Lazy Interworking of Compiled and Interpreted Code for Sandboxing and Distributed Systems

Camil Staps
info@camilstaps.nl
Radboud University Nijmegen
Nijmegen, The Netherlands

John van Groningen
johnvg@cs.ru.nl
Radboud University Nijmegen
Nijmegen, The Netherlands

Rinus Plasmeijer
rinus@cs.ru.nl
Radboud University Nijmegen
Nijmegen, The Netherlands

ABSTRACT

More and more applications rely on the safe execution of code unknown at compile-time, for example in the implementation of web browsers and plugin systems. Furthermore, these applications usually require some form of communication between the added code and its embedder, and hence a communication channel must be set up in which values are serialized and deserialized. This paper shows that in a functional programming language we can solve these two problems at once, if we realize that the execution of extra code is nothing more than the deserialization of a value which happens to be a function. To demonstrate this, we describe the implementation of a serialization library for the language Clean, which internally uses an interpreter to evaluate added code in a separate, sandboxed environment. Remarkable is that despite the conceptual asymmetry between "host" and "interpreter", lazy interworking must be implemented in a highly symmetric fashion, much akin to distributed systems. The library interworks on a low level with the native Clean program, but has been implemented without any changes to the native runtime system. It can therefore easily be ported to other programming languages.

We can use the same technique in the context of the web, where we want to be able to share possibly lazy values between a server and a client. In this case the interpreter runs in WebAssembly in the browser and communicates seamlessly with the server, written in Clean. We use this in the *iTasks* web framework to handle communication and offload computations to the client to reduce stress on the server-side. Previously, this framework cross-compiled the Clean source code to JavaScript and used JSON for communication. The interpreter has a more predictable and better performance, and integration is much simpler because it interworks on a lower level with the web server.

CCS CONCEPTS

• **Software and its engineering** → **Functional languages**; • **Information systems** → *Browsers*; • **Computer systems organization** → *Client-server architectures.*

IFL'19, September 25–27, 2019, Singapore, Singapore

ACM ISBN 978-1-4503-7562-7/19/09…$15.00
https://doi.org/10.1145/3412932.3412941

KEYWORDS

interpreters, functional programming, laziness, sandboxing, WebAssembly

ACM Reference Format:
Camil Staps, John van Groningen, and Rinus Plasmeijer. 2019. Lazy Interworking of Compiled and Interpreted Code for Sandboxing and Distributed Systems. In *Implementation and Application of Functional Languages (IFL '19), September 25–27, 2019, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3412932.3412941

1 INTRODUCTION

The execution of untrusted code, or code that is unknown to the executable when it is started, has become paramount in a large number of different contexts. The most common example is the web browser, which must run JavaScript and WebAssembly code that can interact with the web page but should under no circumstance crash the browser or the rendering engine.

Another use case can be found in plugin systems. We may for example imagine a compiler which can load plugins for various language extensions, where plugins can provide functions for transforming the abstract syntax tree. In this case the plugin has to interwork on an even more fine-grained level with the host program, interacting closely with the same data types.

A similar need is felt in workflow systems like the *iTasks* framework for Task-Oriented Programming [24, 25]. Applications written in this framework center around various tasks and the workflows that users walk through to execute them. Currently, applications must be recompiled to add new possible workflows. However, if we could dynamically add code to a running executable, the iTasks server could remain online during updates.

In this paper, we shall focus on the runtime safety aims required for plugin and workflow systems: we assume that the plugin is trusted (which, in these contexts, can be verified using other techniques, such as digital signatures), so do not need to protect against reading sensitive data and the like, but still need to prevent it from crashing the host program.

In many of these contexts, we furthermore want the "added code" or the "plugin" to be able to be distributed in a platform-independent manner: there are no different JavaScript versions for different binary platforms, nor would one expect to have to download a different compiler plugin on Windows than on Linux. This complicates the matter, because it means we cannot simply distribute machine code which can be linked dynamically.

Lastly, in a lazy functional programming language, we expect that the interface between the host program and the added code is lazy as well. For instance, it should in principle be possible to have

a plugin in some system which computes an infinite list of integers, as long as the host only requests a finite amount of them.

1.1 The solution in a nutshell

Pure functional programming languages are at least theoretically ideally suited for the distribution of unknown code: because functions are first-class citizens, the distribution of code is essentially a special case of serialization. Hence we do not need to treat functions separately (as, for instance, Java does with its remote method invocation interface). Instead, we can design a serialization library which automatically deserializes functions as expected.

Unfortunately, so far, implementers of serialization libraries have taken a traditional, "data-only" approach to serialization. This can be seen in the many JSON libraries that are available for virtually every programming language. In a lazy language, serializing a value to JSON can lead to an amount of work disproportionate to the size of the value in the heap, because the value must be evaluated first. Moreover, shared pointers are lost when serializing to such flat formats, and serialization may not even finish when cyclic or infinite data structures are encountered. Another approach to serialization is to make a copy of the graph without evaluating it [8]. However, because this graph may contain references to the machine code that is needed to further evaluate it, it can only be deserialized by the exact same executable, or, when using symbolic addresses, executables that already contain the relevant code [12, 20]. These solutions, while useful in some distributed systems, are therefore insufficient for the use cases described above.

We describe a new solution. To the outside, it looks like a serialization library, with built-in support for functions. Internally, it builds on existing functionality to copy graphs [20]. However, in addition to the string representation of the graph, serialized expressions now also contain a bytecode which can be used to evaluate it. The serialization library includes an interpreter for this bytecode. This way we combine the compiled code of the native host program with the interpreted code of deserialized expressions.

Our library has the following properties. First, it is cross-platform:[1] the serialized representation of a value is independent of the execution platform, and can be deserialized anywhere else, including by executables that do not contain the source code needed to evaluate them. Second, after deserialization, it is possible to handle serialized expressions in the same way as native values, i.e., without constantly using special functions to access them. Third, deserialization of functions happens in a sandbox and cannot crash the host program. As will become apparent, the second and third requirements exclude each other. Both are implemented, but the programmer must choose whether seamless integration or sandboxing is used.

Lastly, we should note that although our library requires low-level access to the Clean heap to serialize graphs and copy nodes between the native heap and the interpreter heap, it was not necessary to change Clean's native runtime system. We therefore believe our implementation can easily be ported to other contexts.

[1]The interpreter must be available on the target platform. Since it is written in C and only ca. 100 lines of assembly code are needed to connect native Clean with the interpreter, new platforms are easily supported.

1.2 Web applications

We have also successfully applied our interpreter in the context of the web. A main issue here is that the frontend and the backend of a web application are usually written in different languages. The frontend has to be executed in the browser, and is therefore limited to very few languages (primarily JavaScript). For programmers it is however much more convenient to write the entire application in one language. For this reason there exist many frameworks that let one compile server-side languages to JavaScript.

However, compiling functional programming languages for the browser is difficult, because they typically rely heavily on features like deep recursion and non-local goto's that cannot easily be mimicked in browser languages. These issues can be worked around, for example by using continuation-passing style ("trampolined code") and simulating a heap, which is done when Clean is compiled to JavaScript through Sapl [10] or when Haskell is compiled to WebAssembly by Asterius [16]. However, each trick that is used to work around these issues creates a bigger gap between the native runtime system and that used in the browser. Hence, it becomes more difficult to use the graph copying mechanism for communication between the server and the client, and performance becomes more and more unpredictable compared to that of native code.

Therefore we take a different approach here. We cross-compile the interpreter used in the serialization library to WebAssembly, so that it can be used to interpret functional programs in the browser. Interworking is then established between the server, which runs the native version of the executable, and the client, which interprets the same functional language and interacts with the web page to create an interactive web application.

1.3 Organisation

The next section describes the ABC machine, the abstract machine that our interpreter simulates. Section 3 describes the (de)serialization library and the interworking with the compiled host program. In section 4 we describe how the same interpreter is used in the browser. Section 5 provides benchmarks for various implementations of the ABC interpreter. We finish by describing related work and summarizing our conclusions in sections 6 and 7.

2 THE ABC MACHINE

The ABC machine [19, pp. 35–56] is an abstract imperative stack machine designed for the execution of lazy functional programs. The Clean compiler generates ABC code, from which actual machine code is generated. Clean's graph rewriting semantics [26, §1.1] are directly mirrored in the architecture of the ABC machine in that it has a heap which contains the graph that is being rewritten. Additionally it uses three stacks, which contain return addresses, basic values (e.g. integers), and references to nodes in the graph.

2.1 Compilation of Clean

Each Clean function is compiled to a sequence of ABC instructions with a number of entry points. Which entry point is used depends on the context in which the function is called. The most basic entry point is the *strict* entry: this entry point is used when the function result is guaranteed to be needed; the arguments are passed on the stack, and the strict arguments have been evaluated. Other

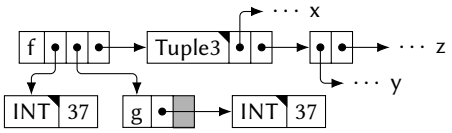


Figure 1: The runtime memory layout of a Clean program.

entry points are used to facilitate currying and lazy evaluation. When a function application is not guaranteed to be needed for the outcome of the program, a thunk is created which contains the arguments and points to the *lazy* entry of the function. When the thunk needs to be evaluated, the program jumps to this entry, which moves the arguments from the node to the stack, evaluates the strict arguments, and proceeds to the strict entry. Afterwards the original node is filled with the result. When a closure is applied to a new argument and this satisfies the arity requirement for evaluation of that node, the *apply* entry is used. This situation is similar to that for the lazy entry, but in this case one or more arguments are already on the stack and a new node is created for the result.

2.2 Concrete implementations

There are several code generators which generate actual machine code for various common instruction sets from the intermediate ABC code. To these target platforms, we have added an ABC interpreter which can execute a bytecode very close to ABC code. The main differences with actual ABC code are a number of straightforward optimizations and the addition of specialized instructions for frequent instruction sequences. This is needed because the ABC language is primarily designed to be an easy compilation target rather than an efficiently interpretable bytecode.

The implementation of the instructions in the interpreter is generated from a small DSL embedded in Clean, with views to produce both C and WebAssembly versions. The supporting code consists of standard bootstrap code (to parse and load the program and print the result) and a copying garbage collector. Only standard techniques were used to implement the interpreter; see [13] for an overview. The C interpreter uses direct threaded code when the compiler supports it (as is the case for GCC and Clang, but not MSVC). The WebAssembly interpreter uses a giant switch approach, which is also the fallback option for the C interpreter.

The runtime layout of the heap is the same whether a program is run natively or in the ABC interpreter. The heap contains two types of nodes: thunks and head normal forms. For both types, the first node points to a *descriptor* containing information about the object, such as the arity and the different entry points. A thunk then contains a contiguous block of pointers to its arguments. A head normal form has the same structure, but is split over two memory blocks if it has more than two arguments. This is exemplified in Figure 1, which shows the memory layout for the expression f 37 (g 37) (x, y, z): it is a thunk with three arguments, and (g 37) a thunk with one argument and a reserved spot. The INT descriptor is for head normal forms (indicated with the triangle in the upper right corner) with one unboxed argument; Tuple3 is a descriptor for head normal forms with three (boxed) arguments. By reserving three words for all thunks (as seen for g) and splitting large head normal form nodes up (as seen for Tuple3), a thunk can always be overwritten by a head normal form (as is done in the lazy entry point).

[1]Here and elsewhere we use the term "closure" for unsaturated function applications. Internally these are represented as head normal form nodes. The head refers to the function that is to be applied when all arguments have been "curried" into it; the node arguments are the (possibly zero) arguments that have already been bound.

3 (DE)SERIALIZATION AND INTERWORKING

Given the memory layout described above, it is easy to see how graphs can be serialized: the graph is traversed in a depth-first manner, marking visited nodes so that shared pointers remain shared after serialization. This technique is commonly referred to as *graph packing* [8, p. 41 and references therein]. It is also implemented for Clean under the name *GraphCopy* [20], with a wrapper, *GraphCopy-with-names*, that translates descriptor addresses to descriptor names. This wrapper can be used to work around the well-known issue that graphs serialized this way can normally only be deserialized by the same executable, because they contain hard-coded addresses (e.g. [12, p. 126]), but does not allow one to extend program functionality yet: all code required to evaluate the expression must be present in the deserializing executable.

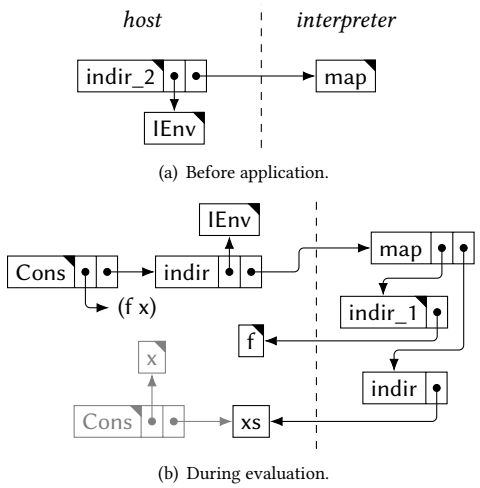
The new serialization library is essentially a wrapper around GraphCopy-with-names. A serialized expression now contains the graph, the descriptor names, and the bytecode. The necessary bytecode is extracted using reachability analysis starting from the descriptors that are used in the graph.

To deserialize an expression using the ABC interpreter, the following steps are performed. First, new heap and stack spaces are allocated, and the bytecode is parsed and loaded into memory. The symbol table of the bytecode program is matched against the symbol table of the executable, to facilitate interworking later on (see section 3.2). Second, the descriptor names in the serialized expression are replaced with the actual addresses of the instantiated bytecode program. After this, the graph can be unpacked with the straightforward inverse operation of the graph packing algorithm described above. The interpreter can now start to evaluate the graph.

3.1 Interworking with compiled code

To make sure deserialization is lazy, the interpreter does not evaluate the whole graph to a normal form at once. Instead, it evaluates it to head normal form. This head normal form is then copied to the native Clean program, called the host.

Two representative stages of this process are shown in Figure 2, where the well-known function map is interpreted and applied to two native arguments. The native heap is shown on the left of these figures; the interpreter's heap on the right. Figure 2(a) shows the initial state in which a map closure has been created in the interpreter and an indirection has been made in the host. Figure 2(b) reflects the state after the function is applied to some native operator f and a native list (Cons x xs), and after the first Cons node of the result has been copied to the native heap. This is the top left Cons node; the other Cons node is the head of the original list (which can now be garbage collected if the host has no other references to it). The first result element (f x) is not shown for brevity. The tail of the resulting list is a reference to a map thunk in the interpreter. The IEnv nodes are required arguments for



(a) Before application.

(b) During evaluation.

Figure 2: Snapshots of the heaps when interpreting map.

indirections from the host to the interpreter; this will be discussed below. We now describe each case of the copying process in detail.

Node arguments that are unboxed values or are in head normal form (and are not closures) are copied directly. This could for example be the case for the first argument of the result Cons node in Figure 2(b) (the evaluation of f x), if we are dealing with a head-strict list. Although theoretically this can lead to large values being copied even though they are not needed in the host (because they just happen to have been evaluated in the interpreter), this did not appear to be a problem in practice. We expect that creating indirections for nodes that are already in head normal form would lead to additional overhead, so we have decided to copy these nodes directly instead. However, we currently do not have data to confirm or reject this hypothesis.

It is in principle possible to also copy thunks for which the relevant code is present in the host program (for example, the first argument of the result Cons node in case of a lazy list). However, this would impose a security risk: if this lazy value causes a runtime fault, this must be caught by the interpreter, so the lazy value cannot be copied to the host.[2] Instead, an indirection thunk is created in the host for every thunk that is encountered, as can be seen in the indir node on the host side in Figure 2(b). This indirection node references the node in the interpreter heap. When evaluated, the interpreter is used to evaluate the node (with the *lazy* entry point), after which a new copy cycle begins.

During the copying of a node, visited nodes are marked, so that cyclical data structures can be copied without problems and shared pointers remain intact. However, this matching between interpreter and host nodes is not retained after the copy cycle has finished (this would require a significant amount of extra work to keep the matching up to date during garbage collection).

So far we have covered situations in which a serialized expression copied to the host may not be fully evaluated but will still eventually be rewritten to a graph that solely consists of data. It is however also possible that the interpreted expression contains closures. These can then be applied to native arguments. As with thunks, we cannot copy these to the host because of security risks and because the relevant evaluation code may not be present in the host. Instead, also in this case an indirection is created in the host. This is the case shown in Figure 2(a). This indirection is a closure which still requires as many arguments as the underlying interpreter node; hence, indir_2 is used here instead of indir. The arguments are therefore collected in the native Clean program; only when enough arguments are present they are copied to the interpreter and is the function evaluated. This is done using the *apply* entry of the node.

In the case of closures in deserialized expressions it is therefore necessary to copy values from the host to the interpreter. This works in much the same way as in the other direction. Hence the interpreter heap may also contain indirections to the host heap, as is seen in Figure 2(b). The indir node here references the tail of the original list to which map is applied; it is a thunk, because xs has not been evaluated in the host yet. The indir_1 node refers to the operator argument; it is a closure which still requires one argument, mirroring the node in the host.

As can be seen, the relationship is much more symmetrical than the term "host" suggests, and indirections in both directions are required to facilitate laziness. One important difference is, however, that there may be several interpreters while there is always only one host. Hence, indirections from the host to the interpreter have an extra argument, the *interpretation environment* (IEnv in Figure 2), which contains such information as the locations of the bytecode, heap and stack of the interpreter. These are needed to start the interpreter to evaluate the referenced node. Indirections from the interpreter to the host do not need this extra argument.

It is possible that the host applies an interpreted function to a value that is deserialized using another interpreter. This can become quite complex, if there are multiple interpreters and higher order functions involved. This case is automatically caught by the setup presented here, but not very efficiently: indirections from one interpreter to another will in fact be indirections to a host indirection to that other interpreter. This can in principle be optimized by allowing interpreter indirections in the interpreter itself.

3.2 Translation of descriptors

The host program and the interpreter do not share their descriptors, because the descriptor layout is different for different binary platforms and contains references to program code.[3] The consequence is that descriptor addresses must be translated when copying a node. To do this efficiently, the descriptor blocks in the interpreter are expanded with an extra field which is a pointer to the corresponding descriptor in the host. Hence, translation of the descriptor from the interpreter to the host can be done in constant time.

Because a host descriptor may have to be mapped to several interpreter descriptors (when multiple expressions are being deserialized simultaneously), we cannot translate host descriptors to interpreter descriptors in the same way. Instead, each interpreter keeps a separate binary search tree for this purpose.

[2] For the same reason, interpreter instances do not share common libraries with each other or with the host.

[3] To be precise, descriptors contain pointers to entry points to evaluate the node or add arguments to a closure. Depending on the type of node other information can be included as well, e.g. the arity (for garbage collection), the name (of constructors, for printing), the types of unboxed values (for printing), etc.

It is possible that a descriptor of the interpreter does not exist in the host or vice versa. This can for example occur when the interpreted expression uses types unknown to the host or when the interpreted expression is a polymorphic function and is called with a native argument of a type unknown to the interpreter. In this case we allocate a new block of memory and copy the descriptor. This is needed because the garbage collector uses information from the descriptors. Since only fully saturated head normal forms are copied, we do not need all elements from the copied descriptor. For instance, descriptors contain a curry table which is used to curry new arguments into closures. This curry table is not copied, because it will never be used. Effectively this means that we only need to copy data in a descriptor, not references to code.

3.3 Impact on garbage collection

As described above, both the host heap and the interpreter heap(s) contain indirections to each other. When garbage collection runs, these indirections must be taken into account so that, for example, interpreter nodes that are referenced from the host (but not from the interpreter heap itself) are kept in memory. Furthermore, references in both directions must be updated when a garbage collector moves objects. As has been mentioned above, the interworking setup is much more symmetric than one may expect, and hence ideas to solve this problem were taken from implementations of distributed systems [18, pp. 102–111]. Nevertheless we are able to present a solution that relies on only one special feature of the native runtime system. This feature can be emulated in other runtime systems with more common features (see note 4).

To retain host nodes referenced from the interpreter, the interpretation environment (in the host) contains an array of nodes shared with the interpreter. Because this array lives in the host, the references are updated by the host garbage collector — and because the interpretation environment is referenced from every indirection from the host to the interpreter, the array is kept alive as long as necessary. An indirection from the interpreter to the host is simply an index into this array and therefore does not need to be updated after garbage collection has taken place in the host. When garbage collection runs in the interpreter, the array is cleaned up.

To retain interpreter nodes referenced from the host, all references from the host to the interpreter are kept in a weakly linked list in the host. The Clean garbage collector ensures that values in the linked list that are not reachable from elsewhere in the heap are removed from the list.[4] The interpreter's garbage collector makes sure that objects referenced from the weakly linked list are kept in memory and have their references updated.

Finally, the interpretation environment is wrapped in a finalizer, so that a C function is called when it is removed by the garbage collector. This C function then frees all related memory (program code, heap, stack, and descriptor matching); only descriptors that have been copied to the host remain (because they may still be in use in the host).

[4]This is the only uncommon feature of the Clean runtime system that we use. A similar solution can be implemented in runtime systems with hooks at strategic places in the garbage collector, to ensure that the references from the weakly linked list are not considered to determine whether an object can be freed, but *are* updated when objects are moved. This solution can also be emulated in runtime systems with support for finalizers (the ability to execute code after a referenced object has been removed from memory), but with a larger memory footprint.

3.4 Sandboxed deserialization

While it is possible for the ABC interpreter to implement the ABC instructions to deal with file I/O, pointers, and the C foreign function interface, this has not been done so far. The reason behind this is that interpreted programs should in principle not have side effects, because deserialization is a pure function. Should a host program wish to offer interpreted programs access to such features, it can do so through an ad hoc interface.

However, interpreted programs can still crash, for example by running out of heap or stack space or dividing by zero. In some contexts it is required that expressions are interpreted in such a way that it cannot crash the host program. This is for example the case in plugin systems where the plugins would be interpreted. However, this turns out to be tedious in a lazy language that is furthermore strongly typed and must therefore reflect "exceptions" (like when interpretation failed) in types.

3.4.1 Safety and laziness. Where the type of our deserialization function was previously simply (String → a),[5] it would now have to be (String → MaybeDeserialized a), with MaybeDeserialized recognizing different kinds of exceptions:

```
:: MaybeDeserialized a
   = HeapFull | StackOverflow | IllegalInstruction // various exceptions
   | Ok a                                          // success
```

However, it cannot be decided after the first interpretation step (i.e., when a head normal form has been reached) which constructor must be created. If no fault was detected, we still cannot create an (Ok *val*), because *val* may contain indirections to the interpreter, which, when evaluated, may cause a fault.

For evaluations further down the evaluation tree we even cannot wrap the result in a type like MaybeDeserialized. Suppose we are deserializing a value of type List Int:

```
:: List a = Cons a (List a) | Nil
```

Because both arguments of the Cons constructor may trigger a fault, a naive deserializer would now generate values like (Ok (Cons (Ok 1) (Ok Nil))), which are ill-typed. Indeed, faults further down the evaluation tree *must* be reflected in a top-level wrapper type. This requires fully evaluating the expression to a normal form to decide which constructor must be used. We have nevertheless implemented this functionality; the programmer can now choose between using a lazy deserialization variant which may crash the host program, and a hyperstrict variant which catches faults. Our library therefore provides two functions: deserialize :: String → a and safe_deserialize :: String → MaybeDeserialized a.

With GHC's exceptions (which can be thrown in pure expressions and caught in the IO monad) it would not be necessary to reflect possible failure in the type, and it may be possible to combine safety and laziness.

3.4.2 Function types. There are still situations in which the normal form that is the result of the full evaluation of the serialized value can cause more interpretation steps in the future. This is the case

[5]In reality, the type is a little different, because the function needs access to the executable's symbol table to be able to translate descriptors. The function also has an extra argument for a settings record with which the programmer can for example set the heap size given to the interpreter. These additional arguments are orthogonal to the discussion here and are omitted for clarity.

when the normal form contains closures. If the host curries more arguments into these nodes, this may trigger a new interpretation step. In this context, we do not explicitly cause the interpretation step, so the programmer has no way to choose between deserialize and safe_deserialize.

The only sensible choice here is to assume that such evaluation steps must be performed in a safe context as well. Imagine a plugin system where plugins are records of a number of functions and metadata. The initial deserialization step only evaluates the metadata but cannot apply the functions yet, while actually these functions are what must be sandboxed. When the library creates an unsaturated indirection from the host to the interpreter as the result of a safe interpretation, this indirection is therefore also safe. This has the unfortunate consequence that the type of the interpreted value changes. If the serialized value was of type (Int, a → b), the deserialized value will now be of type (MaybeDeserialized (Int, a → MaybeDeserialized b)). Unfortunately, this means that programs using sandboxed deserialization may need unsafe type casts to handle these values later on.

3.4.3 Notes on the implementation. Some faults, like illegal or prohibited instructions, are caught quite easily as an additional instruction which exits the interpreter loop. To check for stack overflows, a segmentation fault handler is installed, and the C functions (sig)setjmp/(sig)longjmp to save and restore application state are used to jump back to the start of interpretation after a fault. The stack pointers are then reset to their original value, but the heap pointer remains untouched, because part of the original heap may now have references to nodes added in the last interpretation step.

Because an interpretation may trigger evaluation in the host (when the host has shared unevaluated function arguments), which may in turn trigger new interpretations, there can be a theoretically endless chain of interpretation steps. This happens, for example, when applying a lazy, interpreted variant of foldr to native arguments, where the interpreted foldr and the native operator argument become intertwined on the stack. It is important to distinguish the different interpretation steps when resetting the application state in case of a fault. Consider the following example:

```
x = deserialize "f" (g (deserialize "bad"))
```

Assume that (deserialize "f") yields a function in the interpreter, that g is a normal native function, and that (deserialize "bad") results in a fault in the interpreter. If g handles the fault (for example, by plugging in a default value in case a fault has occurred), x must evaluate to a "good" deserialized value. This must be distinguished from the case that x = deserialize "bad". Hence, the chain of interpretation steps must be reflected in the data structure that is used to restore the application state. We therefore use a stack of restore points so that each interpretation step can be recovered from.

However, we also cannot create a restore point for each individual interpretation step, but only for those which were requested to catch faults by the programmer (i.e., those that were created for safe_deserialize, and closures in the results of such safely deserialized values). If nodes are copied from the interpreter to the host as the argument of a (host) function, these must *not* be wrapped in the MaybeDeserialized type, and hence we must not create a restore point for these values. Should evaluation of these nodes fail, this will cause the interpretation that required application of that host function to fail instead, as expected. The purity of the host function that is called in combination with the strictness of the safe deserialization step guarantees that the unsafe indirection created this way cannot appear elsewhere after the overarching safe interpretation step has finished. In our implementation, we use one bit in each indirection from the host to the interpreter to signal whether lazy and seamless or strict and safe interpretation is to be used for a particular node.

3.5 Type-safe deserialization

Even with the sandbox environment provided by separate heap and stack spaces and the segmentation fault handler described in the previous subsection, it turns out to be relatively easy for *malicious* interpreted expressions to crash the host program. Serialized expressions contain no information about their type. The type of the deserialization function is simply (String → a) (or String → MaybeDeserialized a), so the context in which the deserialization function is called determines a, the type that the interpreted expression is assumed to have. Since this information is not available at runtime, it cannot be checked that the expected type matches that of the interpreted value when it is copied into the host heap. In other words, the interpreter may copy an integer into the host heap, and the host may interpret it as a pointer, etc.

Another issue is that descriptors are matched by looking at symbol names only. Therefore, if the host and the interpreter contain a constructor with the same name (and from the same module), these are matched regardless of whether their arity or the types of their arguments match. This is not only a security concern: the same situation can occur when the host and the interpreted program share source code from some common library, but have used different versions of that library between which types have been changed.

The latter issue can be solved relatively easily by storing more type information in descriptors. Currently, descriptors contain such information as their arity and name (to print them), and for nodes with unboxed values a type string to identify the types of the arguments. However, this type string only distinguishes the various basic values (integers, booleans, etc.) and node values, which are boxed; there is no distinction between nodes of different types. If information about the types of node arguments were added to descriptors, runtime checks could be added to verify type coherence.

This does not solve the first issue yet, since it would still be possible, for example, to deserialize an integer and use it as a pointer. To resolve this issue we can use Clean dynamic types on top of our serialization library. With dynamic types, almost any[6] Clean expression can be packed into an opaque type Dynamic, which internally consists of the expression and a representation of its type [1, 2, 23]. Dynamics can be unpacked using a custom pattern match against types, which internally uses a unification algorithm to match the expected type against the actual one.

We could thus write a deserialization function with type (String → MaybeDeserialized Dynamic). For the reasons outlined in section 3.4.1, it does not make sense to combine this approach with lazy, non-sandboxed deserialization. When the expression is in normal form, it must be checked that (1) the generated graph is well-typed

[6] Some values, such as files, cannot be packed into dynamics. However, these are typically also the kind of values for which (de)serialization does not make sense.

(i.e., conforms to the type specifications of the referenced descriptors), (2) the types of the used descriptors match the types of the descriptors in the host, and (3) the type of the root node is Dynamic. If this is the case, the value can be copied safely to the host.

Unfortunately, the types of closures will still change in this setup, in the same way as described in section 3.4.2. A way around this would be to not use MaybeDeserialized, but instead define a new type DeserializationException:

```
:: DeserializationException
   = HeapFull | StackOverflow | IllegalInstruction
```

The type of the deserialization function now becomes (String → Dynamic). The dynamic contains either a value of type DeserializationException, or a value of any other type which has been correctly deserialized. If the types of the closures are chosen well and all return a Dynamic, we no longer need to change these types, so that unsafe type casts can be avoided in the host program. This advantage outweighs the obvious drawback that in this setup there is no way to distinguish a faulty interpreted expression from an interpreted expression that correctly evaluates to a value of type DeserializationException.

4 INTERWORKING ON THE WEB

The previous section remarked at several stages how symmetrical the interworking between a native Clean program and our interpreter is: it is, essentially, a specific type of distributed system. Recognizing this, it becomes easy to see that we can also apply a similar technique in web applications, by plugging in a communication channel to transmit values back and forth. We use the WebAssembly version of the interpreter currently in *iTasks*, a task oriented programming framework for developing web applications [24, 25].

Commonly, web applications use at least two different programming languages: one on the server, and one on the client. This is because only few languages can be used in web browsers: most notably JavaScript and WebAssembly [14]. When the server is written in another language, the developers need to make sure that the client-server interface is correct: when one side changes, the other must as well. Even with standards like JSON this remains tedious.

In iTasks, this problem was solved by Sapl [10]: a minimal functional programming language that can be targeted by the Clean compiler and compiled to JavaScript. This allowed part of the iTasks application to be compiled to JavaScript and run in the browser. Communication was done using generic functions (targeting JSON, although this is an implementation detail). However, there were several issues with this approach. First, due to the way Clean was compiled to Sapl, not all Clean values could be sent to the browser: they had to be evaluated to a normal form first. Second, because Sapl branched off relatively early in the compiler toolchain, it was time-consuming to maintain. Third, the compiled JavaScript had a bad worst-case performance, in part due to tricks required to work around the low stack recursion limit in JavaScript engines. This led programmers to write two versions of some functions: one optimized for native Clean, and one to be run in JavaScript in the browser. This is however at odds with the overall goal of the system: to write the entire application in one source language.

To overcome these issues, the latest version of the iTasks framework now uses our new ABC interpreter in the browser. Because the ABC language is on a much lower level than Sapl, this simplified the compiler toolchain significantly. More importantly, the memory model in the interpreter is the same as that of native Clean, so it is now straightforward to copy any value from the server to the client (using the GraphCopy module described earlier). For the same reason, the performance penalty has become much more predictable and there is no need for different implementations of the same function any more.

4.1 Interworking with the iTasks server

Communication between the iTasks server and the interpreted Clean program running in the browser can be set up in several ways. The iTasks framework already provides a web socket with which messages can be passed efficiently without polling the server. It would hence be possible to apply the same method of interworking as in the deserialization setting here, with the difference that nodes would be copied over the network rather than in memory. However, this would lead to a lot of network traffic due to lazy evaluation of the result. Furthermore, we are usually not interested in the actual result of the function that is sent to the client. For instance, it may only be used to interface with JavaScript to set up part of the user interface, such as using a third-party JavaScript library to add a date picker to an input field.

For this reason, the iTasks framework currently only allows sending a limited set of functions to the client: functions of type (JSVal *JSWorld → *JSWorld).[7] Such a function is executed when an iTasks component is instantiated. The *JSWorld is an opaque type, realized internally as a boxed integer but inaccessible to the programmer. Because all functions in the JavaScript foreign function interface require a value of *JSWorld, and uniqueness typing ensures that this value cannot be duplicated, the programmer can enforce the execution order. The JSVal contains a reference to the underlying iTasks component, which is a JavaScript object. This value can be used to set up the initial state of the component, including installing Clean functions as callbacks for JavaScript events.

In some cases, initialisation of a form field is all that needs to be done. For instance, when a date field is created, we call a third-party JavaScript library to initialise a date picker, but the changes to the value of the input field are communicated with the server through the core iTasks framework. Other cases are more complex, and do require that we can send a message to the server from within the WebAssembly interpreter. For instance, in interactive SVG images [4], both a model and a view of the image is kept. When an event is handled, these can be changed. Some changes can be handled entirely locally, while in other cases the changes must be synchronised with the server. To this end, the iTasks component has a doEditEvent property: a JavaScript function which is used to send events to the server. It would be possible to set up lazy communication in this case: we could use the GraphCopy functionality to copy the graph from the WebAssembly interpreter heap to the server. However, this imposes a security risk, since the doEditEvent method can be called by the user of the web application as well,

[7] Here, the asterisk in *JSWorld indicates that the type is *unique*: the value may only be used once [6] (cf. *linear types* for Haskell [7]). This is the way file I/O and destructive updates are handled in Clean: through unique types like *File and unique arrays [26, §9]. Also, recall that Clean function types can have an arity larger than 1. The Clean type (a b → c) corresponds to the Haskell type (a → b → c).

who can then evaluate arbitrary code on the server. For this reason, we currently use JSON encoding to transmit edit events, and hence require the values to be fully evaluated.

Communication from the server to the client (besides the initial initialisation step) is also handled through common iTasks functionality. In this case, the server sends an *attribute change* to a specific iTasks component. The component can have a listener installed on attribute changes, which is called when this happens. In the initialisation function of the component, the programmer may thus set up a Clean function as an attribute change listener. Because attribute values are simply strings, the programmer is free to use any encoding. This includes serializing a possibly lazy Clean value on the server and sending the serialized value to the client. A dedicated function in the JavaScript interface, `jsDeserializeGraph`, can be used to deserialize this value into the WebAssembly heap.

4.2 Interworking with JavaScript and the DOM

While the new client runtime integrates much more neatly with the server, our approach also has some downsides compared to the previous system. One of the main goals of the integration on the browser is to make the application interactive. For this, we need access to the Document Object Model (DOM) of the webpage to create and remove elements or change their attributes.

In the Sapl setup, Clean programs could contain a dedicated set of functions which had a hard-coded implementation in JavaScript and thus allowed calling arbitrary JavaScript functions and handling DOM elements. This was interwoven with the JavaScript code generated for the actual Clean program. Hence, there was no concept of a foreign function interface between Clean and JavaScript; in the compiled code, these were identical.

Interaction with the DOM is currently impossible in WebAssembly, so we now need to call JavaScript functions from the WebAssembly interpreter to interact with JavaScript objects and the DOM. Therefore, the current system does include a foreign function interface, with a small number of traps: dedicated ABC instructions which bail out to JavaScript to evaluate JavaScript expressions. Because we do not know in general which functions can be called, this interface must be general: it is essentially a wrapper around JavaScript's `Function` constructor. Given a string, this constructor parses it as a JavaScript function body, and returns the corresponding function. Because this is a client-side operation and the string is controlled by the iTasks application, this does not introduce security issues. The result of the evaluation of this function is then passed back to Clean. The string that is sent to JavaScript is immediately freed from the Clean heap to reduce heap usage. However, the string must still be built and parsed, so the interface with JavaScript has become slower in the new setup.

4.3 Impact on garbage collection

Not all JavaScript values can be copied to Clean. When a JavaScript evaluation results in an object or function, it is stored in a common array on the JavaScript side and Clean receives a reference to that array instead. This way, a Clean program can still create JavaScript objects and call functions on them. It also turns out to be useful to be able to store an arbitrary Clean value on the JavaScript side. For instance, a Clean function that is set as a callback for a JavaScript event may need to access Clean values that are modified from elsewhere. These references are also stored in an array on the JavaScript side.

To clean the array of shared JavaScript values up, the Clean garbage collector keeps track of the references it finds to that array. After a garbage collection cycle, the array is cleaned up. This is analogous to cleaning shared host nodes in the deserialization setting.

The other direction is trickier, because we do not have access to JavaScript's garbage collector. Hence there is no way to determine whether a Clean value referenced from JavaScript must be kept in memory or not. There is a proposal under consideration for inclusion in the JavaScript standard (properly termed ECMAScript) to add finalizers to the language [29]. However, this new feature will be of little use here (or elsewhere), because finalizers will not be *guaranteed* to run after an object is garbage-collected.

We must therefore implement our own small garbage collector on the JavaScript side as well. We do this by explicitly linking every Clean value shared with JavaScript explicitly to an iTasks component. This component can be as small as a single text field or can be a larger collection of different views. As opposed to arbitrary JavaScript values and DOM nodes, iTasks components do have proper finalizers (provided by the iTasks framework). The components now keep track of the linked Clean values, and remove them from the shared array when they are destroyed.

To this basic scheme we add one optimisation, which is useful when components live so long that we cannot wait for them to be destroyed: when the Clean program sets a JavaScript variable to some value, it is first checked whether the variable contained a reference to Clean. If this is the case, the reference is freed directly. This is useful in situations where the same variable is continuously updated with new Clean values.

However, in some cases, even this is not enough. In a library used to create interactive SVG images [4], for every update of the SVG image new Clean functions could be attached to DOM nodes as event callbacks. Because these are not overwritten with a new value but simply disappear from the DOM, the previously described optimisation does not target this case. Currently, the only solution in this case is to keep track of the installed callbacks in the SVG library itself and remove them when the corresponding object has been removed.

A current proposal to add garbage collector integration to WebAssembly [27] may allow us to implement parts of the JavaScript interface in a neater way, but it is not yet advanced enough to determine whether it will be helpful here or not.

4.4 The JavaScript foreign function interface

To illustrate the above, this section describes the foreign function interface that can be used to access the JavaScript world from Clean. Other than the interworking described in section 3, this is a proper foreign function interface where copying of values has to be explicit. This is necessary because the execution models of JavaScript and Clean are too different.

A representative excerpt of the interface is given in Figure 3. The server-side iTasks application contains code using this foreign function interface, which is sent to the browser as described in

```
:: *JSWorld
:: JSVal
:: JSFun :== JSVal

(.?) infixl 1 !JSVal !*JSWorld → (!JSVal, !*JSWorld)

jsValToInt    :: !JSVal → Maybe Int // and similar for other types
jsIsUndefined :: !JSVal → Bool // check for JS value 'undefined'

class (.#) infixl 3 attr :: !JSVal !attr → JSVal
instance .# String, Int

generic gToJS a :: !a → JSVal
derive gToJS Int, Bool, String, JSVal // etc.

(.=) infixl 1 !JSVal !a !*JSWorld → *JSWorld | gToJS{|*|} a

class toJSArgs a :: !a → {!JSVal}
instance toJSArgs Int, Bool, String, JSVal, () // etc.
instance toJSArgs (a,b) | gToJS{|*|} a & gToJS{|*|} b // and for (a,b,c), etc.

(.$)  infixl 2 :: !JSFun !a !*JSWorld → (!JSVal, !*JSWorld) | toJSArgs a
(.$!) infixl 2 :: !JSFun !a !*JSWorld →           *JSWorld  | toJSArgs a

jsGlobal :: !String → JSVal
jsNew :: !String !a !*JSWorld → (!JSVal, !*JSWorld) | toJSArgs a
addJSFromURL :: !String !(Maybe JSFun) !*JSWorld → *JSWorld

jsMakeCleanReference :: a !JSVal !*JSWorld → (!JSVal,   !*JSWorld)
jsGetCleanReference  ::   !JSVal !*JSWorld → (!Maybe a, !*JSWorld)
jsFreeCleanReference ::   !JSVal !*JSWorld →             *JSWorld

jsWrapFun :: !({!JSVal} *JSWorld → *JSWorld) !JSVal !*JSWorld
    → (!JSFun, !*JSWorld)

jsDeserializeGraph :: !String !*JSWorld → (!a, !*JSWorld)
```

Figure 3: The JavaScript foreign function interface.

section 4.1. The programmer is responsible for making sure this interface is only used on the client-side, as it uses trap instructions which are not defined in the native Clean runtime system. This is trivial, because the JSWorld can only be obtained on the client-side since it is provided by the supporting JavaScript code.

As can be seen, JSVal is an opaque type. Internally, it is an algebraic data type representing different kinds of JavaScript expressions, including references to the arrays of shared JavaScript and Clean values. The JSFun synonym type is merely provided to improve readability in function types.

To evaluate a Clean expression, the .? operator can be used. Note that the use of *JSWorld here and elsewhere enforces evaluation order, as has been mentioned above. Because not all JavaScript values can be copied to Clean, the result of this function is still a JSVal. Other functions, like jsValToInt and jsIsUndefined, can be used to move the result into the Clean world.

The .# class is used to get properties of JavaScript objects. For instance, obj .# "key" corresponds to the JavaScript expression obj["key"], while obj .# 0 corresponds to the expression obj[0].

The generic function gToJS converts Clean values to JavaScript values. This is possible for basic types and records (for which JavaScript objects are created), but not for algebraic data types, which have no direct equivalent in JavaScript. The .= operator sets a certain JavaScript reference to some value using gToJS.

To call JavaScript functions, the .$ and .$! operators can be used, the difference being only that the first returns the result. The type of the arguments of the JavaScript function must instantiate the toJSArgs class. This class is essentially the same as the generic function gToJS, with the exception that it allows the void type () for function applications without arguments, and tuples of convertible types for functions with more than one argument.

To use global JavaScript objects (e.g., window or document), jsGlobal can be used. Similarly one can use jsNew to create new JavaScript objects with the new keyword. It is also possible to load external JavaScript files using addJSFromURL. The second argument to this function is an optional callback, which will be executed after the file has loaded. This is for example used in iTasks to load the external *Pikaday* library for date pickers [9], after which jsNew "Pikaday" (…) is used to initialize the input field.

The functions jsMakeCleanReference and jsGetCleanReference are used to store and retrieve references to Clean values in the JavaScript world. The second argument to jsMakeCleanReference must be an iTasks component. This component is used for garbage collection, as explained above: the shared Clean value is kept in memory as long as the iTasks component exists. Alternatively, one may use jsFreeCleanReference to free the value earlier.

To install Clean functions as callbacks for JavaScript events, jsWrapFun is used. This is nothing more than a wrapper around jsMakeCleanReference, so this function also has a JSVal argument which must be a reference to an iTasks component and is used for garbage collection. Once called, the function arguments are passed as a single array of JSVal values. This is needed because JavaScript functions can be called with any number of arguments.

Finally, jsDeserializeGraph is the WebAssembly version of the graph unpacking function in GraphCopy. It is provided so that the iTasks server can send lazy serialized expressions to the client.

5 BENCHMARKS

We have benchmarked the different implementations of the interpreter and compared its performance against that of native Clean and Sapl. The results are in Table 1, with time relative to native Clean plotted in Figure 4. The benchmarks come from a standard set of small programs,[8] for easy comparison with more minimal runtime systems, and some somewhat larger programs from [15][9] to more accurately estimate performance in real use cases. Additionally, we have benchmarked a real-world program, namely the Cloogle search engine [28].[10]

The runtimes benchmarked are: native Clean; the C interpreter with direct threaded code (compiled with gcc 6.3 and -Ofast -fno-unsafe-math-optimizations); the C interpreter with a giant switch

[8] The benchmarks are implemented as follows: nfib naively computes the 43rd Fibonacci number, adding 1 for each function call; tak computes $\tau(36, 24, 12)$, where τ is the Takeuchi function; lqueen solves the N-queens problem for a 13 × 13 chess board; reverse reverses a list of 30,000 elements 30,000 times; fsieve computes the 10,000th prime number using the sieve of Eratosthenes 200 times.

[9] complab is an image processing program; event simulates 1,500,000 transitions in a flip-flop; ida is an implementation of the Iterative Deepening A* algorithm; sched is a job scheduler; parstof is a parser; transform performs an AST transformation.

[10] The implementation of the benchmark can be found on https://gitlab.science.ru.nl/cloogle/benchmark.

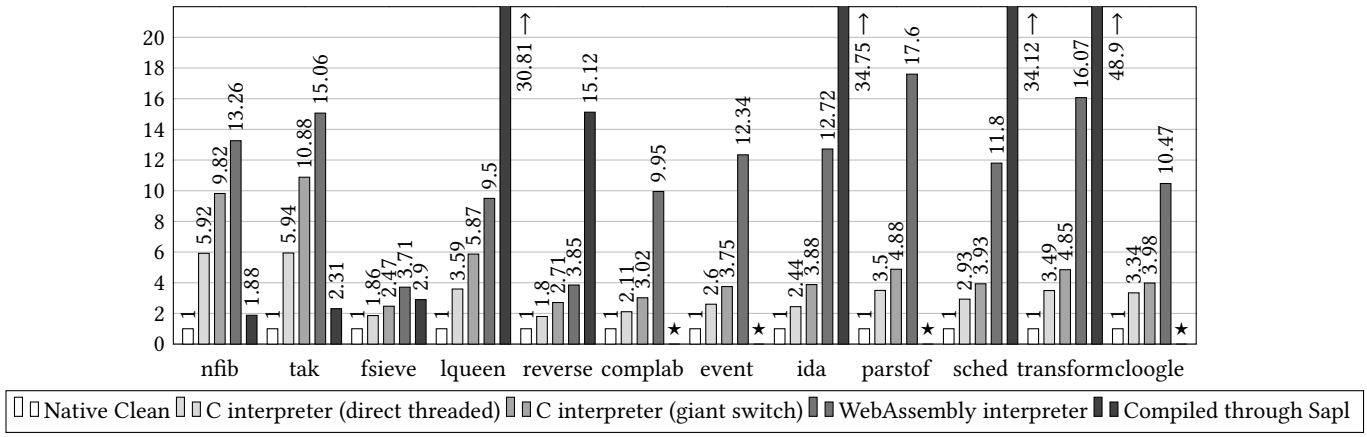


Figure 4: Performance of the ABC interpreter and Sapl, relative to native Clean (lower is better).

Benchmark	Native Clean	C (direct threaded)	C (giant switch)	WebAssembly	Sapl
nfib 43	2.38	14.08	23.36	31.56	4.47
tak 36 24 12	5.77	34.26	62.76	86.87	13.32
lqueen 13	2.26	8.11	13.26	21.47	69.62
reverse 30,000	4.05	7.28	10.96	15.60	61.23
200× fsieve 10,000	1.53	2.85	3.78	5.68	4.43
30× complab	2.44	5.15	7.38	24.27	★
10× event	1.62	4.21	6.08	19.99	★
10× ida	3.13	7.64	12.16	39.81	108.8
1,000× parstof	1.80	6.30	8.79	31.68	★
sched	1.08	3.16	4.24	12.74	36.85
3,000× transform	0.86	3.00	4.17	13.82	42.05
cloogle	181.5	605.4	722.9	1900	★

Table 1: Wall-clock time in seconds for native Clean, the ABC interpreter, and Sapl.

(idem); the WebAssembly interpreter (in SpiderMonkey 68.0); Sapl (idem). The benchmarks were run on an i7-5500U processor at 2.4GHz. Native Clean and the ABC interpreter received 200MB of heap for each program. Because Sapl uses JavaScript expressions to emulate the Clean graph, we cannot easily limit the heap size in this setup. A number of benchmarks failed to run with Sapl, for a variety of reasons (stack depth limits; run-time exceptions; lack of support for unique array selections). The lqueen benchmark was adapted to solve a smaller problem (on a 12×12 chess board); the results were then scaled accordingly. The other failing benchmarks are indicated with a star: "★". Lastly, the reader should keep in mind that the native Clean compiler generates highly performant code.

As can be expected, programs that rely heavily on the heap (such as reverse) have a much smaller performance penalty than those that use more basic values on the stack (such as nfib and tak). This is because the former programs spend more time in garbage collection, which is nearly as fast as in native Clean, and because the latter are overall simpler programs, with smaller instructions and hence a larger dispatch overhead in the interpreter.

On most practical benchmarks for which Sapl does not crash, the WebAssembly interpreter is faster by a factor of around 3. Only simple programs that deal with a lot of basic values (like nfib, tak and fsieve) clearly benefit from the fact that the Sapl code is just-in-time compiled instead of interpreted. However, because JavaScript has a small recursion limit, much too small to implement a functional programming language, Sapl uses continuation passing style (i.e., "trampolined code"), causing an overhead even on these small programs. Sapl's performance costs become much clearer for larger programs which use heap structures. To achieve lazy evaluation, Sapl represents thunks and head normal forms as JavaScript arrays, which are evaluated using a central "trampoline" function. How efficient that is, is difficult to predict, as is reflected in the wide range of performance penalties compared to native Clean, between 15 times slower on the reverse benchmark and 49 times slower on transform. Because the ABC interpreter remains much closer to the native Clean runtime environment, its overhead is much more predictable (taking into account the fact that it is faster when a program requires more garbage collection, as mentioned above).

The difference between the native interpreter (with a giant switch) and the WebAssembly interpreter is higher than expected, since WebAssembly is expected to have near-native performance [14, p. 197]. Looking at different JavaScript engines does not help much:

Google's V8 engine (version 7.3), for instance, is about 30% slower on our interpreter than Mozilla's SpiderMonkey (68.0). Upon inspection of the code generated by the just-in-time compiler, this turns out to be a register allocator issue, due to which SpiderMonkey's register allocator performs particularly bad on large interpreter loops like ours.[11] The Clean program counter and heap and stack pointers are therefore continuously stored and reloaded from the WebAssembly stack. To work around this issue, we currently use global variables instead of locals to keep this state of the interpreter, which causes the variables to be stored in memory rather than on the stack and thus reduces the number of spills needed. This is about 40% faster on our benchmarks (included in the results presented above). However, a proper solution would be to store the pointers in registers without spilling them to the stack. The difference also *varies* more than one would expect — at the moment it is unclear why. Unfortunately, it is difficult to investigate this issue without first fixing the aforementioned problem with the register allocator.

Using Emscripten [32] it would have been possible to compile the C version of the ABC interpreter to WebAssembly, instead of creating a separate WebAssembly version. However, initial tests showed that this approach would be rather slow, this WebAssembly interpreter being around seven times slower than the C version, i.e., between 10 and 70 times slower than native Clean. Similar, though not quite as bad, results were obtained in other projects [30]. Generating the WebAssembly interpreter ourselves furthermore keeps our toolchain lightweight and gives more freedom to improve the generated WebAssembly with domain knowledge.

We do not provide benchmarks for the node copying overhead (in the deserialization setting) or the JavaScript interface (in the browser context) here, because we have not yet encountered realistic programs that spend a significant amount of time on these steps. The serialization library has been tested with the *Soccer-Fun* project [3] in which a soccer match is simulated. In our setup, the AIs of the players were interpreted and the gameplay was handled natively. Even though there is a lot of communication to make AIs aware of the state of the game, the copying of nodes back and forth was negligible in comparison to the time spent to calculate the next move, even with simple AIs. In the browser setting, we have tested the JavaScript interface with the interactive SVG editors described in [4]. Here too, only little time is spent in the interface, and most time is spent in the browser's rendering engine and the actual computations done inside the interpreter.

6 RELATED WORK

The ABC machine which our interpreter simulates was originally described by Koopman [19]. Since then, many additions and improvements have been made, most of which are not documented in the literature. Also a comprehensive description of the current native runtime system used by Clean remains a desideratum, although earlier versions have been described [31].

As mentioned above, our serialization library is a wrapper around the GraphCopy library, which was briefly described by Oortgiese et al. [20]. It is similar to solutions in other languages (see e.g. [8] for Haskell); Epstein et al. [12] use a different, class-based approach. While these solutions facilitate sharing of lazy expressions between executables containing *the same code* on *different platforms*, there have also been projects to allow the sharing of *unknown code* on *the same platform*. An example of this is Clean's dynamics system, which on 32-bit Windows systems can write and read values of type Dynamic to and from the disk [22]. The relevant object code is stored centrally and linked dynamically by what we called here the host program. Pil [22, p. 244] already anticipated the goal of sharing these dynamic values between different platforms, but noted: "We would like to stress here that we are only interested in efficient solutions. [...] The representation of a function by its source code, which can be interpreted at run-time, is not a satisfactory solution." Clearly, our knowledge of interpreters as well as processor speed has improved since 1997, and at least for some applications, interpretation has now become a feasible strategy — especially because in some contexts, like the web, remain unsuitable targets for the compilation of functional languages.

Du Bois and Da Rocha Costa [11] experimented with a small functional language that compiles to the JVM and uses Java remote method invocation to parallelise execution over several virtual machines. In this setup, as in ours, nodes in the distributed system can execute code it has not seen before. However, this comes at the cost of *all* code being interpreted or just-in-time compiled; there is no interworking of compiled and interpreted code.

In Erlang, compiled and interpreted code do work together. Code can at runtime be replaced by newer versions to fix bugs [5, pp. 78–80]. While originally Erlang was run in a virtual machine, it can now be just-in-time compiled as well [21]. In this setup, the compiled and the interpreted code work together on the same heap and stack. This clearly avoids the node copying overhead of our serialization library, but is less tailored to sandboxed execution.

In the *iTasks* framework [24, 25], our work has replaced that of Domoszlai et al. [10] as a method to bring Clean code to the browser and have a native Clean server communicate with the simulated Clean program on the client. In the previous setup, Clean was compiled to the minimal functional programming language Sapl [17], which was then compiled to JavaScript. The previous setup was faster on small benchmarks because the overhead of interpretation weighs heavier in this case. For larger programs, interpretation on a lower level (i.e., ABC instead of Clean or Sapl) turns out to be preferable. This also simplified communication between the server and the client. Due to limitations of WebAssembly, the interface with JavaScript has become slower than in the previous setting, but this difference has not been recognizable in actual usage.

7 CONCLUSIONS

We began our paper with the statement that functional languages are theoretically ideally suited for the execution of code unknown at compile-time: because functions are first-class citizens, this is just a special case of deserialization. We then described the implementation of a serialization library which supports this feature, to the best of our knowledge the first to do so in a platform-independent way. Internally, our library relies on an interpreter for Clean's intermediate language ABC, and copies nodes back and forth between the native and the interpreter heaps. This setup works well, and in the practical applications tested the overhead of copying nodes was not felt to be a major drawback.

[11] https://bugzilla.mozilla.org/show_bug.cgi?id=1167445.

We have described the impact on garbage collection and highlighted parallels with distributed systems that can aid implementers of similar systems. Since we rely on only one special feature of Clean's native runtime system, our implementation can be an example for similar libraries in other functional programming languages.

Unfortunately we have not been able to reconcile our two aims of getting the interpreter to interwork lazily and seamlessly with the host on the one hand, and evaluate serialized expressions safely on the other. Exception propagation, as supported by the GHC runtime system for Haskell, may be able to resolve this issue.

As outlined above, our sandboxed interpretation scheme is currently not entirely safe yet. We have described methods that can be used to ensure type coherence. In conjunction with dynamic typing, our serialization library would then become type-safe.

Lastly, we have shown that an interpreter for the stack-based intermediate language ABC is a suitable way to bring functional programming to the browser. Browser languages currently are a sub-ideal compilation target for functional languages due to the lack of recursion depth and non-local goto's. We have seen that the performance of our interpreter is more predictable and in most cases better than other solutions which work around this issue using continuation-passing style. Nevertheless, we intend to investigate whether a synthesis of these approaches is useful to further improve performance, for example by only using the interpreter in parts of the program where deep recursion or non-local goto's are required.

8 ACKNOWLEDGEMENTS

We thank Erin van der Veen for his previous work on the interpreter. We are also grateful to the IFL reviewers for their thoughtful comments.

REFERENCES

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. 1991. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems* 13, 2 (1991), 237–268.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. 1995. Dynamic Typing in Polymorphic Languages. *Journal of Functional Programming* 5, 1 (1995), 92–103.

[3] Peter Achten. 2011. The Soccer-Fun project. *Journal of Functional Programming* 21 (2011), 1–19. Issue 1.

[4] Peter Achten, Jurriën Stutterheim, László Domoszlai, and Rinus Plasmeijer. 2014. Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics. In *Proceedings of the 26nd International Symposium on Implementation and Application of Functional Languages, IFL '14*. ACM, New York.

[5] Joe Armstrong. 2003. *Making reliable distributed systems in the presence of software errors*. Ph.D. Dissertation. The Royal Institute of Technology, Stockholm.

[6] Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science* 6 (1996), 579–612. Issue 6.

[7] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. 2 (2018). Issue POPL.

[8] Jost Berthold. 2011. Orthogonal Serialisation for Haskell. In *Implementation and Application of Functional Languages. 22nd International Symposium, IFL 2010 (Lecture Notes in Computer Science)*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer, Heidelberg, 38–53.

[9] David Bushell and Ramiro Rikkert. [n.d.]. Pikaday/Pikaday: A refreshing JavaScript Datepicker. https://github.com/Pikaday/Pikaday. Retrieved June 15th, 2019.

[10] László Domoszlai, Eddy Bruël, and Jan Martin Jansen. 2011. Implementing a non-strict purely functional language in JavaScript. *Acta Universitatis Sapientiae* 3 (2011), 76–98. Issue 1.

[11] André Rauber Du Bois and Antônio Carlos Da Rocha Costa. 2001. Distributed Execution of Functional Programs Using the JVM. In *Computer Aided Systems Theory — EUROCAST 2001 (Lecture Notes in Computer Science)*, Roberto Moreno-Díaz, Bruno Buchberger, and José-Luis Freire (Eds.). Springer, Heidelberg.

[12] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. In *Haskell '11: Proceedings of the 4th ACM symposium on Haskell*. ACM, New York, 118–129.

[13] M. Anton Ertl and David Gregg. 2003. The Structure and Performance of *Efficient* Interpreters. *Journal of Instruction-Level Parallelism* 3 (2003), 1–25.

[14] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2017*. ACM, New York, 185–200.

[15] Pieter H. Hartel and Koen G. Langendoen. 1993. Benchmarking implementations of lazy functional languages. In *FPCA '93 Proceedings of the conference on Functional programming languages and computer architecture*. ACM, New York, 341–349.

[16] Tweag I/O. [n.d.]. WebAssembly as a Haskell compilation target - Asterius. https://tweag.github.io/asterius/webassembly/. Retrieved June 15th, 2019.

[17] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. 2006. Efficient interpretation by transforming data types and patterns to functions. In *Trends in Functional Programming*, Henrik Nilsson (Ed.), Vol. 7. 73–90.

[18] Marco Kesseler. 1996. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*. Ph.D. Dissertation. Radboud University Nijmegen.

[19] Pieter Koopman. 1990. *Functional programs as executable specifications*. Ph.D. Dissertation. Katholieke Universiteit Nijmegen.

[20] Arjan Oortgiese, John van Groningen, Peter Achten, and Rinus Plasmeijer. 2017. A Distributed Dynamic Architecture for Task Oriented Programming. In *IFL 2017: 29th Symposium on the Implementation and Application of Functional Programming Languages*. ACM, New York.

[21] Mikael Pettersson, Konstantinos Sagonas, and Erik Johansson. 2002. The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation. In *Functional and Logic Programming. FLOPS 2002 (Lecture Notes in Computer Science)*, Zhenjiang Hu and Mario Rodríguez-Artalejo (Eds.). Springer, Heidelberg, 228–244.

[22] Marco Pil. 1997. First Class File I/O. In *Implementation of Functional Languages. 8th International Workshop, IFL '96 (Lecture Notes in Computer Science)*, Werner Kluge (Ed.). Springer, Heidelberg, 233–246.

[23] Marco Pil. 1998. Dynamic Types and Type Dependent Functions. In *Implementation of Functional Languages. 10th International Workshop, IFL '98 (Lecture Notes in Computer Science)*, Kevin Hammond, Tony Davie, and Chris Clack (Eds.). Springer, Heidelberg, 169–185.

[24] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. 2007. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*, N. Ramsey (Ed.). ACM, New York, 141–152.

[25] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-oriented Programming in a Pure Functional Language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming. PPDP '12*. ACM, New York, 195–206.

[26] Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. 2011. Clean 2.2 Language Report. http://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf

[27] Andreas Rossberg (champion). [n.d.]. GC Proposal for WebAssembly. https://github.com/WebAssembly/gc/. Retrieved June 15th, 2019.

[28] Camil Staps and Mart Lubbers. 2016–2019. Cloogle: a search engine for the Clean programming language. https://cloogle.org. Retrieved June 15th, 2019.

[29] Dean Tribble, Mark Miller, and Till Schneidereit (champions). [n.d.]. WeakReferences TC39 proposal. https://github.com/tc39/proposal-weakrefs. Retrieved June 15th, 2019.

[30] Noah van Es, Quentin Stievenart, Jens Nicolay, Theo D'Hondt, and Coen De Roover. 2017. Implementing a performant scheme interpreter for the web in asm.js. *Computer Languages, Systems & Structures* 49 (2017), 62–81.

[31] John van Groningen. 1990. *Implementing the ABC-machine on MC680x0 based architectures*. Master's thesis. University of Nijmegen.

[32] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM Press, Portland, Oregon, USA, 301–312.