All operations in the interface conform to the guidelines of interface design of Section 5, and hence soundness of all pure `arr` expressions follows as a free theorem due to the parametricity of our meta-language Haskell. The soundness of the instantiated interpreters then follows from Theorem 3 and the 27 soundness lemmas.

To reflect on the complexity and effort of the soundness proof (RQ2), we want to highlight the soundness proof of the implementation of strategy calls. We show the code of the shared interpreter in Figure 6. A strategy in Stratego accepts two kinds of arguments, strategy arguments and term

```
call :: ... => StratVar -> [Strat] -> [TermVar] -> (Strat -> c t t) -> c t t
call f actualStratArgs actualTermArgs ev = proc a -> do
  senv <- readStratEnv -< ()
  case Map.lookup f senv of
    Just (Closure formalStratArgs formalTermArgs body senv') -> do
      tenv <- getTermEnv -< ()
      mapA bindTermArg -< zip actualTermArgs formalTermArgs
      let senv'' = foldl bindStratArgs (if Map.null senv' then senv else senv')
                                       (zip formalStratArgs actualStratArgs)
      b <- localStratEnv senv'' (ev body) -<< a
      tenv' <- getTermEnv -< ()
      putTermEnv <<< unionTermEnvs -< (formalTermArgs,tenv,tenv')
      returnA -< b
    Nothing -> fail -< ()
  where
    bindTermArg = proc (actual,formal) ->
      lookupTerm (proc t -> insertTerm -< (formal,t)) fail -<< actual

    bindStratArgs senv (v,Call v' [] []) senv =
      case Map.lookup v' senv of
        Just s -> Map.insert v s senv
        _ -> error $ "unknown strategy: " ++ show v'
    bindStratArgs senv (v,s) = Map.insert v (Closure [] [] s senv) senv
```

Fig. 6. Shared implementation of calls of strategies.

arguments. Hence, the interpreter has to bind these two kinds of arguments in the respective environment and then invoke the interpreter recursively on the body of the called strategy.

Traditionally, proving soundness of the concrete and abstract instantiations of this code is a severe challenge: The complexity of the code would be reflected in the proof. With our technique, we can decompose the proof into 2 soundness lemmas about strategy environments (`readStratEnv`, `localStratEnv`), 6 lemmas about term environments (`lookupTerm`, `insertTerm`, `unionTermEnvs`, `getTermEnv`, `putTermEnv`), a few lemmas about language-independent arrow operations, and various lemmas about embedded pure functions. Each of these lemmas is manageable and can be proved in isolation, thus reducing the proof complexity. Our approach also reduces the proof effort. First, some lemmas can be reused in other cases of the interpreter, such as the ones for term environments, which are needed for pattern matching as well. Second, we obtain the soundness lemmas for applications of embedded pure functions `Map.lookup`, `zip`, and `foldl` as free theorems of parametricity. And third, the soundness of the shared interpreter follows for free from the induction principle of Theorem 3.

In summary, we developed an arrow-based shared interpreter for Stratego together with a concrete and an abstract arrow instance. The abstract arrow instance realizes a tree-shape analysis for Stratego. We compositionally proved this analysis sound by verifying 27 smaller and individually provable lemmas. Thus, our technique was applicable to this scenario (RQ1) and, as we argued, the resulting proof is less complex and required less effort than a traditional proof (RQ2).

6.2 Control-Flow Analysis for PCF

We implemented an abstract interpreter for an analysis that has been widely studied in the literature [Midtgaard 2012]: control-flow analysis (CFA). We implemented this analysis for PCF [Plotkin 1977], a language with first-class functions, numbers, an `ifZero` construct, and fixpoint combinator

```
data Val
  = ClosureVal (Expr,Env)
  | NumVal Int
type Env = Map String Val
```

```
data Val^ = Top
  | ClosureVal^ (Set (Expr,Env^))
  | NumVal^ Interval
type Env^ = Map String Addr
type Store^ = Map Addr Val^
```

Fig. 7. Concrete (left) and abstract domain (right) for k-CFA analysis of PCF.

Y. The analysis we implemented is a k-CFA analysis [Shivers 1991] and the fixpoint algorithm we used is due to Darais et al. [2017].

We briefly summarize how our analysis works. The analysis approximates functions (closures) as sets of expression and environment pairs, while natural numbers are approximated using bounded intervals. We ensure termination by employing Darais et al.'s fixpoint algorithm for big-step semantics [2017]. Darais et al.'s fixpoint algorithm memoizes the results of all interpreter calls in a cache. When the interpreter is called with the same expression and environment recursively or repeatedly, it returns the cached result instead of recursing. This guarantees termination since there are only finitely many environments to consider and the interpreter repeats itself eventually. To finitely approximate environments, we adopt a common approach for (k-)CFA [Horn and Might 2010; Shivers 1991]: We allocate the values of an environment in an abstract store that has only finitely many addresses available. There are only finitely many stores if all abstract values are finite domains. For closures this is the case, because there only finitely many expressions that can be evaluated for a given program. And our abstract domain for numbers is finite, because we restrict the maximum bounds of intervals. If an interval exceeds theses bounds, it is approximated with infinity. We summarize the concrete and abstract domain of the k-CFA interpreter in Figure 7.

Figure 8 shows the shared PCF interpreter and the interface that we developed for it. The interface has a total of 16 operations: 4 value operations (class `IsVal`), 2 closure operations (`IsClosure`), 4 environment operations (`ArrowEnv`), a fixpoint operation (`ArrowFix` from Section 2.2), a failure operation (`ArrowFail`), and 4 language independent arrow operations (`Arrow`, `ArrowChoice`). We developed two instances of the interface: A concrete instance and a k-CFA instance. The code of these instances can be found in the artifact of our paper and its accompanying documentation.

We compositionally proved the soundness of k-CFA instantiated interpreter relative to the concrete instantiation of the interpreter. We decomposed the soundness proof into 16 lemmas, one for each operation of the arrow type classes referenced by the shared interpreter. Soundness of all pure `arr` expressions followed by parametricity of the meta-language Haskell (Theorem 5). As is common in proofs by induction, often the induction hypothesis has to be strengthened such that all cases of the induction are provable. We encountered this situation when proving soundness of the environment operations in the `ArrowEnv` type class. We had to strengthen the soundness proposition to guarantee that all environments passed in and out of the abstract arrow operation are consistent with the abstract store, i.e., all environment-bound addresses exist in the store. Note that this strengthened requirement of store consistency is not an artifact of using our techniques: It is necessary for non-compositional soundness proof as well.

To assess the complexity of our proof, we compare it to another proof of a k-CFA for a PCF-like language that can be found in the PhD thesis of Darais [2017]. The proof in Darais' PhD thesis relates in three theorems four different semantics, each proven by induction over derivations. It is not obvious how the cases of the induction can be decomposed further systematically, because of the differences between the concrete and abstract semantics. In comparison, our proof consists of 16 soundness lemmas that relate the concrete and abstract instances directly. The lemmas prove

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

class IsVal v c where
  succ :: c v v
  pred :: c v v
  zero :: c () v
  ifZero :: c x v -> c y v -> c (v,(x,y)) v

class IsClosure v env c where
  closure :: c (Expr,env) v
  apply :: c ((Expr,env),v) v -> c (v,v) v

class ArrowEnv var val env c where
  lookup :: c var (Maybe val)
  getEnv :: c () env
  extendEnv :: c (var,val,env) env
  localEnv :: c x y -> c (env,x) y

class ArrowFail c where
  fail :: c () x

apply' ev = apply $
  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')
      apply' ev -< (fun',arg)
    _ -> fail -< ()
```

```
eval' :: (IsVal v c, IsClosure v env c,
  ArrowChoice c, ArrowFix Expr v c,
  ArrowEnv Text v env c, ArrowFail c)
  => c Expr v
eval' = fix $ \ev -> proc e -> case e of
  Var x -> do
    m <- lookup -< x
    case m of
      Just v -> returnA -< v
      Nothing -> fail -< ()
  Lam x e -> do
    env <- getEnv -< ()
    closure -< (Lam x e, env)
  App e1 e2 -> do
    fun <- ev -< e1
    arg <- ev -< e2
    apply' ev -< (fun, arg)
  Zero -> zero -< ()
  Succ e -> do
    v <- ev -< e
    succ -< v
  Pred e -> do
    v <- ev -< e
    pred -< v
  IfZero e1 e2 e3 -> do
    v1 <- ev -< e1
    ifZero ev ev -< (v1, (e2, e3))
  Y e -> do
    fun <- ev -< e
    env <- getEnv -< ()
    arg <- closure -< (Y e, env)
    apply' ev -< (fun, arg)
```

Fig. 8. Interface and shared interpreter for PCF.

smaller pieces of functionality than the induction cases in Darais' proof. For example, the shared interpreter in Figure 8 uses a helper function `apply'` to apply a closure value to an argument value. Since we had proven the soundness of the language-independent arrow operations, the soundness proof for the shared code in `apply'` decomposed into just 3 soundness lemmas about interface operations: one for `apply`, the operation that unpacks a closure; one for `extendEnv`, the operation that extends the environment with an argument value; and one for `localEnv`, the operation that interprets under the extended environment. The functionality of `apply'` requires a manual proof in Darais' thesis, but in our setting, we get soundness of `apply'` for free because it is shared code and is sound by Theorem 5. There are of course also commonalities between the proofs, most significantly, we borrow the soundness lemma for fixpoints from Darais.

In summary, we developed a k-CFA analysis for PCF in our framework. We compositionally proved this analysis sound by verifying 16 smaller and individually provable lemmas. Thus, our techniques can be used to prove soundness of k-CFA, an interesting and widely studied static

analysis (RQ1). As we argued, the resulting proof is less complex and required less effort than a traditional proof (RQ2).

7 RELATED WORK

Our work is a continuation of a long line of research on constructing and proving the soundness of abstract interpreters. We have already related to many relevant sources throughout this paper. Here, we discuss related work in more detail.

One of the main ideas of abstract interpretation is to systematically *derive* a sound static analysis from a concrete semantics, by using the soundness proposition and proof as the guiding principle. Cousot and Cousot [1979] pioneered the approach, which has since been extended to a wide range of domains and semantic styles [Cousot 1999]. Such derivations enable soundness proofs that follow a systematic sequence of derivation and proof steps. But these proof steps can be involved, especially for interesting languages where one case of the abstract interpreter relates to many cases of the concrete interpreter. The focus of our work is to minimize the effort and complexity involved in proving soundness. We achieve this by factoring the concrete and abstract interpreter into a shared implementation that is parameterized over an arrow-based interface. The abstract instance of that interface can still be derived using techniques described by Cousot [1999]. However, in our experience, a soundness proof after the definition is easier because the proof goal is clear and progress can be made from either concrete and abstract side.

The idea of defining a language by implementing an interpreter in a meta-language (definitional interpreters) was famously described by Reynolds [1998]. In the context of abstract interpretation, the idea was explored even earlier by Jones and Nielson [1994], who describe an approach that translates expressions of the object language into expressions of a suitable meta-language. Constructs of the meta-language then have two different interpretations, one that recovers to the concrete semantics and one that recovers the abstract semantics of the object language. As a reasoning principle for soundness, the authors define a logical relation [Plotkin 1980] over the meta-language. The main benefit from using a logical relation is, soundness of all programs in the meta-language follows from soundness lemmas for each meta-language construct. The logical relation has to be proven when the meta-language is created and maintained when the meta-language changes. Compared to this paper, we use arrows as a meta-language and their induction principle as reasoning tool for soundness. This induction principle is very similar to a logical relation as it allows us to prove soundness of any arrow expression from soundness lemmas for each arrow operation. However, the main benefit of this induction principle is that we do not need to prove or maintain the induction principle itself. The induction principle follows for free from the fact that we use arrows, which are a first order language and can be expressed by an algebraic datatype.

The topic of definitional abstract interpreters was also recently revisited by Darais et al. [2017]. They show that an abstract definitional interpreter inherits properties of the meta-language, such as push-down control-flow precision. Similarly to our work, the concrete and abstract interpreter that Darais et al. present share code, but over a monadic interface instead of one based on arrows. Another similarity is that the abstract interpreters that we present can be regarded as definitional abstract interpreters, since we are using a meta-language to define our interpreters. The main difference between the work of Darais et al. is that we use a restricted meta-language (arrows), not necessarily as a means to inherit functional properties, but as a means to making soundness proofs compositional, which was not the focus of Darais et al.. We provide a generic theorem that ensures the soundness of an arrow-based abstract interpreter based on the soundness lemmas of the arrow operations, and we use parametricity to obtain soundness of embedded pure functions for free.

Monadic abstract interpreters by Sergey et al. [2013] show that concepts in static analysis such as context-sensitivity, poly-variance, flow-sensitivity, etc. are independent of any particular language

semantics and can be captured by an appropriate monad. These results carry over to our abstract interpreters based on arrows, because every monad gives rise to a Kleisli arrow. Sergey et al. describe their semantics using a shared monadic small-step abstract machine, but do not address the question of how to prove soundness of monadic abstract interpreters. We address soundness in this paper by factoring concrete and abstract interpreter into a shared big-step interpreter, which enables compositional soundness proofs. The usage of arrows provides an induction principle, which allowed us to ensure the soundness of the abstract interpreter by construction of sound arrow operations. We expect that it is possible to define a small-step abstract machine in the style of Sergey et al., but using arrows instead of monads in a way that our generic theorems apply.

Galois transformers and modular abstract interpreters by Darais et al. [2015] represent a systematic way to construct monadic abstract interpreters. Galois transformers are monad transformers, whose monadic operations can be proven sound with respect to each other. While our technique decomposes a soundness proof along operations of an interface, Galois transformers decompose a soundness proof along a monad transformer stack. For example, the operations `get` for fetching and `put` for writing state can be proven sound with respect to the concrete and abstract state monad transformer, independent of the rest of the monad transformer stack. The technique described in our paper and Galois transformers complement each other: Galois transformers still require a way to compose the lemmas of operations to a proof of the interpreters, which we provide. And our technique can benefit from decomposing the proof of soundness lemmas even further. In the future we want to combine these two approaches by using arrow transformers to achieve an even larger degree of proof decomposition.

Abstracting abstract machines (AAM) by Horn and Might [2010] is a technique for deriving sound abstract interpreters from concrete language semantics described as abstract machines. The concrete semantics is transformed in multiple steps to an abstract machine that is suitable to be approximated by an abstract interpreter. Each step of the transformation is systematic and preserves soundness with respect to the original concrete semantics. A consequence of this approach is that there must be a one-to-one correspondence between transitions in the concrete and abstract semantics. As we have discussed in Section 2, this is often not the case, for example, for `ifZero` over the interval domain. In contrast, our approach only requires a one-to-one correspondence between concrete and abstract arrow operations, but allows for a mismatch within these operations: An abstract operation can distinguish m cases even if the corresponding concrete operation distinguishes n cases.

Cousot et al. [2006] describe a different technique of soundness proof composition which is orthogonal to ours: The technique is for composing separate abstract analyses by organizing them in a hierarchy, such that analyses further down in the hierarchy can be influenced by the output from analyses further up, but not the other way around. The focus of our paper is not on composing *different analyses*, but rather on composing a soundness proof for a shared abstract interpreter from reusable lemmas about the operations of the language being abstracted.

8 CONCLUSION

We have presented a novel technique for defining concrete and abstract interpreters by sharing code over an interface based on arrows. Such interpreters can be proven sound *compositionally*: Our Theorem 3 tells us how to compose such a proof, and reduces the effort of proving soundness to the effort of proving a context-free soundness lemma for each interface operation and each embedded pure function in the shared interpreter. Our Theorem 5 applies *parametricity* to obtain the soundness of the embedded pure functions *for free*, which further reduces the proof effort. We demonstrated the applicability of our technique by implementing two case study analyses and proving them sound: a tree-shape analysis for Stratego and a k-CFA analysis for PCF. Compared

to traditional soundness proofs abstract interpreters, our soundness proofs are less complex and require less effort because we were able to decompose large proof obligations into independent soundness lemmas, from which the soundness of the abstract interpreters follows by construction. In the future, we want to investigate how our technique scales to even more complicated languages and analyses.

A DESUGARING OF ARROW PRETTY NOTATION

The Haskell standard library defines arrow operations in type classes `Category`, `Arrow`, and `ArrowChoice`. We show the code of these type classes in Figure 9a.[3] Arrow pretty notation [Paterson 2001] provides a simpler notation for arrows in the style of the *do*-notation of monads. We show the EBNF grammar of Paterson's arrow pretty notation in Figure 9b. Arrow pretty notation desugars to the arrow operations of `Category`, `Arrow`, and `ArrowChoice`.

```
class Category c where
  id :: c x x
  (.) :: c y z -> c x y -> c x z

f >>> g = g . f

class Category c => Arrow c where
  arr :: (x -> y) -> c x y
  (***) :: c x y -> c u v ->
    c (x,u) (y,v)
  (&&&) :: c x y -> c x z ->
    c x (y,z)

class ArrowChoice c where
  (+++) :: c x y -> c u v ->
    c (Either x u) (Either y v)
  (|||) :: c x z -> c y z ->
    c (Either x y) z
```

(a) Arrow type classes in Haskell

$$
\begin{aligned}
\mathit{expr} ::= {} & \ldots \\
& \mid \texttt{proc}\ \mathit{pat}\ \texttt{->}\ \mathit{cmd}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{cmd} ::= {} & \mathit{expr}\ \texttt{-<}\ \mathit{expr} \\
& \mid \texttt{form}\ \mathit{expr}\ \mathit{cmd}_1\ \ldots\ \mathit{cmd}_n \\
& \mid \mathit{cmd}_1\ \mathit{op}\ \mathit{cmd}_2 \\
& \mid \kappa\ \mathit{pat}\ \texttt{->}\ \mathit{cmd} \\
& \mid (\mathit{cmd}) \\
& \mid \texttt{do}\ \{\ \mathit{stmt}_1;\ \ldots;\ \mathit{stmt}_n;\ \mathit{cmd}\ \} \\
& \mid \texttt{case}\ \mathit{expr}\ \texttt{of} \\
& \qquad \mathit{pat}_1\ \texttt{->}\ \mathit{expr}_1 \\
& \qquad \ldots \\
& \qquad \mathit{pat}_n\ \texttt{->}\ \mathit{expr}_n
\end{aligned}
$$

$$
\begin{aligned}
\mathit{stmt} ::= {} & \mathit{cmd} \\
& \mid \mathit{pat}\ \texttt{<-}\ \mathit{cmd} \\
& \mid \texttt{rec}\ \{\mathit{stmt}_1;\ \ldots \texttt{l}\ \mathit{stmt}_n\ \}
\end{aligned}
$$

(b) Arrow pretty notation

Fig. 9. Arrow type classes (left) and arrow pretty notation (right).

For example, `mapA` maps an effectful arrow computation `f` over a list of values.

```
mapA :: ArrowChoice c => c x y -> c [x] [y]
mapA f = proc l -> case l of
  [] -> returnA -< []
  (x:xs) -> do
    y <- f -< x
    ys <- mapA f -< xs
    returnA -< y:ys
```

[3] Their original definition is available here https://hackage.haskell.org/package/base/docs/Control-Arrow.html

It desugars to arrow expressions as follows:

```
mapA :: ArrowChoice c => c x y -> c [x] [y]
mapA f = arr (\l -> case l of
                []  -> Left ()
                (x:xs) -> Right (x,xs))
         >>>
         ( (arr (\() -> [])) |||
           (f *** mapA f >>> arr (\(y,ys) -> (y:ys)) )
```

The first arrow expression `arr` embeds a pure function into an arrow computation that destructs a list into an `Either` type. The result is then passed with `>>>` to a computation `|||` that encodes the two branches of the case distinction from before. The left branch of `|||` encodes the first case and returns the empty list. The right branch applies `f` and `mapA f` with `***` to the first and second component of the input tuple respectively, containing the first element and rest of the list. The outputs of `f` and `mapA f` are collected in a tuple and the last `arr` expression constructs the output list from its components.

ACKNOWLEDGEMENTS

This research was supported by DFG grant “Evolute”. We want to thank Robbert Krebbers and Arjen Rouvoet who provided helpful feedback and Jente Hidskes who helped us with the artifact.

REFERENCES

Robert Atkey. 2012. Relational Parametricity for Higher Kinds. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*. 46–61.

Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. 2007. Semantics of static pointcuts in aspectJ. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. 11–23.

Anya Helene Bagge and Karl Trygve Kalleberg. 2006. DSAL= library+ notation: Program transformation for domain-specific aspect languages. In *Proceedings of the Domain-Specific Aspect Languages Workshop*.

Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. 2006. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundam. Inform.* 69, 1-2 (2006), 123–178.

Thierry Coquand and Christine Paulin-Mohring. 1990. Inductively defined types. In *COLOG-88*, Per Martin-Löf and Grigori Mints (Eds.). LNCS, Vol. 417. Springer, 50–66.

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. Abstract Interpretation Frameworks. *J. Log. Comput.* 2, 4 (1992), 511–547.

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2006. Combination of Abstractions in the ASTRÉE Static Analyzer. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*. 272–300.

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.

David Charles Darais. 2017. *Mechanizing Abstract Interpretation.* Ph.D. Dissertation. University of Maryland, College Park, MD, USA.

Maartje de Jonge and Eelco Visser. 2012. A Language Generic Solution for Name Binding Preservation in Refactorings. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA '12)*. ACM, New York, NY, USA, Article 2, 8 pages.

Eelco Dolstra and Eelco Visser. 2002. Building Interpreters with Rewriting Strategies. *Electronic Notes in Theoretical Computer Science* 65, 3 (2002), 57–76.

Giorgios Rob Economopoulos and Bernd Fischer. 2011. Higher-order transformations with nested concrete syntax. In *Language Descriptions, Tools and Applications, LDTA 2011, Saarbrücken, Germany, March 26-27, 2011. Proceeding*, Claus Brabrand and Eric Van Wyk (Eds.). ACM, 4.

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 391–406.

Neil Ghani, Patricia Johann, Fredrik Nordvall Forsberg, Federico Orsanigo, and Tim Revell. 2015. Bifibrational Functorial Semantics of Parametric Polymorphism. *Electr. Notes Theor. Comput. Sci.* 319 (2015), 165–181.

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.

N Jones and Flemming Nielson. 1994. Abstract interpretation: a semantics-based tool for program analysis. *Handbook of logic in computer science* 4 (1994), 527–636.

Sven Keidel and Sebastian Erdweg. 2017. Toward Abstract Interpretation of Program Transformations. In *Proc. Meta*. ACM, 1–5.

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.

Saunders Mac Lane. 1978. *Categories for the Working Mathematician*. Springer New York.

Jan Midtgaard. 2012. Control-flow analysis of functional programs. *ACM Comput. Surv.* 44, 3 (2012), 10:1–10:33.

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92.

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

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.

Gordon D Plotkin. 1980. Lambda-definability in the full type hierarchy. *To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* (1980), 363–373.

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*. 513–523.

John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation* 11, 4 (1998), 363–397.

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.

Elmer van Chastelet, Eelco Visser, and Craig Anslow. 2015. Conf.Researchr.Org: towards a domain-specific content management system for managing large conference websites. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 50–51.

Eelco Visser. 2007. WebDSL: A Case Study in Domain-Specific Language Engineering. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007 (Lecture Notes in Computer Science)*, Ralf Lämmel, Joost Visser, and Jo ao Saraiva (Eds.), Vol. 5235. Springer, Braga, Portugal, 291–373.

Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*. 13–26.

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*. ACM, 347–359.

Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*. 24–52.