Type and Control-Flow Analysis for Scheme in Sturdy

Tobias Hombücher

March 22, 2020

Johannes-Gutenberg University Mainz

Programming Languages Research Group
Institute of Computer Science

Bachelor Thesis

Type and Control-Flow Analysis for Scheme in Sturdy

Tobias Hombücher

1. Reviewer Sebastian Erdweg, Univ.-Prof. Dr.
Institute of Computer Science
Johannes-Gutenberg University Mainz

2. Reviewer André Brinkmann, Univ.-Prof. Dr.
Institute of Computer Science
Johannes-Gutenberg University Mainz

Supervisors Sven Keidel, M.Sc.

March 22, 2020

Tobias Hombücher
Type and Control-Flow Analysis for Scheme in Sturdy
March 22, 2020
Reviewers: Sebastian Erdweg, Univ.-Prof. Dr. and André Brinkmann, Univ.-Prof. Dr.
Supervisors: Sven Keidel, M.Sc.

Johannes-Gutenberg University Mainz
Institute of Computer Science
Programming Languages Research Group
Staudingerweg 9
55128 Mainz

Abstract

Dynamically typed programming languages are prone to run-time type errors. Static type checkers try to detect possible type errors before execution. Scheme is a dynamically typed programming language and allows functions to return values of multiples types, as well as higher-order functions. These features prevent common static type checkers to perform meaningful type checking on Scheme programs. Our implemented static analysis utilizes the approach of abstract interpretation to perform a meaningful type analysis on Scheme programs, nonetheless. Instead of declaring fixed types to functions of a program, our type analysis analyzes the application of each function individually. Additionally our implementation includes a control-flow analysis that allows the analysis of programs with complex control-flow caused by e.g. higher-order functions. We tested the implementation on 10 Scheme benchmark programs, being able prove 8 of them type-safe. The here presented type and control-flow analysis is capable to soundly analyze programs with complex control-flow, however this comes at the cost of a high complexity and a reduction in precision.

Abstract (German)

Dynamisch getypte Programmiersprachen sind anfällig für Typfehler zur Laufzeit. Statische Typchecker versuchen Typfehler vor der Ausführung eines Programmes zu entdecken. Scheme ist eine dynamisch getypte Programmiersprache und erlaubt Funktionen Werte mehrerer Typen zurückzugeben, sowie die Nutzung von Funktionen höherer Ordnung. Diese Eigenschaften verhindern es statischen Typecheckern aussagekräftig Scheme Programme zu typchecken. Unsere implementierte statische Analyse nutzt die Methode der abstrakten Interpretation, um dennoch eine sinnhafte Typanalyse für Scheme Programme durchführen zu können. Anstatt Funktionen feste Typen zu erteilen, wird jede Anwendung einer jeden Funktion einzeln analysiert. Zusätzlich inkludiert unsere Analyse eine Kontrollflussanalyse, die die Analyse von Programmen mit einem komplexen Kontrollfluss, welcher bspw. von Funktionen höhere Ordnung verursacht wird, ermöglicht. Wir haben unsere Analyse für 10 Scheme Benchmark Programme getestet und waren imstande für 8 Typsicherheit zu garantieren. Die hier implementierte Typ- und Kontrollflussanalyse kann Programme mit einem komplexen Kontrollfluss korrekt analysieren, dies ist jedoch verbunden mit einer hohen Komplexität und einer Verringerung der Präzision.

Contents

1 Introduction

Software development is a complex process. Mistake are inevitable. Most published software has bugs and it has been demonstrated many times how costly they can become. There are several techniques to reduce the amount of bugs in software, the most common of which is testing. Quality control by testing has two main drawbacks. First, testing is often performed late in the production cycle, when costs of removing bugs have risen a lot since the beginning. Second, "*testing can only prove the existence of bugs and not their absence*" [1], which is the reason why test coverage should be as high as possible. However, it is impossible to cover every execution path of a non-trivial program. Static analyses provide a way of doing just that. In contrast to their dynamic counterpart, that prevent errors occurring at run-time, static analyses analyze programs before their execution. Therefore, a large benefit of static analyses is that they analyze source code and can be applied incrementally, allowing application in every state of program development, being useful often times a lot earlier in the process than testing. Another advantage is that in order to perform a sound analysis, static analyses conclude over the execution of a program in any possible environment, guaranteeing complete coverage.

This work presents the implementation of a static type and control-flow analysis for Scheme. Scheme is a dynamically typed programming language, not requiring explicit type annotations in its source code. Dynamically typed programming languages are in danger of producing type errors at run-time. Our analysis prevents type errors from occurring, by means of abstract static analysis. It mainly differs from common static type checkers by instead of analyzing functions once and declaring it a type, it analyzes every function in its actual application context. Hence, sound static type checking often performs a very conservative analysis. Consider the following Scheme program:

```
(define (check x)
  (if (list? x) 0
      (if (number? x) 1
          (if (not (list? x)) 0
              (if (integer? x) #t
                  (if (boolean? x) '(0) #f))))))
```

A common static type checker would return three different types, integer, boolean and list, as the return type of the function `check`. This is a sound analysis result,

however not particular precise. Anytime this function is used in a context in which an integer is expected, e.g. integer addition, a type checker would report a type error. The approach developed in this work is able to produce significantly more precise results. Each application of `check` is analyzed individually. Instead of assigning a type to `check` by analyzing its body, our analysis considers every application of `check` anew. This means, instead of declaring every application of `check` in a context where integers are expected a type error, our analysis evaluates `check` for the exact arguments given in that context and only then decides whether type-safety can be guaranteed. In consequence, our analysis is able to detect what a type checker cannot. It realizes the last two if-statement can never be executed. By using the approach of abstract static analysis our analysis considers every possible execution path for a given program in which those non-reachable statements are not included. Therefore, our analysis detects that `check`'s only return type is integer, producing a much more precise result than a standard static type checker.

We can see that by considering each application of a function individually, more precise results can be achieved for our type analysis. However, statically gaining information about the arguments each function is parsed is not trivial. Analyzing the so called data-flow is particularly hard for programming languages such as Scheme, which support higher-order functions. A classic example for such a case, which is also discussed in *Principles of Program Analysis* by Nielson et al. [6], is the following Scheme program:

```
(let ((f (lambda (x) (x 1)))
      (g (lambda (y) (+ y 2)))
      (h (lambda (z) (+ z 3))))
   (+ (f g) (f h)))
```

`f` is bound to a lambda-expression that applies a function parsed to it as argument. Statically analyzing the data-flow is not easily possible because the function that is parsed to the lambda-expression is only certain at run-time. Control-flow analysis are able to analyze the data-flow nonetheless, even though precision has to be sacrificed. It computes for each subexpression of a program a set of functions that they can evaluate to. In this case `f` can be applied either to `g` or `h`, which in turn means the argument `x` of the lambda-expression `f` can evaluate to either `(lambda (y) (+ y 2))` or `(lambda (z) (+ z 3))`. To guarantee a sound result both possibilities have to be accounted for when analyzing the so called control-flow of this program.

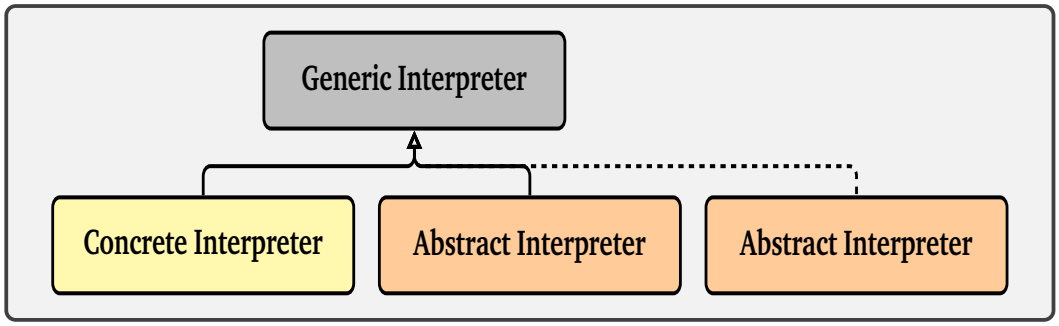

Figure 1.1: Overview Interpreters

We have implemented our analysis in the *Sturdy* [5] framework. The main idea of the framework is to capture similarities between the concrete and abstract interpretation of programs. Similarities between them are explicitly implemented by a generic interpreter that solely operates on interfaces that concrete and abstract interpreters implement. The purpose of our concrete interpreter is to implement the concrete evaluation of a program by instantiating the interfaces defined in our generic interpreter. The purpose of our abstract interpreter is to perform a type and a control-flow analysis by means of abstract interpretation by instantiating the same generic instances as the concrete interpreter [Figure 1.1].

Sturdy also provides already implemented features that can be reused. Several benefits can be gained by this approach to implementing abstract analyses. Proving abstract interpreters sound is a difficult task prone to errors. Soundness proofs can become easier, when similarities between concrete and abstract interpreters are explicitly depicted in the generic interpreter. The ability to reuse structure that has already been proven sound, allows for *compositional soundness proofs* [5], which can further reduce the prove effort. Another benefit of the framework is that adding an abstract interpreter for a language that already has a working concrete, generic and abstract interpreter can be done with much less effort, than implementing an abstract interpreter from scratch.

2 A Generic and Concrete Interpreter for Scheme

In this section we will discuss the generic interpreter, which captures similarities between concrete and abstract interpreters and therefore is a core part of any analysis implemented in *Sturdy* [5]. To better demonstrate the actual functionality of this generic interpreter, we present the concrete interpreter along the way. The generic interpreter operates on interfaces that are implemented by the concrete interpreter as well as the abstract interpreter. The most relevant interfaces are those that provide the data structures and operations for stores, environments, addresses and values. As depicted in Figure 2.1, the concrete and abstract interpreter implement these interfaces with their own instances respectively.

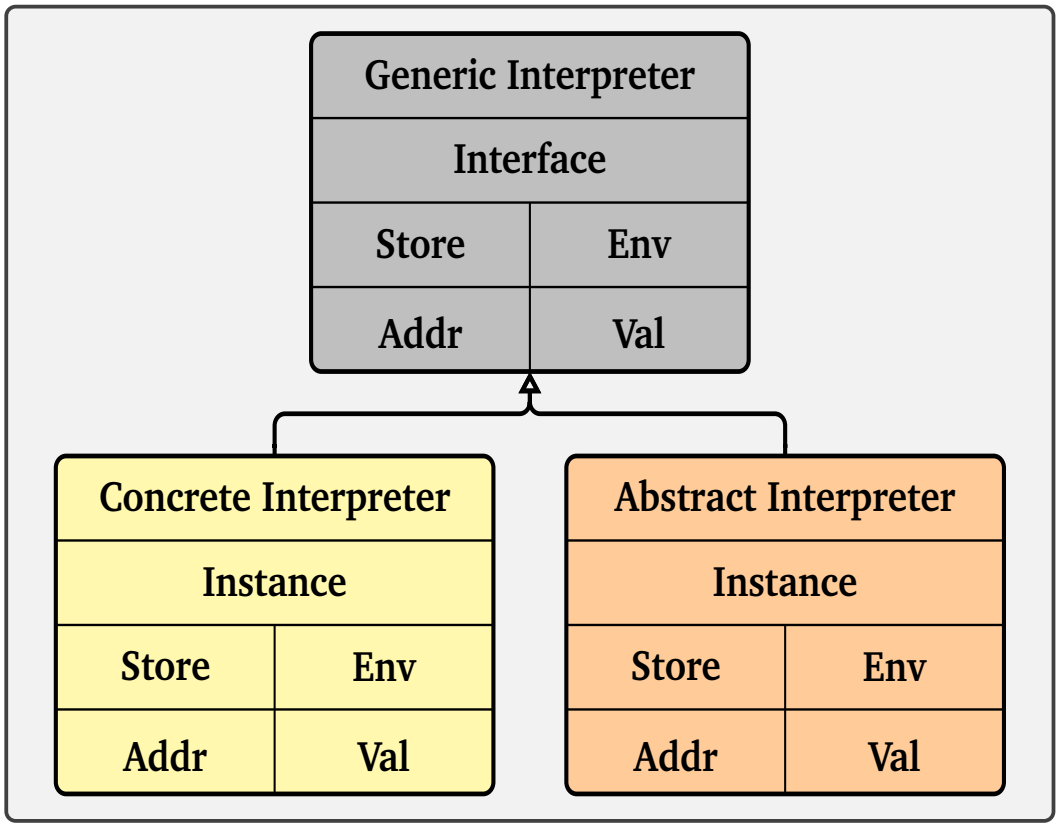

Figure 2.1: Detailed Overview Interpreters

Sturdy uses *arrows* [3] to describe effectful computations. They are also used in the generic interpreter, e.g. `lit :: c Literal v` describes a computation that

consumes `Literal` as input and produces a parametric type `v` as output. The concrete or abstract interpreters will later instantiate `v`, representing their value domain respectively. In order to express similarities in their domains nonetheless, `v` has to be parametric.

Generic Implementations

```
data Expr = Lit Literal Label | ...
data Literal = Number Int | Float Double | Bool Bool | ...
eval = proc e -> case e of
  Lit x l -> lit -< x
```

Generic Interfaces

```
lit :: c Literal v
```

Concrete Instances

```
lit = proc x -> case x of
  Number n -> returnA -< IntVal    n
  Float  n -> returnA -< FloatVal  n
  Ratio  n -> returnA -< RatioVal  n
  Bool   n -> returnA -< BoolVal   n
  Char   n -> returnA -< CharVal   n
  String n -> returnA -< StringVal n
  Quote  n -> returnA -< evalQuote n
  _        -> fail    -< "(lit): unexpected literal"
```

Listing 2.1: Generic and Concrete Literal Expressions

To demonstrate the architecture and the interactions between the generic and concrete interpreter we start by discussing literals, which present the most basic syntactic elements of Scheme [Listing 2.1]. The generic interpreter first encounters the literal in form of an expression. A `Lit`-expression contains a `Literal` type, here represented by `x` that corresponds to any of Scheme's basic types, excluding lists and procedures. Every expression also contains an unique label, here represented by `l`. The interface specified by the generic interpreter defines a simple arrow computation that expects an element of type `Literal` as its input and returns an element of a parametric type `v` as its output. The concrete interpreter determines `v` to represent the concrete value domain. When the generic interpreter encounters a `Lit`-expression, it uses the `lit`-interface to evaluate the expression. As literals are already in their most basic form, no further evaluation has to be performed and `x` can be parsed directly to `lit` as input. Finally, the concrete interpreter

determines the actual evaluation by instantiating the predefined interfaces of the generic interpreter.

The `lit`-interface specifies an arrow computation as its type. Arrows can be constructed by using `proc`, which is similar to a lambda, but instead of creating a function it creates an arrow. Therefore, `(proc x -> case x of ...)` creates an arrow that performs a pattern match on the input `x`, which in our case will be an element of type `Literal`. `returnA` and `fail` are arrow computations themselves. `returnA` is the arrow representation of the identity function and `fail` transforms the string it is given as input into a value that matches the respective specification for `v`. Much like monads it is necessary to end a `proc`-block with a computation and not a binding. Putting it together the `lit`-interface is implemented by a simple pattern match on the input of the computation. Every `Literal` is evaluated to their corresponding representation in the concrete value domain. Any unexpected input will result in a failure.

Generic Implementations

```
data Expr = If Expr Expr Expr Label | ...
eval :: c [Expr] v -> c Expr v
eval run = proc e -> case e of
  If e1 e2 e3 l -> do
    v1 <- run -< [e1]
    if_ run run -< (v1, ([e2], [e3]))
run_ :: c [Expr] v
run_ = fix $ \run -> proc es -> case es of
 ...
```

Generic Interfaces

```
if_ :: c x z -> c y z -> c (v, (x, y)) z
```

Concrete Instances

```
if_ run1 run2 = proc (v1, (e2, e3)) -> case v1 of
  BoolVal False -> run2 -< e3
  _ -> run1 -< e2
```

Listing 2.2: Generic and Concrete If Expressions

Another simple but core part of most programming languages are if-expressions [Listing 2.2]. Expressions of type `If` have three sub-expressions and are, as every expression, uniquely identified by a label `l`. `If`-expressions are evaluated by the `eval` function of the generic interpreter. `eval` is a computation that takes an arrow

computation `c [Expr] v` as argument, an expression `Expr` as input and returns an element of the parametric type `v`. Other than `eval`, the generic interpreter contains another function `run_` that takes part in the evaluation of expressions. `run_` serves as a wrapper and processes the input Scheme program, which has been preprocessed to a list of expressions. Single expressions are then parsed to `eval` for further processing. `run_` evaluates each expression in order, beginning with the first and returning the result of the last expression parsed to `eval`. The evaluation of e.g. `e1` from the `If`-expression takes a seemingly indirect path to evaluation. Instead of directly parsing `e1` to `eval`, it is wrapped to a list and parsed to a `run_` arrow computation that is specially parsed to `eval` as argument. This indirection is necessary, because the fix-point algorithms that determine over termination and non-termination are parametric in their type and need consistent control over every expression that might diverge and create non-termination. To guarantee consistency the fix-point is only calculated over `run_`, with the consequence that `eval` is parsed the arrow computation `run` of type `run_` over which the fix-point is calculated and only ever uses it to evaluate further expressions.

The `if_`-interface specified in the generic interpreter takes two arrow computations as arguments `c x z` and `c y z` and expects `v,(x,y)` as input. The generic interpreter first evaluates `e1` to a value. This is the conditional that decides whether the if- or else-branch is executed. Hence, `e2` and `e3` represent the if- and else-branches. For the moment they are not evaluated, instead two `run`-objects are parsed to `if_` as arguments that will later evaluate `e2` or `e3` in the interpreters on demand. The concrete `if_`-instance decides how evaluation is to be continued. *Scheme's language standard* [7] determines that every value that is not explicitly `#f` is considered `#t` and causes the if-branch to be evaluated. Therefore, only if the conditional matches `BoolVal False` the else-branch is evaluated by providing `run2` with the input `e3`. In any other cases the if-branch is evaluated by providing `run1` with the input `e2`. Further evaluation is now again performed by the generic interpreter.

A language feature that is more exclusive to Scheme than if-statements, are let-expressions. Scheme's `let`-expressions introduce bindings to a local scope in which their expression-body is evaluated. `let` can have multiple bindings each consisting of a variable as well as an expression. Aside from `let`, Scheme supports `let*` and `letrec`, which differ in the order in which bindings are evaluated and added to the local environment. Bindings in `let` are practically evaluated at the same time, that is no information provided by any other bindings is considered when evaluating an individual binding. Bindings in `let*` are evaluated one after the other, that is information from previous bindings can be considered when evaluating a binding occurring after them, making expressions such as `(let* ((x 2) (y x)) y)` valid. Bindings in `letrec` have the same order of evaluation as bindings in `let*`, with the addition that for each binding a location in the environment is allocated

before the evaluation of the bindings begins. This enables mutual recursion of bindings, because closures in bindings can capture an environment in which all the other bindings, including itself are already present. `letrec` allows e.g. following expression: `(letrec ((y (lambda () x)) (x (lambda () y))) y)`. However, as the order in which bindings are evaluated is still relevant, expressions such as: `(letrec ((x y) (y 2)) x)` are not allowed. Fortunately we only need to natively implement `let`- and `letrec`-expressions. `let*`-expressions can be desugared by creating for each binding in the expression an individual `let`-expression and nesting the created `let`-expression in the order of the bindings' occurrence. The deepest nested `let`-expression will carry the body of the original `let*`-expression.

Here we will discuss the implemented representations of `let`- and `letrec`-expressions, beginning with `let` [Listing 2.3]. `Let`-expressions consist of `bnds`, which is a list of tuples consisting of variable names and their associated expression, and `body`, which is a list of expressions. The evaluation of `Let`-expressions relies on three different interfaces `alloc`, `write` and `extend`. Before `body` can be evaluated all bindings have to be added to a local environment. The function `evalBindings` evaluates each expression within `bnds` to a value, allocates an address, using the `alloc`-interface and adds the binding of type `(Addr,Val)` to the store, using the `write`-interface. `evalBindings` returns a list of tuples `[(Var, Addr)]`, consisting of variable names and their associated addresses. `Env.extend'` is responsible for adding these bindings to the environment, it is a function that simply wraps around the `extend`-interface to allow the extension of a list of bindings, instead of only single ones. As the environment is only locally scoped, `extend` is parsed a computation that is to be computed under the extended environment, as well as its corresponding input. In this case `run` is added as computation and `body` as the computation's input.

The concrete domain represents addresses as simple integers. `alloc` uses Sturdy's *State* instance [4] to keep a counter `nextAddr`, which starts at 0 and is increased by 1 whenever an address is allocated. The concrete environment is realized as a mapping from variable names to integers `type Env = Map Var Int` and the concrete store as a mapping from integers to values `type Store = Map Int Val`. The environment and store are threaded through computations and kept consistent using the *Reader*, as well as again the *State* instance provided by the Sturdy framework. Therefore, updating the store with a new binding becomes quite simple. Using `State.get` a current version of the store can be retrieved, and using `State.put` a store, which has been updated can be set as the new current store. Extending the environment follows the same example, `Reader.ask` retrieves a current environment, which can then be extended with a binding. `Reader.local` performs the computation that is parsed to it as argument, here `run`, under the extended environment that has been parsed, using `x` as its input, which is `body` for our `Let`-expression.

Generic Implementations

```
data Expr = Let [(Text, Expr)] [Expr] Label | ...
eval :: c [Expr] v -> c Expr v
eval run = proc e -> case e of
  Let bnds body -> do
    vs <- evalBindings -< bnds
    Env.extend' run -< (vs,body)
  where
    evalBindings = proc bnds -> case bnds of
      [] -> returnA -< []
      (var,expr) : bnds' -> do
        val <- run -< [expr]
        addr <- alloc -< var
        write -< (addr,val)
        vs <- evalBindings -< bnds'
        returnA -< (var,addr) : vs
```

Generic Interfaces

```
alloc :: c Text addr
write :: c (addr,val) ()
extend :: c x y -> c (var,addr,x) y
```

Concrete Instances

```
alloc = proc _ -> do
  nextAddr <- get -< ()
  put -< nextAddr + 1
  returnA -< nextAddr
write = proc (addr, val) -> do
  store <- State.get -< ()
  State.put -< Map.insert addr val store
extend run = proc (var,addr,x) -> do
  env <- Reader.ask -< ()
  Reader.local run -< (Map.insert var addr env, x)
```

Listing 2.3: Generic and Concrete Let Expressions

`LetRec`-expressions are implemented very similarly to `Let`-expressions [Listing 2.4]. They use exactly the same interfaces and only differentiate in their generic implementation. During the evaluation of `LetRec`-expressions, the generic interpreter first allocates an address for every binding and matches them with its corresponding variable name, summarizing the bindings that have to be added to the environment

before the actual evaluation of the bindings in `envbnds`. `strbnds` summarizes the bindings that now have to be evaluated after the environment has been extended with `envbnds`. The evaluation happens in `evalBindings'` where each expression is evaluated to a value and then, paired with the respective address, added to the store. This is repeated for every expression in bnds. When no more bindings have to be evaluated `evalBindings'` finally evaluates `LetRec`'s body by parsing it as input to `run`

Generic Implementations

```
data Expr = LetRec [(Text, Expr)] [Expr] Label | ...
eval :: c [Expr] v -> c Expr v
eval run = proc e -> case e of
  LetRec bnds body l -> do
    addrs <- map alloc -< [(var)|(var,_) <- bnds]
    let envbnds = [(var,addr)|((var,_),addr) <- zip bnds addrs]
    let strbnds = [(addr,expr)|((_,expr),addr) <- zip bnds addrs]
    Env.extend' evalBindings' -< (envbnds,(storebnds,body))
where
  evalBindings' = proc (bnds,body) -> case bnds of
    [] -> run -< body
    (addr,expr) : bnds' -> do
      val <- run -< [expr]
      write -< (addr,val)
      evalBindings' -< (bnds',body)
```

Generic Interfaces

```
alloc :: c Text addr
write :: c (addr,val) ()
extend :: c x y -> c (var,addr,x) y
```

Listing 2.4: Generic LetRec Expressions

As already mentioned when discussing literals, closures and lists are not represented by `Literal`-expressions, but are implemented individually. Nonetheless, both are part of the concrete and abstract value domain. We will continue by discussing first closures and then lists.

Closures occur when lambda-expressions are evaluated. Our representation of lambda-expressions consists of a list of variable names, a list of expressions and of course a label [Listing 2.5]. `Lam`-expressions are evaluated by the `eval` function of the generic interpreter, which uses the `closure`-interface. The `closure`-interface

is an arrow computation that expects an expression as argument and returns an element `cls`. `cls` is a parametric closure value that will be declared by the concrete and abstract interpreters to depict their respective representation of closure values. The concrete interpreter instantiates `cls` with `type Cls = Closure Expr Env`. The `closure`-interface is instantiated by an arrow computation that uses Sturdy's *Reader* instance to retrieve the environment that is current at the point of evaluation and captures it together with the `Lam`-expression, which originally was evaluated.

The application of lambdas is represented by the `App`-expression. It consists of an expression, which will be a `Lam`-expression, a list of expressions, which are the arguments and again a label. The generic interpreter evaluates `App`-expressions by first evaluating the `Lam`-expression, here depicted by `e1`, to a closure value, depicted by `fun`. Then the arguments are evaluated by parsing them one by one to the `run` arrow computation that is parsed to `eval` as argument. The closure is finally applied by the `apply`-interface. This interface takes an arrow computation `c (expr,args) v` as argument, `(cls,args)` as input and returns a parametric type `v`, which will be instantiated by the respective interpreters' value domain. The generic interpreter lastly passes the evaluated closure `fun` and arguments `args` to the `apply`-interface, as well as a function `applyClosure`.

The concrete interpreter instantiates the `apply`-interface by using Sturdy's *Reader* instance and its `local`-function to set the environment the closure captures as the current environment. In this environment the parsed arrow computation `applyClosure` is called with the expression of the closure, as well as the arguments. `applyClosure` checks that the expression indeed is a `Lam`-expression and that the amount of parsed arguments matches the amount required by the `Lam`-expression. If they match, addresses are allocated for all variables of the `Lam`-expression and all arguments bound to the corresponding variable are written to the store. Lastly the `extend'`-function is used again to add the list of bindings to the environment. An expressions `Apply body l`, which simply evaluates the body of the `Lam`-expression in the just constructed environment, is parsed to `extend'` to be the input of the also parsed `run` arrow computation. It is important to note that instead of just parsing the body itself, a special expression is created. This is necessary because the fix-point algorithms are using functions of this kind as a reference to recognise recursion, as this is the only point in any evaluation that recursion can occur.

Generic Implementations

```
data Expr = Lam [Text] [Expr] Label | App Expr [Expr] Label
          | Apply [Expr] Label | ...
eval :: c [Expr] v -> c Expr v
eval run = proc e -> case e of
  Lam xs es l -> closure -< Lam xs es l
  App e1 e2 l -> do
    fun <- run -< [e1]
    args <- map run -< chunksOf 1 e2
    apply applyClosure -< (fun, args)
  Apply es l -> run -< es
  where
    applyClosure = proc (e, args) -> case e of
      Lam xs body l ->
        if eqLength xs args
          then do
            addrs <- map alloc -< xs
            map write -< zip addrs args
            Env.extend' run -< (zip xs addrs, [Apply body l])
          else fail -< "Mismatching amount of arguments"
      _ -> fail -< "Found unexpected epxression in closure"
```

Generic Interfaces

```
closure :: c expr cls
apply :: c (expr,args) v -> c (cls,args) v
alloc :: c Text addr
write :: c (addr,val) ()
extend :: c x v -> c (var,addr,x) v
```

Concrete Instances

```
type Cls = Closure Expr Env
closure = proc expr -> do
  env <- Reader.ask -< ()
  returnA -< Closure expr env
apply applyClosure = proc (Closure expr env,args) ->
  Reader.local applyClosure -< (env,(expr,args))
```

Listing 2.5: Generic and Concrete Lambda and Application Expressions

Lists in Scheme are tail-recursive, that is they consist of a concrete head value and a tail value that itself is a list, or in case of a list of length one, the empty list. There

exist two expressions that are needed to represent this structure in our interpreters, `Nil` and `Cons` [Listing 2.6]. Every list that has to be evaluated is of the following form `(Cons x (Cons y (Cons z Nil)))`. The generic interpreter first evaluates the head element of `Cons`-expressions and then the tail element. Because the tail element is itself a list, this will cause a recursive evaluation until an expression `(Cons x Nil)` is reached. `Nil`-expressions are evaluated by the `nil_`-interface, which only needs a `Label` and returns a parametric type `v`. When the recursive call is finished the generic interpreter passes the evaluated head and tail values, along with the label of the expressions they originate from, to the `cons_`-interface.

Generic Implementations

```
data Expr = Nil Label | Cons Expr Expr Label | ...
eval :: c [Expr] v -> c Expr v
eval run = proc e -> case e of
  Nil l -> nil_ -< l
  Cons x xs l -> do
    v <- run -< [x]
    vs <- run -< [xs]
    cons_ -< ((v,label x),(vs, label xs))
```

Generic Interfaces

```
nil_ :: c Label v
cons_ :: c ((v, Label), (v, Label)) v
```

Concrete Instances

```
data Val = ListVal Addr Addr | EmptyList | ...
  nil_ = proc _ ->
    returnA -< EmptyList
  cons_ = proc ((v1,_),(v2,_)) -> do
    a1 <- alloc -< ""
    a2 <- alloc -< ""
    write -< (a1,v1)
    write -< (a2,v2)
    returnA -< ListVal a1 a2
```

Listing 2.6: Generic and Concrete List Expressions

The concrete implementation of the `nil_`-interface simply returns a value `EmptyList` that represents the empty list. The concrete implementation of the `cons_`-interface is perhaps somewhat unexpected. Instead of directly representing the head and tale values they are represented by addresses, which are specifically allocated for each

value and written to the store. This is not entirely necessary for the concrete evaluation, however it reduces the effort when implementing the abstract representation of lists. The here discussed representation of lists can lead to very large stores, which is the reason why any figure and example in this work uses Scheme's high-level representation for lists.

3 An Abstract Interpreter for Scheme

In this section we discuss the abstract interpreter, which implements the generic interpreter to perform a type and control-flow analysis. We will begin by discussing its abstract domain. Our abstract domain has to fulfill following properties, it has to abstract the concrete domain to a finite domain, and it has to be sufficiently precise to allow conclusion over the type-safety of a program in a meaningful way.

An important property of abstract interpreters is the guarantee of termination even for non-terminating programs. This can be ensured by keeping the abstract domain finite and is the reason most concrete values are abstracted from and lose information in the abstract value domain. Consider the abstract implementation of the evaluation of literals in Listing 3.1.

Generic Interfaces

```
lit :: c Literal v
```

Abstract Instances

```
lit = proc x -> returnA -< case x of
  Number _             -> NumVal    IntVal
  Float  _             -> NumVal    FloatVal
  Ratio  _             -> Bottom
  Bool   True          -> BoolVal   B.True
  Bool   False         -> BoolVal   B.False
  Char   _             -> StringVal
  String _             -> StringVal
  Quote  (Symbol sym) -> QuoteVal  (singleton sym)
  _                    -> Bottom
```

Listing 3.1: Abstract Literal Expressions

Just as the concrete interpreter, the abstract interpreter instantiates the `lit`-interface defined by the generic interpreter. However, the domain that literals are evaluated to has changed. The concrete values of `Number` literals are not regarded anymore. Instead they are abstracted to a more general type `NumVal IntVal`. The same can be noted for all other literals. `Float` is evaluated to `NumVal FloatVal`. `Bool` and

`Quote` are the only literals that keep their values in the abstract value domain. It is important to sustain as much precision as possible, even when it is not relevant to know e.g. whether a value is true or false to determine that it is a boolean, we will see that the precision of e.g. if-statements heavily relies on the precision of its conditional, which most of time is represented by a boolean.

Ensuring a finite abstract domain, does not only restrict itself to abstract values. The abstract environment, store, and set of addresses have to be finite as well. Addresses in the concrete interpreter are represented by an infinite set of integers. Every time a variable needs to be added to the store a fresh address is allocated. This can no longer be done when ensuring a finite domain.

Abstract address allocation is implemented by instantiating the `alloc`-interface of the generic interpreter. Addresses that are allocated for variables consist of a tuple of their variable name and their context [Listing 3.2]. The context is a string of the history of expression-labels that led to the allocation of the variable, with the label of the latest expression that was evaluated being the first element in the string. The length of the call-string is determined by the grade k of the control-flow analysis. That is for $k = 0$ the length of the call-string is 0 and does not hold any information over the call context, for $k = 1$ it holds information for the most recent call context. The context can be retrieved by using the *Ctx* [4] instance, which is predefined in Sturdy's library.

Generic Interfaces

```
alloc :: c Text addr
```

Abstract Instances

```
data Addr = VarA (Text,Ctx) | ...
alloc = proc var -> do
  ctx <- Ctx.askContext @Ctx -< ()
  returnA -< VarA (var,ctx)
```

Listing 3.2: Abstract Allocation Strategy

It can be easily reasoned why this change to address allocation guarantees the amount of possible addresses to be finite. The amount of unique variable names, as well as the amount of expressions are finite in any program. The set of possible contexts is finite as well, as the contexts are bounded by k and their elements are the labels of expressions of which there can only be as many as there are expressions. Therefore, the set of tuples consisting of variable names and contexts is finite.

Of course limiting the set of addresses, enables duplicate allocation of addresses. Hence, it is possible to allocate the same address for two different variables. As an abstract interpreter is only sound if it over-approximates the concrete interpreter, bindings with the same address cannot simply overwrite each other in the store. This problem is resolved by introducing a monotone stone, which can only grow and, instead of loosing information, only looses precision [9]. Because the store never loses information, if a value is written to an address that has already been allocated, the values are joined and stored as their joined value. This sacrifice is necessary, as the allocation strategy is the most crucial feature to allow termination of non-terminating programs.

```
instance Complete Val where
  Bottom        ⊔ x             = x
  x             ⊔ Bottom        = x
  StringVal     ⊔ StringVal     = StringVal
  NumVal     n1 ⊔ NumVal     n2 = NumVal     (n1 ⊔ n2)
  ...
  _             ⊔ _             = Top
instance Complete Number where
  IntVal   ⊔ IntVal   = IntVal
  FloatVal ⊔ FloatVal = FloatVal
  _        ⊔ _        = NumTop
```

Listing 3.3: Widening Operator

Joining values is performed by a widening operator ⊔ that functions as a least-upper-bound for the defined abstract values and depicts our lattice [Listing 3.3]. `Bottom` is the lowest element in our lattice and every element joined with it results in itself. Another trivial case are values of type `StringVal` that, because strings are naively abstracted, joined with itself result in `StringVal` without any real loss of precision, as there was none to begin with. Values of all other types, joined with a value of the same type respectively, result in themselves with their sub-types joined. Values of type `NumVal IntVal` as well as `NumVal FloatVal`, each joined with the same value, will result in themselves and do not loose any precision. Joining any other combination of elements of type `NumVal` results in `NumVal NumTop`, e.g. `NumVal IntVal` ⊔ `NumVal FloatVal` results in `NumVal NumTop`.

Values of type `Top` or values of two different types are joined to `Top`. `Top` represents any possible value of the lattice. Joining values to `Top` introduces a great amount of imprecision, as most operations that are parsed `Top` as an argument will result

in `Top` as well. Therefore, once a `Top` value has been introduced, the analysis of a given Scheme program will almost always result in `Top`.

Consider the following concrete and abstract evaluation of a trivial Scheme program, which demonstrates the use of the monotone store [Example 3.1].

```
(define x 2)
(set! x 2.3)
x
```

Concrete Trace

Environment	Store
`[]`	`[]`
`(define x 2)`	
`[x ↦ #0]`	`[#0 ↦ IntVal 2]`
`(set! x 2.3)`	
`[x ↦ #0]`	`[#0 ↦ FloatVal 2.3]`
`x`	
`FloatVal 2.3`	

Abstract Trace

`[]`	`[]`
`(define x 2)`	
`[x ↦ (x,[])]`	`[(x,[]) ↦ NumVal IntVal]`
`(set! x 2.3)`	
`[x ↦ (x,[])]`	`[(x,[]) ↦ NumVal NumTop]`
`x`	
`NumVal NumTop`	

Example 3.1: Simple Concrete and Abstract Evaluation

The concrete evaluation of the program returns `FloatVal 2.3`, the abstract evaluation however returns `NumVal NumTop`. The abstract intepreter first evaluates `(define x 2)`. For this the interpreter evaluates `2` to `NumVal IntVal`, allocates an address for `x` and writes `NumVal IntVal` to that address. Then it evaluates `(set! x 2.3)` by evaluating `2.3` to `NumVal FloatVal`, looking up the address for `x` in the current environment and its corresponding value in the store and writing the joined value of the looked up value and `NumVal FloatVal` to the address of `x`

in the store. The monotone store obviously causes a significant loss of precision, however it is, along with the allocation strategy the most important features to not only guarantee termination of non-terminating programs, but also to guarantee a faster computation of the fixed-point of a program.

If we remember from the section before closures and lists are treated specially, this is also the case for their joining. When joining closures it has to be guaranteed that the body of each closure can be evaluated in its respective environment. If a joined closure value is evaluated, each body is evaluated to a value that has to be joined. Fortunately Sturdy includes a *Closure* instance that provides a data structure, which performs all necessary operations, including the implementation of the associated `closure`- and `apply`-interfaces.

Generic Interfaces

```
nil_ :: c Label v
cons_ :: c ((v, Label), (v, Label)) v
```

Abstract Instances

```
data Addr = LabelA (Label,Ctx) | ...
data Val = ListVal List | ...
data List = Nil | Cons (Set Addr) (Set Addr)
          | ConsNil (Set Addr) (Set Addr)
nil_ = proc _ -> returnA -< ListVal Nil
cons_ = proc ((v1,l1),(v2,l2)) -> do
  a1 <- allocLabel -< l1
  a2 <- allocLabel -< l2
  write -< (a1,v1)
  write -< (a2,v2)
  returnA -< ListVal (Cons (singleton a1) (singleton a2))
allocLabel = proc l -> do
  ctx <- Ctx.askContext @Ctx -< ()
  returnA -< LabelA (l,ctx)
```

Listing 3.4: Abstract List Expressions

Before discussing the joining of lists we have to discuss their abstract representation [Listing 3.4]. We will start by explaining why it is not possible to use a more intuitive representation such as e.g. `data List = Nil | Cons Val Val | ...`. When adding this kind of value to out lattice, we introduce the possibility of creating an infinite lattice, by constructing infinite large lists. Hence, we cannot allow any element of type `Val` to contain elements of type `Val`. We abstract lists by introducing

the indirection of addresses. For each head and tail value we allocate addresses and write them to the store. The abstract interpreter uses a special type of address, which consists of a label and the context, instead of a variable name and the context. This is necessary since lists do not have access to variable names. Nonetheless, these addresses are a finite set, because labels as well as contexts are finite sets as well. The abstract interpreters' implementation of the `nil_`- and `cons_`-interface is very similar to the concrete implementation, mainly differing in the way addresses are allocated. Another smaller difference is that abstract lists do not have a single address as their head and tail value, but a set. This allows us to remain a higher precision when joining two lists.

```
instance Complete List where
  Nil            ⊔ Nil            = Nil
  Cons    x1 x2 ⊔ Cons    y1 y2 = Cons    (x1 ⊔ y1) (x2 ⊔ y2)
  ConsNil x1 x2 ⊔ ConsNil y1 y2 = ConsNil (x1 ⊔ y1) (x2 ⊔ y2)

  Cons    x1 x2 ⊔ Nil            = ConsNil x1 x2
  Nil            ⊔ Cons    x1 x2 = ConsNil x1 x2
  ConsNil x1 x2 ⊔ Cons    y1 y2 = ConsNil (x1 ⊔ y1) (x2 ⊔ y2)
  Cons    x1 x2 ⊔ ConsNil y1 y2 = ConsNil (x1 ⊔ y1) (x2 ⊔ y2)
  Nil            ⊔ ConsNil y1 y2 = ConsNil y1 y2
  ConsNil y1 y2 ⊔ Nil            = ConsNil y1 y2
```

Listing 3.5: Widening Operator Lists

Coming back to the widening operator and its application to lists [Listing 3.5]. `Nil`, which represents the empty list, `Cons` which represents common lists and `ConsNil`, which represents either `Nil` or `Cons`, joined with an element of the same type respectively, result in themselves with their sub-types joined. For `Cons` and `ConsNil` this means that the head, as well as the tail set of addresses are joined. All other possible combinations result in `ConsNil` with their sub-types joined if there are any that is.

With the changed representation of addresses and the adjusted allocation strategy our abstract domain is finite. So how do we ensure the termination of non-terminating programs? Sturdy provides several fix-point algorithms that determine a point in the evaluation of a program after which it is not going to change any further. All operate in a slightly different way, however they all terminate, whenever the same expression is evaluated twice under the same context, which is represented by the abstract environment and store. Consider the following concrete and abstract evaluation traces of a non-terminating recursive Scheme program [Example 3.2].

```
(define (rec x) (rec 1))
(rec 1)
```

Concrete Trace

Environment	Store
`[]`	`[]`
`(define (rec x) (rec x))`	
`[rec ↦ #0]`	`[#0 ↦ {λ(x)(rec x), [rec ↦ #0]}]`
`(rec 1)`	
`[rec ↦ #0,`	`[#0 ↦ {λ(x)(rec x), [rec ↦ #0]},`
`x ↦ #1]`	`#1 ↦ 1]`
`(rec 1)`	
`[rec ↦ #0,`	`[#0 ↦ {λ(x)(rec x), [rec ↦ #0]},`
`x ↦ #2]`	`#1 ↦ 1, #2 ↦ 1]`
`(rec 1)`	
`...`	

Abstract Trace

`[]`	`[]`
`(define (rec x) (rec x))`	
`[rec ↦ (rec, [])]`	`[(rec, []) ↦ {λ(x)(rec x), [rec ↦ (rec, [])]}]`
`(rec 1)`	
`[rec ↦ (rec, []),`	`[(rec, []) ↦ {λ(x)(rec x), [rec ↦ (rec, [])]},`
`x ↦ (x, [])]`	`(x, []) ↦ NumVal IntVal]`
`(rec 1)`	
`[rec ↦ (rec, []),`	`[(rec, []) ↦ {λ(x)(rec x), [rec ↦ (rec, [])]},`
`x ↦ (x, [])]`	`(x, []) ↦ NumVal IntVal]`
`(rec 1)`	
`NonTerminating`	

Example 3.2: Concrete and Abstract Evaluation of Non-terminating Recursion

The concrete evaluation of this program will of course never terminate. As already discussed, any time a variable needs to be allocated a new address is provided. The concrete evaluation trace shows that `(rec 1)` will never be evaluated under the same environment and store, because a new binding is added for every recursive call. However, when considering the abstract evaluation trace, `(rec 1)` is evaluated under

the same environment and store, in which case the fix-point algorithm terminates evaluation and sets the result to `NonTerminating`.

In general the purpose of fix-point algorithms is not to terminate non-terminating programs, but to calculate the least-fixed of a program. This is not only necessary for non-terminating programs, but in particular for programs with loops and recursions. Abstract interpretation aims to over-approximate concrete evaluation of a program. Before discussing an example we have to introduce the abstract evaluation of if-expressions.

Generic Interfaces

```
if_ :: c x z -> c y z -> c (v, (x, y)) z
```

Abstract Instances

```
if_ run1 run2 = proc (v,(e1,e2)) -> case v of
  BoolVal B.False ->  run2 -< e2
  BoolVal B.Top   -> (run1 -< e1) <⊔> (run2 -< e2)
  Top             -> (run1 -< e1) <⊔> (run2 -< e2)
  _               ->  run1 -< e1
```

Listing 3.6: Abstract If Expressions

Abstract `If`-expressions implement the same generic interface as their concrete counterpart and therefore are parsed the exact same arguments and inputs [Listing 3.6]. One major distinction to the concrete implementation is the possibility for conditionals to be neither `BoolVal True` nor `BoolVal False`, but `BoolVal Top`. Another value that can be parsed as conditional is `Top`, in which case the type of the value cannot be decided by the abstract interpreter, however there is a possibility that the value might be either `BoolVal True` or `BoolVal False`. Both of those cases have to be accounted for. In consequence, the case distinction is more complex than it is in the concrete case. What remains is that only values that can be considered `#f` lead to the execution of the else-branch. Hence, the first case `BoolVal B.False -> run2 -< e2` remains unchanged. Every value for which there is no possibility to be considered `#f` during any execution will, just as in the concrete case, lead to the execution of the if-branch, which is represented by the last case `_ -> run1 -< e1`. For the remaining cases, which are `BoolVal Top` and `Top`, it cannot be certainly decided whether the if- or else-branch has to be evaluated, both cases are possible. For this reason both branches are evaluated and their resulting values are joined by the widening operator.

```
(define (f n) (if (= n 0)
    1
    (f (- n 1))))
(f 100)
```

Abstact Trace

Environment	Store
`[]`	`[]`
`(define (f n) (...))`	
`[f ↦ (f,[])]`	`[(f,[]) ↦ {λ(n)(...),[f ↦ (f,[])]}]`
`(f 100)`	
`[f ↦ (f,[]),` ` n ↦ (n,[])]`	`[(f,[]) ↦ {λ(n)(...),[f ↦ (f,[])]},` ` (n,[]) ↦ NumVal IntVal]`
`(if (= n 0) 1 (f (- n 1)))`	

```
(= n 0)
BoolVal Top
```

```
1
NumVal IntVal
```

```
(f (- n 1))
```

```
(- n 0)
NumVal IntVal
```

`[f ↦ (f,[]),` ` n ↦ (n,[])]`	`[(f,[]) ↦ {λ(n)(...),[f ↦ (f,[])]},` ` (n,[]) ↦ NumVal IntVal]`
`(if (= n 0) 1 (f (- n 1)))`	

```
NonTerminating
```

```
NumVal IntVal ⊔ NonTerminating
NumVal IntVal
```

Example 3.3: Abstract Evaluation of Recursion

Returning to the over-approximation of loops and recursion. Consider the Scheme program and its abstract evaluation trace depicted in Example 3.3. Of course our analysis is not going to call f a hundred times. Fix-point algorithms allow us to approximate the behaviour of this program. The fix-point of a program guarantees

us that the state of the program is not going to change in further evaluation. The evaluation begins by adding bindings that associate `f` with the for it defined function to the environment and store. Thereafter, `f` is applied to `100`, `100` is evaluated to `NumVal IntVal` and the respective bindings to `n` are added to the environment and store. In this modified context the function body of `f` is evaluated, beginning with the `if`-expression. As already discussed the conditional is evaluated first. `(= n 0)` evaluates to `BoolVal Top` as integers are abstracted to `NumVal IntVal` and it cannot be certainly decided whether two values of type `NumVal IntVal` are truly equal. This causes both branches of the `if`-expression to be evaluated and their resulting values to be joined. `1` evaluates to `NumVal IntVal`, however `(f (- n 1))` evaluates to `NonTerminating`, as `(if (= n 0) 1 (f (- n 1)))` is evaluated in the same context twice. This results in the joining of `NumVal IntVal` and `NonTerminating`. `NonTerminating` is equivalent to the `Bottom` element in our lattice and any value joined with it will result in itself. Hence, fix-point algorithms allow us to analyze programs without evaluating it completely, in this case saving us many evaluation steps.

With most elements of the abstract domain discussed, let us focus again on the implementation of some abstract expressions, beginning with `let`-expressions.

Generic Interfaces

```
alloc :: c Text addr
write :: c (addr,val) ()
extend :: c x y -> c (var,addr,x) y
```

Concrete Instances

```
write = EnvStoreT $ askConst $ \widening ->
  proc (addr, val) -> do
    store <- State.get -< ()
    State.put -<
      Map.insertWith (\old new -> snd (widening old new))
                     addr val store
extend (EnvStoreT f) = EnvStoreT $ proc (var,addr,x) -> do
  env <- Reader.ask -< ()
  Reader.local f -< (Map.insert var addr env, x)
```

Listing 3.7: Abstract Let Expressions

If we remember from the section before, the generic implementation of `Let`-expressions makes use of three different generic interfaces, `alloc`, `write` and `extend`. The abstract implementation of `alloc` has already been discussed. Because the abstract

environment has not been changed significantly from the concrete environment, no changes have to be made to the abstract implementation of the `extend`-interface. The concrete store was changed to an abstract monotone store. To accommodate to this change the implementation of the generic `write`-instance has to be slightly adjusted. The concrete implementation simply inserts new bindings to the store and overwrites bindings when the key to be added was already present. Instead of overwriting bindings the abstract implementation of `write` applies the widening operator to values that share the same key and inserts their joined value.

Because `Let` and `LetRec` use exactly the same generic interfaces, no more changes have to be done to implement abstract letrec-expressions.

The just discussed changes to the implementation of our abstract interpreter perfectly highlight one of the major benefits when implementing abstract analyses in Sturdy. Apart from the changes made to the respective domain, the effort of implementing the necessary generic instances is quite minimal, allowing easy additions of new analyses for programming languages that already have a generic and concrete interpreter implemented in Sturdy.

4 Evaluation

Our original goal was to implement an analysis that can type check Scheme programs. For this two main issues had to be resolved. First, the precision of our analysis has to be good enough to deem the results relevant and meaningful. Second, our analysis has to be able to analyze Scheme programs that make use of high-order functions or otherwise have a complex control-flow. In order to evaluate our implemented analysis[1], we pose two research questions that we answer:

(RQ1) Type Analysis for Scheme: Is our analysis able to perform a relevant and meaningful type analysis on Scheme programs?

(RQ2) Control-Flow Analysis for Scheme: Is our analysis able to analyze Scheme programs with higher-order functions and other complicated control-flows?

We tested our implementation against several Scheme benchmark programs[2], which originate from a paper that implemented and evaluated another abstract analysis for Scheme [2]. Every benchmark for which all necessary functionality is supported in our analysis has been tested, however in particular large programs with more than 50 lines of code did not produce results, due to time and memory constraints. We were able to produce meaningful results for 10 out of 16 benchmarks, presented in the paper's evaluation. The in this evaluation considered benchmarks are from the Gabriel benchmark programs: *cpstak*, *deriv*, *diviter*, *divrec*, *takl* and from the Scala-AM benchmark programs: *collatz*, *gcipd*, *nqueens*, *primtest*, *rsa*.

The meaning of results is different for every abstract analysis, so we will continue by discussing how to interpret the results given by our analysis and their consequences. Our analysis can have three different types of results. First `NonTerminating`, which is returned if the analyzed program does not terminate. Second `MayFail (Errors Val)`, which is returned if at some point in the analysis an operation is called that expects a specific value, but is given a different value. Consider the following program:

```
(define x 1)
(set! x '(1 2))
(if (number? (car x)) 1 1)
```

[1] Publicly available at `https://gitlab.rlp.net/plmz/sturdy/-/tree/scheme/scheme`
[2] Publicly available at `https://gitlab.rlp.net/plmz/sturdy/-/tree/scheme/scheme%2Fscheme_files`

`x` can be either `1` or `'(1 2)`, however `car` expects a list as input and fails for any other input. Instead of failing, our analysis will catch the error that may occur during the evaluation of `(car x)` and continues with the analysis of the program. Because the parsed value *may* contain the expected value `(car x)` evaluates to `Top`. Therefore, our analysis returns the following result for this program: `MayFail (["Excpeted list as argument for car, but got Top"], Int)`. However, the respective expression evaluates to `Bottom`, if the parsed value cannot contain the expected value. The third possible return type of our analysis is `Success Val`, which states that the program is completely type-safe and does not produce any type errors. In order to have a meaningful result, `Val` is required to be another value than `Top` or `Bottom`.

Gabriel and Scala-AM Benchmarks

		k = 0		k = 1	
Benchmark	LOC[a]	Type	#False Positives	Type	#False Positives
cpstak	17	✓	0	——	——
deriv	46	×	3	OOM[b]	OOM[b]
diviter	118 [c]	✓	0	——	——
divrec	117 [d]	✓	0	——	——
takl	18	×	2	✓	0
collatz	17	✓	0	——	——
gcipd	10	✓	0	——	——
nqueens	28	✓	0	——	——
primtest	33	✓	0	——	——
rsa	40	×	1	×	1

[a] measured with cloc
[b] out of memory
[c] large amount only due to formatting, 23 lines of meaningful code
[d] large amount only due to formatting, 21 lines of meaningful code

Table 4.1: Comparison of Concrete and Analysis Results

Table 4.1 summarizes our analysis results. Every benchmark program that our analysis was able to proof type-safety for, is marked with a check-mark. 7 out of the 10 tested benchmarks can be proven type-safe with a context-sensitivity of 0. Three benchmarks *deriv*, *takl* and *rsa* terminated with `MayFail` and produced 3, 2 and 1 false positive errors respectively.

So what happened in the cases in which the analysis returned false positives? To answer the question, let us first consider the abstract evaluation traces for the

application of the `equal?` function of which two of those benchmarks, *deriv* and *takl* make use [Example 4.1].

```
(define (equal? x y)
  (if (eq? x y)
      #t
      (if (and (null? x) (null? y))
          #t
          (if (and (cons? x) (cons? y))
              (and (equal? e (car y))
                   (equal? (cdr x) (cdr y)))
              #f))))
(equal? '(2) '(2))
```

Abstract Trace k=0

Environment	Store
`[]`	`[]`
`(define (equal? x y) (...))`	
`[equal?↦(equal?,[])]`	`[(equal?,[])↦{λ(x y)...}]`
`(equal? '(#t) '(#t))`	
`[equal?↦(equal?,[]),`	`[(equal?,[])↦{λ(x y)...},`
`x↦(x,[]), y↦(y,[])]`	`(x,[])↦'(#t), (y,[])↦'(#t)]`
`(if (eq? x y) #t (if ...))`	
`(if (and (null? x) (null? y)) #t (if ...))`	
`(if (and (cons? x) (cons? y)) (and ...) #f)`	
`(and (equal? (car x) (car y)) ...)`	
`[equal?↦(equal?,[]),`	`[(equal?,[])↦{λ(x y)...},`
`x↦(x,[]), y↦(y,[])]`	`(x,[])↦Top, (y,[])↦Top]`
`(equal? x y)`	
`...`	

Example 4.1: Concrete and Abstract Evaluation of Recursion, k = 0

Abstract Trace k=1

Environment	Store
[ ]	[ ]
1:(define (equal? x y) (...))	
[equal? ↦ (equal?,1)]	[(equal?,1) ↦ {λ(x y)...}]
2:(equal? '(#t) '(#t))	
[equal? ↦ (equal?,1), x ↦ (x,2), y ↦ (y,2)]	[(equal?,1) ↦ {λ(x y)...}, (x,2) ↦ '(#t), (y,2) ↦ '(#t)]
(if (eq? x y) #t (if ...))	
(if (and (null? x) (null? y)) #t (if ...))	
(if (and (cons? x) (cons? y)) (and ...) #f)	
(and 3:(equal? (car x) (car y))	
4:(equal? (cdr x) (cdr y)))	

3:(equal? (car x) (car y))

[equal? ↦ (equal?,3), x ↦ (x,3), y ↦ (y,3)]	[(equal?,1) ↦ {λ(x y)...}, (x,2) ↦ '(#t), (y,2) ↦ '(#t), (x,3) ↦ #t, (y,3) ↦ #t]

(if (eq? x y) #t (if ...))

#t

BoolVal B.True

4:(equal? (cdr x) (cdr y))

[equal? ↦ (equal?,1), x ↦ (x,4), y ↦ (y,4)]	[(equal?,1) ↦ {λ(x y)...}, (x,2) ↦ '(#t), (y,2) ↦ '(#t), (x,3) ↦ #t, (y,3) ↦ #t, (x,4) ↦ '(), (y,4) ↦ '()]

(if (eq? x y) #t (if ...))

#t

BoolVal B.True

#t

BoolVal B.True

Example 4.2: Concrete and Abstract Evaluation of Recursion, k = 1

The evaluation begins by defining a closure, which contains a lambda that takes two arguments `x` and `y` and checks the values that will be bound to them for equality as is defined in `equal?`. The evaluation is straight-forward up to the point

where `equal?` is called recursively with `(car x)` and `(car y)` as its arguments. In this case `(car x)` as well as `(car y)` each evaluate to `NumVal IntVal`. After their evaluation, when `equal?` is applied to `NumVal IntVal` and `NumVal IntVal`, two bindings have to be added to the environment and store. These bindings are `x` $\mapsto$ `(x,[])` and `y` $\mapsto$ `(y,[])`, which have to be added to the environment, as well as `(x,[]` $\mapsto$ `NumVal IntVal)` and `(y,[]` $\mapsto$ `NumVal IntVal)`, which have to be added to the store. Because the keys `(x,[])` and `(y,[])` already exist in the store, the values to be added have to be joined with the already existing values for that key. As discussed, the widening operator [Listing 3.3] is responsible for the joining of values, returning in both cases `Top`. Now with `x` and `y` both associated to `Top`, all branches of all if-statements have to be evaluated. At the very latest when `(equal? (car x) (car y))` comes up again, a type error will occur. Scheme's `car` is only defined for lists, however we cannot be sure it will applied to a list. Due to this imprecision in the evaluation our analysis fails to provide meaningful results in these cases.

One way to solve this issue is to increase the context-sensitivity. The aforementioned execution trace used a context-sensitivity of 0, what if we used a context-sensitivity of 1 instead? As we discussed, our analysis associates each binding with a call-string that represents the last k expressions that have been evaluated when the binding is allocated. The evaluation trace, depicted in Example 4.2, is similar to the evaluation trace with a context-sensitivity of 0. Instead of a binding being associated with the empty context `[]`, now it is associated with the label of the expression that was evaluated last, before the binding was allocated. E.g. when `equal?` is first defined, the expression last evaluated was `(define (equal? x y) (...)`, with label 1. In consequence the address `(equal?,1)` is added to the environment and store, instead of the address `(equal?,[])`. The most important point in the evaluation trace is the evaluation of the following expression:

```
(and 3:(equal? (car x) (car y))
     4:(equal? (cdr x) (cdr y)))
```

For k = 1, our analysis can now differentiate between the bindings that have been created by the original `equal?` call, labeled with 1, and the two recursive `equal?` calls, labeled with 3 and 4. This means that the values that were allocated for the original `equal?` call will never have to be joined with the values of the two recursive `equal?` calls. In addition the length of the lists to be compared is only one, hence only one iteration is necessary to terminate and fresh bindings can be allocated for every argument `x` and `y` of `equal?`. Therefore, our result `BoolVal True` is precise, not only in its type, but also in its concrete value.

However, assume the length of the lists to be compared exceeds one and more than one iteration is necessary. This in turn means `4:(equal? (cdr x) (cdr y))` will be called more than once. Every time new bindings for `x` and `y` have to be allocated and every time this binding will be associated with the same label and the the same variable name, namely `(equal?,4)`. Therefore, beginning with the second iteration values have to be joined again and precision is lost. Nonetheless, one crucial benefit is gained by increasing the context-sensitivity from 0 to 1. Even with larger lists no type errors will occur. Remember for a context-sensitivity of 0 `car` and `cdr` were called with `Top`, when they required the type of their argument to be list. This occurred because the values that were evaluated by `(car x)`, that usually are no lists, were joined with the lists originally parsed to `equal?` as arguments. Because the three `equal?` calls are now separated, this cannot happen. This demonstrates that increasing the context-sensitivity even by just one, can be an effective tool, when proving type-safety.

Unfortunately increasing the context-sensitivity comes with a very high cost in complexity and run-time, allowing only one of these two imprecise benchmarks to benefit from it [Table 4.1]. David Van Horn proves in his dissertation [8] a 0-control-flow analysis to be complete for polynomial time and any k-control-flow analysis with $k > 0$ to be complete for exponential time, verifying our empirical observation of drastically rising run-times for context-sensitivities greater than 0.

The use of the `equal?` function is responsible for producing two false positive errors, one for the application of `car` to `Top` and one for the application of `cdr` to `Top`. Therefore, one false positive for *deriv* and one for *rsa* remain unresolved. Both programs include an `if`-statement of the following form:

```
(if (check x) (foo x) (error "check did not pass"))
```

A check of this sort cannot be certainly declared true in either case, hence the if- and else-branch have to evaluated and their respective values joined. The execution of the else-branch inevitably leads to the introduction of a false positive error, that is caused by low precision.

Let us come back to the example program, which demonstrated the necessity for an analysis that can handle complex control-flow caused by higher-order functions. This program was the following:

```
(let ((f (lambda (x) (x 1)))
      (g (lambda (y) (+ y 2)))
      (h (lambda (z) (+ z 3))))
   (+ (f g) (f h)))
```

Sturdy provides a feature to record the control-flow graph of the analyzed program. The recorded control-flow graph for this program is depicted in Figure 4.1. Remember, the control-flow of this program is hard to statically decide, because the function `1` is applied to, within the `lambda`-expression that is bound to `f`, is only decided at run-time. We implemented the control-flow analysis to be able to associate an expression with a set of expressions that it might transfer control to. We declared the set, which `x` transfers control to, as the following: {`(lambda (y) (+ y 2))`, `(lambda (z) (+ z 3))`}. Therefore, we expect the control-flow graph to illustrate that `x` transfers control to both of theses expression, as both have to be considered whenever `x` is applied.

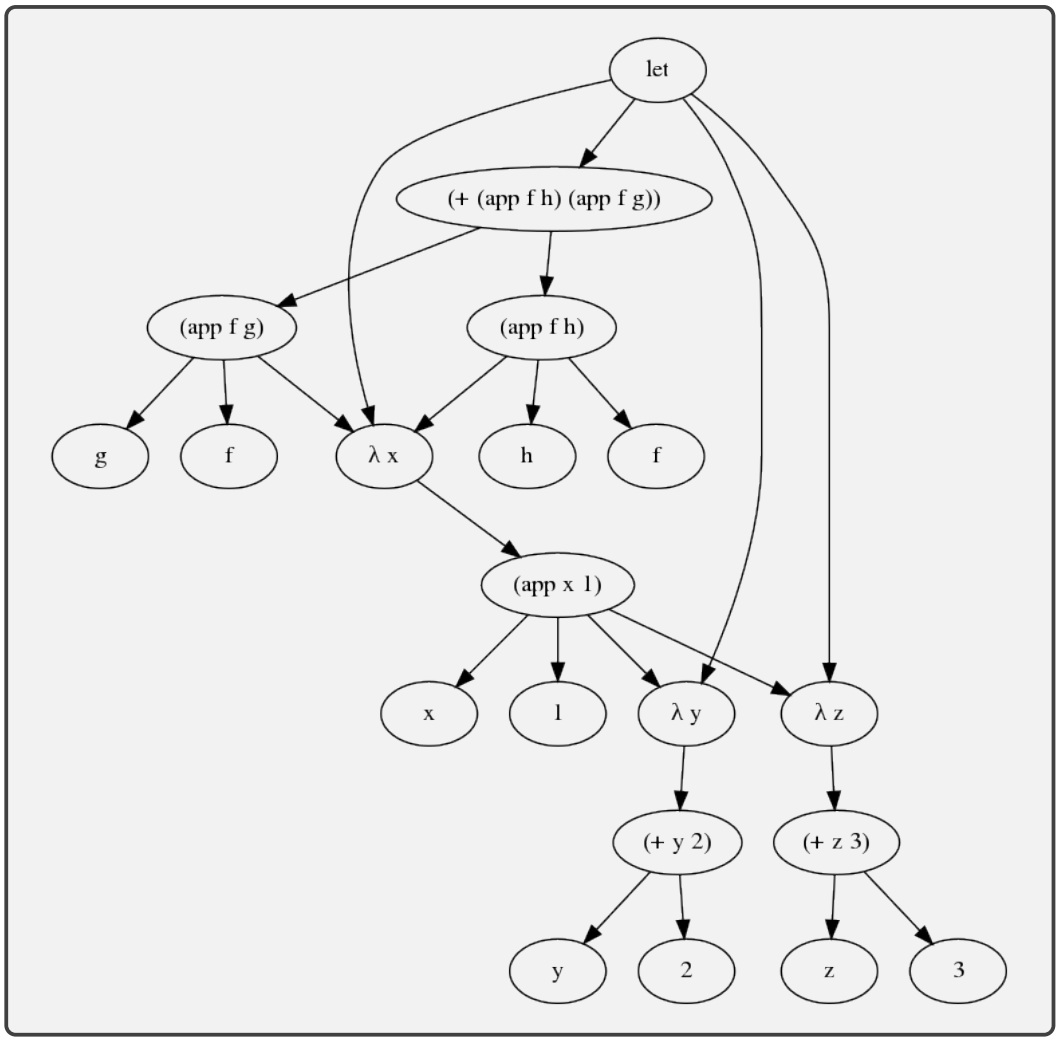

Figure 4.1: Control-Flow Graph of Program with Higher-Order Functions

So how is the control-flow depicted by the control-flow graph? The arrows from one expression to another imply a transfer of control in the direction of the arrow. The control is first handled by the `let`-expression, which parses control to four expressions. This is to be expected as `let` first evaluates its 3 bindings, here depicted as

λ `x`, λ `y` and λ `z`. `let` then transfers control to its body `(+ (app f h) (app f g))`, which transfers control to both of its sub-expressions `(app f g)` and `(app f h)`. Both sub-expressions parse control to two expressions that performs variable look-ups, `(app f g)` to `f` and `g` and `(app f h)` to `f` and `h`. `(app f g)` as well as `(app f h)` have to transfer control to λ `x` to which `f` is bound. λ `x` transfers control to its body `(app x 1)`, which itself transfers control to another expression that performs a variable look-up `x`, and a literal expression `1`. Now it transfers control to λ `y` and λ `z`, which is the exact behavior we expected and wanted from our control-flow analysis.

Lastly we want to answer our posed research questions:

(RQ1) Type Analysis for Scheme: Our implemented type analysis is able to prove 8 out of 10 tested benchmarks type-safe, where a standard type checker most likely would not have been able to prove a single program type-safe. However, the benchmark programs are not large programs by any means, the longest having about 50 lines of meaningful code. Our implementation struggled to process larger programs, due to our analysis having a high complexity in space and time. We conclude, that our implemented analysis and its abstract domain are precise enough to perform a type analysis on small Scheme programs.

(RQ2) Control-Flow Analysis for Scheme: The implemented analysis is able to analyze Scheme programs that have complicated control-flows and use higher-order functions, which we demonstrated by illustrating the control-flow graph for an exemplary Scheme program that uses higher-order functions [Figure 4.1] and of course by being able to analyze our benchmark programs [Table 4.1]. Using the example of the `equal?` function [Example 4.1, Example 4.2], we demonstrated the cost which comes with this kind of analysis. Because our analysis cannot take the order of expressions into account and relies on joining values in the store, a lot of precision can be lost. In turn this means that our type analysis might return false positives, if we cannot guarantee a high context-sensitivity, which is not always possible because of the large increment in complexity. We conclude, that even though the control-flow is analyzed soundly, in practise precise results can only be guaranteed for less complex Scheme programs.

5 Related Work

The issue we encountered when analyzing programs that used `equal?`, is a known issue of control-flow analyses. There are approaches to solve this problem, improving performance as well as precision. We shortly present the two most promising, which were developed by Noah Van Es et al. [2] and Dimitrious Vardoulakis et al. [10].

Noah Van Es et al. use the approach of abstract garbage collection to improve the precision and performance of control-flow analyses. Garbage collection is a known strategy applied during concrete evaluation to free memory that is no longer used and can also be applied during abstract evaluation. It is motivated by the exact problem we encountered in our implementation. A bounded set of addresses can cause addresses to be allocated multiple times for different values. When same addresses are allocated, garbage can be brought *back to life*, in the sense that values that in a concrete evaluation are non-reachable, continue to influence the abstract evaluation. For our analysis this was the case when arguments of the previous `equal?` iteration had to be joined with arguments of the next iteration. Noah Van Es et al. use abstract reference counting to free addresses that are bound to values that cannot be reached. Their implementation guarantees a sound and complete abstract garbage collection, completely removing any garbage. Empirical results prove that precision and performance can be increased significantly. Therefore, abstract garbage collection is able to improve precision without the usual cost in performance. Their approach might also benefit the in this work presented type and control-flow analysis.

Dimitrious Vardoulakis et al. describe the problem we encountered as a mismatch of calls and returns. This is caused by the introduced approximation of unbound recursive calls to a call-string bounded by k. For k=0, as demonstrated for `equal?`, a function can return to any of its callers, it cannot be distinguished e.g. between the original call of `equal?` and the two recursive calls, requiring the joining of values, which causes imprecision. Just as in our case this can lead to false positives, which might weaken the performance of the analysis. The loss of precision can cause spurious paths to be evaluated which in turn can introduce further spurious paths creating a *chain* effect. Dimitrious Vardoulakis et al. present CFA2, the first flow analysis with precise call and return matching for typed and untyped language that make use of higher-order functions and tail calls. Instead of a store and an environment, CFA2 uses a stack and heap, in which variable bindings are stored and looked up. For each function call a new frame is pushed to the stack that holds its

variable bindings including its parsed arguments. Whenever a function returns or terminates the top-most frame is popped from the stack. Therefore, it is guaranteed that each function call has a fresh environment and no need to join values arises. Their empirical results prove their analysis being more precise on a program than either a 0-control-flow analysis or a 1-control-flow analysis.

6 Conclusion and Future Work

We presented an abstract analysis that performs a type and control-flow analysis on small Scheme programs. The analysis is implemented in the Sturdy framework. Sturdy requires abstract analyses to be separated in a generic and abstract interpreter and additionally requires a concrete interpreter. We have implemented these three interpreters for Scheme. The generic interpreter captures the similarities between concrete and abstract interpreters by operating solely on interfaces, which are implemented by the concrete and abstract interpreters respectively. The concrete interpreter performs concrete evaluation on a Scheme program, the abstract interpreter performs said type and control-flow analysis.

A large benefit of analyses implemented in Sturdy is the ease of extensibility. Because every abstract interpreter only instantiates the interfaces of a generic interpreter, it is fairly easy to implement other abstract analyses for a programming language that already has an existing structure implemented in the framework. Therefore, future work on this project may include the addition of other abstract interpreters.

Because this work does not support every feature of Scheme, extending the implemented interpreters to support e.g. vectors, as well as Scheme's `call/cc` present themselves as further additions to this work.

As discussed in the *Related Work* section CFA2 [10] and abstract garbage collection [2] are promising approaches to further improve precision and performance of control-flow analysis. In addition both approaches promise an increase in performance. A comparison in performance and precision between our implemented analysis and a similar analysis utilizing these approaches can yield interesting and relevant results.

Bibliography

[1]Edsger W. Dijkstra. *Notes on Structured Programming, Section 3*. (EUT report. WSK, Dept. of Mathematics and Computing Science; Vol. 70-WSK-03), (EWD; Vol. 249), 1970 (cit. on p. 1).

[2]Noah Van Es, Quentin Stiévenart, and Coen De Roover. *Garbage-Free Abstract Interpretation Through Abstract Reference Counting*. Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 10:1–10:33 (cit. on pp. 29, 37, 39).

[3]John Hughes. *Generalising monads to arrows*. Sci. Comput. Program. 37, 1-3 (2000), 67-111, 2000 (cit. on p. 5).

[4]Sven Keidel and Sebastian Erdweg. *Sound and Reusable Components for Abstract Interpretation*. Proc. ACM Program. Lang. 3, OOPSLA, Article 176 (October 2019), 28 pages, 2019 (cit. on pp. 9, 18).

[5]Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. *Compositional Soundness Proofs of Abstract Interpreters*. Proc. ACM Program. Lang. 2, ICFP, Article 72 (September 2018), 26 pages, 2018 (cit. on pp. 3, 5).

[6]Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis, Chapter 3*. Springer, 1999 (cit. on p. 2).

[7]Alex Shinn, John Cowan, and Arthur A. Gleckler. *Revised^7 Report on the Algorithmic Language Scheme*. 2017 (cit. on p. 8).

[8]David Van Horn. *The Complexity of Flow Analysis in Higher-Order Languages*. Nov. 2013 (cit. on p. 34).

[9]David Van Horn and Matthew Might. *Abstracting Abstract Machines: A Systematic Approach to Higher-Order Program Analysis*. Vol. 54. May 2011 (cit. on p. 19).

[10]Dimitrios Vardoulakis and Olin Shivers. *A Context-Free Approach to Control-Flow Analysis*. Logical Methods in Computer Science, Vol.7 (2:3) 2011, pp. 1-39, 2011 (cit. on pp. 37, 39).

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.

Declaration

I hereby declare that I have written the present thesis independently and without use of other than the indicated means. I also declare that to the best of my knowledge all passages taken from published and unpublished sources have been referenced. The paper has not been submitted for evaluation to any other examining authority nor has it been published in any form whatsoever.

Mainz, March 22, 2020

Tobias Hombücher