Provably Sound Typechecking of JavaScript

Matthijs Bijman

***Abstract*— Since its inception in 1995, JavaScript usage has grown far beyond its initial domain of interactive websites. As the size of applications developed in the language grows, so does the desire for static analysis such as typechecking to provide safety and reliability. Many developments have been made in recent years on increasing the precision of analysis of JavaScript. This work introduces a concrete and a type interpreter for LambdaJS implemented within the Sturdy framework. The goal of this work is to evaluate the feasibility of implementing a shared interpreter using the Sturdy framework, evaluate the correctness of the concrete interpreter, and evaluate the feasibility of proving the typechecker sound. The resulting interpreters are tested experimentally with two test suites. The experimental evaluation gives confidence in the correctness of the concrete and abstract interpreter. A small part of the abstract interpreter is proven sound to evaluate the difficulty of creating a complete soundness proof. The successful implementation of the interpreters shows that implementing a shared arrow-based interpreter within the Sturdy framework is feasible for languages with complex semantics, that the utilities available reduce the effort required to do so, and that proving the abstract interpreter sound is simplified by using the library.**

I. INTRODUCTION

In recent years JavaScript has grown to become one of the most widely used programming languages in the world. Originally created for creating interactive websites, it has far exceeded this purpose and has expanded to areas such as server-side and desktop programming. Developed as a simple scripting language however, its type system is weak and dynamic. Strong type systems can help a programmer in several ways. Simple type system can ensure that variables are structured consistently, while more advanced type systems can guarantee memory safety[1][2], and prevent data races across threads[2]. Moreover, they make static analysis easier due to the abundance of information available through type annotations. This allows modern IDEs to automatically refactor code with high accuracy. Due to the weak and dynamic nature of JavaScript, the type system gives very little guarantees and consequently static analysis tools have been less common, powerful, and accurate[3][4].

Type systems attempt to give strong guarantees about program behaviour, but many contain gaps that cause the typechecker to accept programs which can fault at runtime[5][6]. The extent to which a typechecker is able to reject faulty programs is called the *soundness* of a type system. The weak type system in JavaScript means that it is exceptionally unsound, which can cause unexpected errors to happen at runtime. Scripts running within a browser sandbox might not require very strong guarantees, but the increasing usage of JavaScript in larger, critical applications has lead to the development of tools improving safety by either defining a new language[7], extending JavaScript with annotations[8], or analyzing plain JavaScript[9]. Most of the existing work must allow for gaps in the type system to be compatible with pure JavaScript, or contain complex soundness proof that are difficult to verify.

This work differs from existing work by focusing on proving soundness using Sturdy. It describes a provably sound typechecker for LambdaJS[10], a reduction semantics for JavaScript, implemented using the Sturdy framework[11]. Sturdy is a Haskell library containing utilities for creating abstract interpreters, reducing the burden of proof and enabling compositional soundness proofs. The implementation of this work consists of a concrete interpreter, a typechecker, and a shared interpreter interface describing the shared semantics. Validation of the concrete interpreter is done using two test suites. A set of 98 simple tests testing interpretation of operations and expressions independently, and an existing test suite used to test the LambdaJS desugarer, consisting of 76 tests. The concrete interpreter is also used for testing the abstract interpreter, since the output of the typechecker must always be an approximation of the concrete interpreter. Finally, a part of the typechecker is proven sound to give an indication of the feasibility and complexity of constructing a complete soundness proof for an interpreter implemented using Sturdy.

We evaluate this work by asking and answering three research questions.

- **Q1:** Is it feasible to implement a concrete and abstract arrow-based interpreter for JavaScript using Sturdy?

We specifically investigate whether the utilities offered by the library make the implementation easier.

- **Q2:** Is the implementation of the concrete interpreter correct?

Correctness of the concrete interpreter is important to ensure that the implemented semantics match the LambdaJS specification, since this work is built upon the LambdaJS desugarer.

- **Q3:** Is it feasible to prove the abstract interpreter sound?

While proving the entire typechecker sound is outside of the scope of this work, a soundness proof for a small part of the typechecker is included, the exception semantics, to give an indication of the complexity of a complete proof.

```
class Arrow c => AbstractValue v c | c -> v where
  -- values
  objectVal :: c [(Ident, v)] v
  getField :: c Expr v -> c (v, Expr) v
  -- environment/store ops
  lookup :: c Ident v
  apply :: c Expr v -> c (v, [v]) v
  set :: c (v, v) ()
  get :: c v v
  -- control flow
  label :: c Expr v -> c (Label, Expr) v
  break :: c (Label, v) v
  catch :: c Expr v -> c (Expr, Expr) v
  throw :: c v v
  ... [some omitted]
```

Fig. 1. Part of the arrow interface implemented by both interpreters

```
eval :: (ArrowChoice c, AbstractValue v c,
Show v) => c Expr v
eval = proc e -> do
  case e of
    EId id -> lookup -< id
    EApp body args -> do
      lambda <- eval -< body
      args <- mapA eval -< args
      apply eval -< (lambda, args)
    EGetField objE fieldE -> do
      obj <- eval -< objE
      getField eval -< (obj, fieldE)
    ESetRef locE valE -> do
      loc <- eval -< locE
      val <- eval -< valE
      set -< (loc, val)
      returnA -< loc
    EThrow e -> do
      val <- eval -< e
      throw -< val
    EFinally e1 e2 -> do
      res <- eval -< e1
      eval -< e2
      returnA -< res
    ... [some omitted]
```

Fig. 2. Part of the shared interpreter, with calls to the functions of the interface defined in fig. 1

```
newtype ConcreteArr x y = ConcreteArr
  (Except
    (Either String Exceptional)
    (Environment Ident Value
      (StoreArrow Location Value
        (State Location (->)))) x y)
```

Fig. 3. Concrete arrow transformer stack

```
newtype TypeArr x y = TypeArr
  (Except
    String
    (Environment Ident Type'
      (StoreArrow Location Type'
        (State Location (->)))) x y)
```

Fig. 4. Abstract arrow transformer stack

II. SEMANTICS

Understanding the semantics of the subject language is necessary to understand the challenges in creating a sound and precise static analysis. This section will describe part of the semantics of LambdaJS. The full semantics of LambdaJS are described by Guha et al.[10]. The semantics discussed here are limited to exceptions, labels, references, objects, and sequences of expressions, as these are largely unique to LambdaJS. The language of course includes operations for arithmetic, control flow, etc. but the semantics of these expressions are greatly simplified in the desugaring process of JavaScript to LambdaJS, and are commonly found in other languages. The entire interface implemented by the concrete and abstract interpreters is shown in fig. 1. A shared interpreter, shown in fig. 2, uses the interface to delegate the actual interpretation to either the concrete or abstract interpreters. This way the common interpretation structure between the implementations is shared. An example of this is the *finally* expression. The interpretation of this expression does not rely on the interface. This means that a new implementation of the interface does not need to explicitly support this expression, and that a soundness proof for such an implementation does not need to prove *finally* sound.

```
try {
  throw 1.0;
} catch (x) {
  x;
}
```

```
(ECatch
  (EThrow (ENumber 1.0))
  (ELambda ["x"] (EId "x")))
```

Fig. 5. Example of throw and catch semantics, evaluates to 1.0

A. Exceptions

Exceptions in LambdaJS are equivalent to exceptions in JavaScript, meaning arbitrary values can be thrown and caught. A *throw* expression is accompanied by a value, and will unwind the stack propagating this value until a *catch* expression is found. The *finally* expression contains two expressions. The first expression is evaluated, after which the second is evaluated regardless of thrown exceptions in the first evaluation, with the result of the first expression then being returned. The combination of *throw*, *catch*, and *finally* is enough to implement the common try-catch-finally structure in JavaScript, even though evaluation of *finally* is orthogonal to the other two expressions. The implementation of exceptions is done by implementing the *throw* and *catch* functions of the abstract interface shown in fig. 1. The *finally* expression is expressed in terms of the rest of the interface and does not need to be specialized. An example of *try* and *catch* semantics is shown in fig. 5. The first branch of the *catch* expression throws the value 1.0. This exception is then caught in the second branch, which simply returns the value.

Concrete interpretation of exceptions is implemented using the *Except* arrow transformer, parameterized with the *Exceptional* type, as visible in fig. 3. This allows the interpreter to return an exceptional value indicating an exception occurred, which can then be handled explicitly by the implementation of the *catch* expression.

The abstract interpreter does not return an exceptional value as visible in fig. 4 in the lack of the *Exceptional* type, since it is unknown during analysis if the *throw* expression is actually executed at runtime. Instead, the thrown value is added to the set of possible output types of that expression, which then propagates upwards as long as the expression remains uncaught.

```
1  label f1 = proc (l, e) -> do
2    (l, res) <- tryCatchA (second f1) (proc
3      ((label, _), err) ->
4        case err of
5          Left s -> failA -< Left s
6          Right (Break l1 v) -> case l1 == label of
7            True  -> returnA -< (label, v)
8            False -> failA -< (Right \$ Break l1 v)
9          Right (Thrown v) -> do
10           failA -< (Right \$ Thrown v)) -< (l, e)
11   returnA -< res
12
13 break = proc (l, v) -> do
14   failA -< Right (Break l v)
```

Fig. 6. Implementation of the *label* and *break* arrow operations

B. Labels

Labels in LambdaJS are a more powerful version of labels in JavaScript. Whereas in JavaScript only labels can only be used with loops and referenced in continue and break statements, LambdaJS allows for labeling any expression. If the expression contained within a label breaks to that label, the value accompanying the break is the result of the expression. If the contained expression does not break, or breaks to a different label, then the label expression propagates the result. The concrete implementation of labels is shown in fig. 6. The implementation uses the *tryCatchA* arrow operation, part of Sturdy, for implementing exceptional control flow. By using this arrow operation, we can intercept exceptional values to check if the value was the result of a break (case on line 6), and if so, if the accompanying label name matches the name of this label (comparison on line 6). If this is the case, we return the wrapped value (case on line 7). In all other cases (thrown value, interpretation error, or normal value) we propagate the result.

An example of a label expression is shown in fig. 7. The LambdaJS code wraps a *seq* expression with a *label* expression. The first branch of the snippet breaks to the label with the numeric value of 1.0, while the second branch returns the value *undefined*. The result of the expression is the number 1.0, as the second branch of the *seq* expression is never reached.

```
var x;
label1:
{
  x = 1.0;
  break label1;
  x = undefined;
}
x;
```

```
(ELabel
  (Label "label1")
  (ESeq
    (EBreak (Label "label1") (ENumber 1.0))
    (EUndefined)))
```

Fig. 7. Example of label semantics, evaluates to 1.0. The JavaScript snippet does not directly desugar to the LambdaJS code but shows similar semantics.

Similarly to exceptions, concrete interpretation of labels is implemented using the *Except* arrow transformer, parameterized with the *Exceptional* type. The *Exceptional* type can either contain a thrown value or a pair of a label and a value. The first corresponds to an exception, the second corresponds to a label. The implementation of the *Label* expression explicitly handles exceptional return values. If it contains a label-value pair with a matching label, the expression results in the accompanying value. If the exceptional value contains a thrown value or if the label doesn't match, it propagates the value upwards.

The abstract interpretation of labels happens in the same way as exceptions. The label type is added to the set of possible result types and is propagated as long as no matching label is found. No stack unwinding happens in abstract interpretation, since it is unknown during abstract interpretation if the exception is thrown at runtime.

C. References

References are the primary mechanism for allowing read and write side-effects. References are explicitly created and dereferenced in LambdaJS as opposed to JavaScript. A reference expression evaluates an expression, puts the result on the heap, and returns the location of the value on the heap. Dereferencing this location evaluates to the value on the heap corresponding to that location, and a value on the heap can be updated. Identifiers in LambdaJS always evaluate to a location (if the identifier is valid), and thus must be explicitly dereferenced. An example of references is shown in fig. 8. First, the variable *a* is declared and set to 1.0. Then in the first branch of the *seq*, the value is updated to 2.0. Finally, in the second branch, the value of *a* is returned. If this expression was implemented without references it would evaluate to 1.0, since the mutation in the first branch of the *seq* expression would not propagate to the second branch.

```
var a = 1;
a = 2.0;
a;
```

```
(ELet
  [("a", ERef $ ENumber 1.0)]
  (ESeq
    (ESetRef (EId "a") (ENumber 2.0))
    (EDeref (EId "a"))))
```

Fig. 8. Example of reference semantics, evaluates to 2.0

```
1  getField = proc (VObject obj, VString name) -> do
2    let fieldV = find (\(fn, fv) -> fn == name) obj
3    case fieldV of
4      -- E-GetField
5      Just (n, v) -> returnA -< v
6      Nothing ->
7        let proto = find isProtoField obj in
8          case proto of
9            -- E-GetField-Proto-Null
10           Just (_, VNull) -> do
11             returnA -< VUndefined
12           -- E-GetField-Proto
13           Just (_, VRef l) -> do
14             protoVal <- read -< (l, ())
15             getField_ -< (protoVal, VString name)
16           -- Other type of proto
17           Just (_, _) -> returnA -< VUndefined
18           -- E-GetField-NotFound
19           Nothing -> returnA -< VUndefined
```

Fig. 9. Implementation of the GetField operation

Concrete interpretation of references makes use of the *Environment* and *Store* arrow transformers as visible in fig. 3. The environment contains a mapping from identifiers to locations, and the store contains a mapping of locations to values. The location corresponding to an identifier is lost once the scope containing the variable declaration is exited, i.e. a *let* or *lambda* expression. An identifier introduced by these expressions is only visible in the expressions within it. Appropriately, the environment is not part of the output of evaluating an expression.

Abstract interpretation of references closely matches concrete interpretation. The environment contains a mapping from identifiers to sets of locations, as the location an identifier might map to can depend on the control flow of the program. Similarly, the store contains a mapping from locations to sets of types, because the values written to certain locations can depend on control flow of the program. The differences between the concrete arrow transformer stack and the abstract arrow transformer stack are visible in fig. 3 and in fig. 4 respectively, by noting that the *Environment* and *StoreArrow* are parameterized with different types: *Location* vs. *Location'* (set of locations), and *Value* vs. *Type'* (set of types).

D. Objects

Objects in LambdaJS are the immutable equivalent of JavaScript objects. Operators for getting, setting, and deleting fields of an object return a new object with the changes applied to it. To change an object on the heap a combination of operations must be performed. A location must be dereferenced to obtain the value, the desired mutation must be applied, and the location on the heap must be updated to the resulting object. Attempting to read a field of an object that does not contain the field evaluates to an undefined value, similar to JavaScript. Deleting a non-existent field returns the original object. Updating a non-existent field is the same as creating a new field with the given value. An example of a *getfield* expression is shown in fig. 10. The example shows that reading a field of an object will walk up the prototype chain if the field cannot be found.

Prototypes are supported in LambdaJS, with simplified semantics compared to JavaScript. When a read operation is performed on an object not containing the specified field, and if that object contains a prototype object, the interpreter attempts to read the field from the prototype object, moving up the chain of prototypes until the field is found or until no prototype object is found. The implementation for this operation is visible in fig. 9. The implementation returns the field with the given name if is found (line 5). If not, it searches for the prototype field. If this field exists and is a reference, a recursive call is made searching for the same field but in the prototype. If the prototype doesn't exist or is of the wrong type the function returns *undefined*. Prototypes are the most complex part of these semantics. The delete and write operations do not consider prototypes and are thus considerably simpler.

```
({
  a: 1.0,
  b: 2.0,
  __proto__: { c: 3.0 }
}).c;
```

```
(EGetField
  (EObject
    [("a", ENumber 1.0),
    ("b", ENumber 2.0),
    ("$proto",
      ERef (EObject [("c", ENumber 3.0)]))])
  (EString "c"))
```

Fig. 10. Example of object semantics, both snippets evaluate to 3.0

Objects in the concrete interpretation are simply implemented using a map from identifiers to values, with some special logic for handling non-existent fields. The abstract interpreter can deal with objects in a variety of ways, greatly influencing the accuracy of the analysis. For simplicity, only references to fields of an object that are certain to exist are considered valid. If the existence

of a field within an object depends on control flow, it is considered to be a type error if that field is referenced.

E. Sequential Expressions

There are no blocks in LambdaJS like in JavaScript. The body of a function is only a single expression. To support a sequential list of statements an expression consisting of two expressions is implemented. These expression are evaluated sequentially. The result of the entire expression is the result of the second contained expression. An example of a *seq* expression can be seen in fig. 8, where both branches are executed sequentially. If the first expression throws an exception or breaks to a label then the second expression is not evaluated.

Concrete interpretation of a sequential expression is simple. The expression evaluated first can only have an effect on the output of the program either by mutating the store or by producing an exceptional value (i.e. label or throws).

Abstract interpretation follows similar semantics. If the set of types produced by the first expression contains an exceptional type, this exceptional type is merged with the output of the second expression. Moreover, mutations to the store from the first expression are propagated to the store used in the interpretation of the second expression.

III. EVALUATION

This section contains the evaluation of the research. The goal of this work was to answer three research questions. First, is it feasible to implement a concrete and abstract arrow-based interpreter for JavaScript using Sturdy? Second, is the concrete interpreter correct?. And finally third, is it feasible to prove the abstract interpreter correct? For each question, the limitations and open questions are also discussed.

A. Q1: Implementing Interpreters

Implementing a concrete and abstract arrow-based interpreter for JavaScript has been done by implementing the interpreters for LambdaJS and using the existing JavaScript to LambdaJS desugarer. The complexity of the LambdaJS specification is much lower than the JavaScript specification, thus making the implementation of the interpreters feasible. Moreover, the utilities present in the Sturdy library such as the *Store*, *Environment*, and arrow operations such as *tryCatchA* made the implementation of the effects of the language very concise.

The results of testing indicate that both the concrete and abstract interpreter implement the majority of the specification properly. We conclude that it is therefore feasible to implement a concrete and abstract arrow-based interpreter for LambdaJS using Sturdy. Since this work is based on the JavaScript to LambdaJS desugarer we can extend this conclusion to include JavaScript.

While it is shown that it is feasible to implement a JavaScript interpreter using Sturdy, there are some limitations of the current implementation that must be mentioned. First, the *eval* statement of JavaScript is currently not implemented. This is due to limitations of the desugarer, but there is work ongoing into fixing this limitation. Second, parts of the JavaScript standard library are not implemented. This is similarly due to limitations of the desugarer, but are more easily fixed than the *eval* issue. Third, there are some missing features in the interpreter itself, e.g. regex operations, since their implementation was not directly important to answering the research question. Finally, while the typechecker supports the same features as the concrete interpreter, the precision of the interpreter is limited, and thus its usability. An interesting continuation of this work would be to improve the precision while maintaining soundness.

B. Q2: Correct Concrete Interpreter

Correctness of the concrete interpreter is an important property as it implies adherence to the LambdaJS specification, which is necessary for the interpreter to properly process the output of the JavaScript to LambdaJS desugarer. If the interpreter contains large gaps or errors, the output of the desugarer would be wrongly processed.

The concrete interpreter has been tested thoroughly, giving confidence in its correctness. The interpreter has been tested with a total of 174 tests. 98 of these tests consist of testing basic features such as math operations, branching, function calls, and objects. 76 tests, adapted from the LambdaJS project, test more complex features such as scoping, environments, and closures. These tests each include the initialization of the entire standard library. Morover, they test the entire language pipeline, as they are parsed as JavaScript, desugared to LambdaJS code, and finally interpreted by the concrete interpreter. There are 3 tests in this test suite that do not produce the correct result, due to missing implementations of $Array.prototype.length$ and $Array.prototype.join$.

This test suite gives confidence in its correctness, however it is no guarantee that it is fully correct. The test suite used is thorough enough to ensure that the most important semantics of the LambdaJS semantics are implemented properly. Future work could include expanding the test suite to include all LambdaJS tests, including those derived from the SpiderMonkey JavaScript engine.

C. Q3: Sound Abstract Interpreter

The soundness of the type interpreter guarantees that the output it produces is an approximation of the concrete interpreter, i.e. the concrete interpreter will never output a value that is not part of the approximation made by the abstract interpreter. This characteristic can be formally defined, as in eq. 1. Soundness of the type interpreter is an important property. An unsound typechecker can produce incorrect results, meaning the properties that it attempts to prove as part of the typechecking process (e.g. reject programs that contain invalid operations) cannot be taken as guarantees.

$$\alpha(\{eval\ e\ p \mid p \in \gamma(\hat{p})\}) \subseteq \hat{eval}\ e\ \hat{p} \tag{1}$$

For many applications complete soundness of the languages semantics and typechecker is not a necessity. This allows some typecheckers to sacrifice soundness in exchange for better precision and speed, or to allow a language to include features that are desirable but not safe. Other applications require strong guarantees about program properties (e.g. memory safety) which poses restrictions on the semantics of the programming language used. While JavaScript was not designed for use in critical applications, and its semantics allow for many potentially unwanted or implicit operations, the language has become commonly used in applications with higher reliability requirements. This has created a demand for a sound typechecker that can improve reliability of these applications by proving these operations do not happen.

The goal of this work is to develop a typechecker for LambdaJS within the Sturdy framework and investigate the effort required to prove it sound. Creating a soundness proof for the entirety of the typechecker is outside of the scope of this work. Instead, a small part of the semantics is proven sound to give an indication of the complexity of such a proof. The part of the semantics that is proven sound is the typechecking of exception throwing and catching. The benefit of using Sturdy for creating a sound typechecker is that the soundness of each arrow operation (shown in 1) can be proven seperately. Moreover, the shared code (shown in 2) must only be proven sound once, instead of for each implementation.

The soundness proof of exceptions can be found in the Appendix. This proof indicates that a full soundness proof is feasible, but more effort is required before this can be confirmed. The complete set of operations that must be proven sound can be seen in fig. 1. The difficulty of proving an operation sound is highly dependent on the complexity of the operation. Some simple operations in the interface, such as *numVal*, *boolVal*, *nullVal*, are trivial to prove sound. Some complex operations that use the Sturdy library such as *lookup*, *set*, *new*, and *get*, are much easier to prove sound due to them relying on the Sturdy library. Finally, some operations such as *getField*, *apply*, *label*, and *break* are complex, and are less dependent on the Sturdy library, and as such their soundness proof will be more complex.

For a more conclusive answer, the remaining complex operations of the interface should also be proven sound. Particularly the soundness proofs of the operations that heavily rely on the Sturdy library will give an indication of the reduction in proof complexity due to the usage of the library utilities. Ideally a full soundness proof would be done as this would give a full view of the complexity of the proof and confirm that it is feasible to prove a language with complex semantics sound with the help of the Sturdy library.

IV. RELATED WORK

The research questions of this work are aimed at the implementation of shared arrow-based interpreters within the Sturdy framework. These questions are specific to this framework, and the typechecker is not very precise. Nevertheless, we discuss related work in the area of static analysis of JavaScript to discuss applications and future work of this research.

LambdaJS[10] is a small reduction semantics for JavaScript. Creating a typechecker for the entire semantics of JavaScript is infeasible due to the large and informal specification. By limiting the scope of the interpretation to LambdaJS instead of JavaScript, the surface area that must be covered is greatly reduced. The reduced complexity of the specification is what enabled this work to implement a comprehensive interpreter and typechecker, and to test it with real JavaScript code.

Jensen et al.[9] developed a pure JavaScript program analyzer, with no type annotations or other changes to the language. Because the subject language is pure JavaScript, the analysis can be easily adopted for use in existing projects. The results of their work show reasonable precision in a set of benchmarks, however in some cases the runtime performance was low and memory usage was high. This work is relevant as it also aims to develop an analyzer without changing the JavaScript syntax. The work also mentions soundness, but no proof was present. Future work could use this proof as a benchmark to compare the complexity of the soundness proof.

TeJaS[12] is a framework for building type systems for JavaScript. Due to the dynamic nature of JavaScript, what should be considered an error differs from project to project. TeJaS defines a core type language for JavaScript that can be extended to include domain specific rules. Their work has been validated by extending the core type system with several extensions. While the goal of TeJaS is different from the goal in this work, their approach on domain specific rules is interesting. By implementing a typechecker with such rules using Sturdy, a large part of the interpreter could potentially be shared with an existing implementation. Interesting future work would be to investigate the feasibility of proving a large amount of similar analyzers (differing by

domain specific rules) with many shared semantics.

TypeScript[7] is a strict superset of JavaScript, adding classes, interfaces, modules, and gradual typing to the language. The gradual type system makes it possible to transform an existing codebase towards typechecking. Moreover, the compiler emits JavaScript, and thus TypeScript is usable in the same environments as JavaScript. TypeScript is widely used and under active development by Microsoft. While the semantics of TypeScript allows for typechecking beyond what is possible with regular JavaScript, it still requires a user to change their code before analysis becomes useful. The scope of TypeScript extends beyond improving the safety of JavaScript by providing programmers with tools to write more maintainable software. Soundness of the desugaring of TypeScript to JavaScript is essential as mistakes in this process could lead to faults in many applications. A potential application for this work could be to create a provably sound analysis with domain specific rules for checking the generated JavaScript code.

FlowJS[8] is a type interpreter for plain JavaScript with optional type annotations to improve precision. Similarly to TypeScript it allows for gradual addition of type annotations to improve results of the analysis. It differs from TypeScript in that it does not add language features beyond type annotations. The type inference algorithms used by Flow has been proven sound. Similarly to TypeScript, FlowJS has a considerably different approach to improving the safety of JavaScript. The inference algorithm used by FlowJS could be used to improve the precision of the typechecker presented in this work. The complexity of the inference algorithm is much greater than that of the typechecker, which would lead to interesting challenges in maintaining soundness.

V. FUTURE WORK

This work has been limited to investigating the feasibility of implementing sound and/or correct interpreters using Sturdy. The positive results indicate that future work could be worthwhile in better understanding the value of Sturdy. A complete soundness proof for the typechecker presented in this work would make it possible to better understand the role of Sturdy in reducing proof complexity, with existing soundness proofs for JavaScript analyzers serving as comparison. Specifically a complete soundness proof for the complex semantics such as objects and exceptions would be valuable.

To the same end, it would be valuable to implement additional abstract interpreters under the same interface to evaluate the benefit of proof composition enabled by Sturdy. Especially abstract interpreters that share much of their implementation, e.g. typecheckers with various levels of strictness, could benefit from simpler soundness proofs.

Other areas for future work could include creating analyzers with high precision, as presented in related work, while maintaining soundness. To create usable and valuable abstract interpreters the precision must be improved. An interesting challenge would be to create several sound abstract interpreters for various domains, e.g. control flow analysis, escape analysis, heap analysis, etc., combining their results to create more precise type analysis.

VI. CONCLUSION

The goal of this work was to research the feasibility of implementing correct and sound interpreters for JavaScript within the Sturdy framework. Specifically the goal was to evaluate the feasibility of implementing the complete JavaScript semantics, testing the correctness of the concrete interpreter, and to evaluate the feasibility of proving the abstract interpreter sound.

By using the JavaScript to LambdaJS desugarer, the complexity of the interpreters was reduced considerably. As the test results show, the majority of the semantics have been implemented. Moreover, the results give a reasonable confidence in the correctness of the concrete interpreter. A small part of the abstract interpreter has been proven sound, giving an indication of the complexity of a full soundness proof. The partial proof is a positive result indicating that a full soundness proof is feasible.

REFERENCES

[1] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim, "Safe manual memory management in cyclone," *Science of Computer Programming*, vol. 62, no. 2, pp. 122–144, 2006.

[2] N. D. Matsakis and F. S. Klock, II, "The rust language," *Ada Lett.*, vol. 34, pp. 103–104, Oct. 2014.

[3] M. Schäfer, "Refactoring tools for dynamic languages," in *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, (New York, NY, USA), pp. 59–62, ACM, 2012.

[4] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip, "Tool-supported refactoring for javascript," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, (New York, NY, USA), pp. 119–138, ACM, 2011.

[5] N. Amin and R. Tate, "Java and scala's type systems are unsound: the existential crisis of null pointers," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 838–848, ACM, 2016.

[6] "Rustlang Unsoundness Issues." `https://github.com/rust-lang/rust/issues?q=is%3Aopen+is%3Aissue+label%3A%22I-unsound+%F0%9F%92%A5%22`, 2018. [Online; accessed 7-June-2018].

[7] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript," in *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, (New York, NY, USA), pp. 257–281, Springer-Verlag New York, Inc., 2014.

[8] A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi, "Fast and precise type checking for javascript," *CoRR*, vol. abs/1708.08021, 2017.

[9] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *International Static Analysis Symposium*, pp. 238–255, Springer, 2009.

[10] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of javascript," in *European conference on Object-oriented programming*, pp. 126–150, Springer, 2010.

[11] "Sturdy." https://github.com/svenkeidel/sturdy, 2018. [Online; accessed 7-June-2018].
[12] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi, "Tejas: Retrofitting type systems for javascript," *SIGPLAN Not.*, vol. 49, pp. 1–16, Oct. 2013.

APPENDIX

SOUNDNESS PROOF EXCEPTION SEMANTICS

A. Concrete and Abstract Semantics

Concrete

$$\frac{e_1 \hookrightarrow except\ v_1 \quad e_2[x := v_1] \hookrightarrow v_2}{catch\ e_1\ e_2 \hookrightarrow v_2}$$

$$\frac{e_1 \hookrightarrow v}{catch\ e_1\ e_2 \hookrightarrow v}$$

Abstract

$$\frac{e_1 \hookrightarrow t_1 \quad except \in t_1 \quad e_2 \hookrightarrow t_2}{catch\ e_1\ e_2 \hookrightarrow t_1 \cup t_2}$$

$$\frac{e_1 \hookrightarrow t_1 \quad except \in t_1}{catch\ e_1\ e_2 \hookrightarrow t_1}$$

B. Proof

Hypothesis 1

$$\alpha(\{eval\ e_1\ p \mid p \in \gamma(\hat{p})\}) \subseteq \hat{eval}\ e_1\ \hat{p}$$

Hypothesis 2

$$\alpha(\{eval\ e_2\ p \mid p \in \gamma(\hat{p})\}) \subseteq \hat{eval}\ e_2\ \hat{p}$$

Goal

$$\begin{aligned} & \alpha(\{eval\ (catch\ e_1\ e_2)\ p \mid p \in \gamma(\hat{p})\}) \\ \subseteq\ & \hat{eval}\ (catch\ e_1\ e_2)\ \hat{p} \end{aligned}$$

Case Distinction

Case 1

$$\hat{eval}\ e_1\ \hat{p} = X \cup \{\hat{Except}\} \mid \hat{Except} \notin X$$

From rule 1 of the abstract semantics it follows:

$$\hat{eval}\ (catch\ e_1\ e_2)\ \hat{p} \equiv \hat{eval}\ e_1\ \hat{p}\ \cup\ \hat{eval}\ e_2\ \hat{p} \qquad (2)$$

With this we proof our goal as follows:

$$\begin{aligned} & \alpha(\{eval\ (catch\ e_1\ e_2)\ p \mid p \in \gamma(\hat{p})\}) \\ \subseteq\ & \alpha(\{eval\ e_1\ p \mid p \in \gamma(\hat{p}) \wedge eval\ e_1\ p \in \gamma(X)\} \\ & \quad \cup \{eval\ e_2\ p \mid p \in \gamma(\hat{p}) \wedge eval\ e_1\ p \in \gamma(\{\hat{Except}\})\}) \quad \textit{[case dist.]} \\ \subseteq\ & \alpha(\{eval\ e_1\ p \mid p \in \gamma(\hat{p})\} \cup \{eval\ e_2\ p \mid p \in \gamma(\hat{p})\}) \quad \textit{[set laws]} \\ \subseteq\ & \alpha(\{eval\ e_1\ p \mid p \in \gamma(\hat{p})\}) \cup \alpha(\{eval\ e_2\ p \mid p \in \gamma(\hat{p})\}) \quad \textit{[Galois laws]} \\ \subseteq\ & \hat{eval}\ e_1\ \hat{p}\ \cup\ \hat{eval}\ e_2\ \hat{p} \quad \textit{[Hypothesis 1 and 2]} \\ \subseteq\ & \hat{eval}\ (catch\ e_1\ e_2)\ \hat{p} \quad \textit{[from eq. 4]} \end{aligned}$$

Case 2

$$\hat{eval}\ e_1\ \hat{p} = X \mid \hat{Except} \notin X$$

From rule 2 of the abstract semantics it follows:

$$\hat{eval}\ (catch\ e_1\ e_2)\ \hat{p} \equiv \hat{eval}\ e_1\ \hat{p} \qquad (3)$$

With this we proof our goal as follows:

$$\begin{aligned} & \alpha(\{eval\ (catch\ e_1\ e_2)\ p \mid p \in \gamma(\hat{p})\}) \\ \subseteq\ & \alpha(\{eval\ e_1\ p \mid p \in \gamma(\hat{p}) \wedge eval\ e_1\ p \in \gamma(X)\}) \quad \textit{[case dist.]} \\ \subseteq\ & \alpha(\{eval\ e_1\ p \mid p \in (\hat{p})\}) \quad \textit{[set laws]} \\ \subseteq\ & \hat{eval}\ e_1\ \hat{p} \quad \textit{[Hypothesis 1]} \\ \subseteq\ & \hat{eval}\ (catch\ e_1\ e_2)\ \hat{p} \quad \textit{[from eq. 5]} \end{aligned}$$

By combining the results of the case distinction we prove our goal:

$$\begin{aligned} & \alpha(\{eval\ (catch\ e_1\ e_2)\ p \mid p \in \gamma(\hat{p})\}) \\ \subseteq\ & \hat{eval}\ (catch\ e_1\ e_2)\ \hat{p} \end{aligned}$$

□