Towards Efficient and Verified Virtual Machines for Dynamic Languages

Martin Desharnais
National Cyber Defence Research Institute (CODE)
Universität der Bundeswehr München
Germany
martin.desharnais@unibw.de

Stefan Brunthaler
National Cyber Defence Research Institute (CODE)
Universität der Bundeswehr München
Germany
brunthaler@unibw.de

Abstract

The prevalence of dynamic languages is not commensurate with the security guarantees provided by their execution mechanisms. Consider, for example, the ubiquitous case of JavaScript: it runs everywhere and its complex just-in-time compilers produce code that is fast and, unfortunately, sometimes incorrect.

We present an Isabelle/HOL formalization of an alternative execution model—optimizing interpreters—and mechanically verify its correctness. Specifically, we formalize advanced speculative optimizations similar to those used in just-in-time compilers and prove semantics preservation. As a result, our formalization provides a path towards unifying vital performance requirements with desirable security guarantees.

CCS Concepts: • **Software and its engineering** → **Correctness**; **Software verification**; *Software performance*; • **Security and privacy** → *Software and application security*.

Keywords: formalization and verification, Isabelle, semantics, dynamic typing, speculative optimizations, interpreters, just-in-time compilers, inline caching, unboxing

ACM Reference Format:
Martin Desharnais and Stefan Brunthaler. 2021. Towards Efficient and Verified Virtual Machines for Dynamic Languages. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3437992.3439923

1 Motivation

Every day, every person with a computer or smartphone executes enormous amounts of JavaScript—knowingly or not. Confident that the machinery executing JavaScript works correctly, we use it day in and day out. A closer look at the correctness of JavaScript virtual machines shows that this confidence is unwarranted. Through abuse of implementation errors, attackers hijack victim devices through arbitrary code execution. Recently, Google's Project Zero published a complete series on so-called "JITSploitation" [18–20].

This should not come as a surprise, particularly as prior research has already looked at the prevalence of implementation errors in compilers [51]. Their comparison of the LLVM, GCC, and CompCert compilers provides strong evidence of the power of formalization and verification to reduce implementation errors.

To establish confidence in the JavaScript computing machinery, one would have to replicate the CompCert [30] effort for a JavaScript virtual machine. Prior research has shown that this approach is non-trivial [36]. Just-in-time compilers rely on self-modification and speculative optimizations to speed up programs. Both of these optimization techniques are at odds with the CompCert approach.

An alternative strategy to overcome these obstacles would be to sidestep just-in-time compilation and focus on interpreters instead. The expected advantages are ease of implementation, no self-modification, and no dynamic generation of native-machine code. Together, these advantages would also simplify the formalization and verification process.

But what kind of impact would such a strategy have on performance? Conventional wisdom states that interpreters are slow, and that performance requires just-in-time compilation. Prior research in interpreter optimization, however, reports remarkable and important speedups [8, 9, 13, 45, 50].

In this paper, we build on prior results in interpreter optimization to formalize and mechanically verify speculative optimizations: inline caching and unboxing. This formalization gives way to virtual machine interpreters that are both efficient and correct. Since our technique applies to all dynamic programming languages, it enables the construction of efficient and correct virtual machine interpreters for many popular languages, such as Lua, Perl, Python, and Ruby.

While these verifiably correct interpreters will not match the peak performance of their highly tuned just-in-time compiled counterparts, they offer acceptable performance for a

CPP '21, January 18–19, 2021, Virtual, Denmark
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8299-1/21/01.
https://doi.org/10.1145/3437992.3439923

```
1  JSValue jsAdd(JSGlobalObject* global, JSValue v1, JSValue v2) {
2    if (v1.isNumber() && v2.isNumber())
3      return jsNumber(v1.asNumber() + v2.asNumber());
4
5    if (v1.isString() && !v2.isObject()) {
6      if (v2.isString())
7        return jsString(global, asString(v1), asString(v2));
8      String s2 = v2.toWTFString(global);
9      return jsString(global, asString(v1), s2);
10   }
11
12   // All other cases are pretty uncommon
13   return jsAddSlowCase(global, v1, v2);
14 }
```

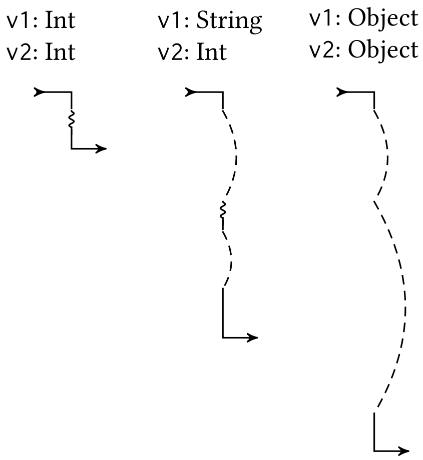

Figure 1. Resolving dynamic types and its impact on control flow. Squiggly lines indicate branches taken, dashed lines branches not taken. Arrows and horizontal lines indicate function entry and exit, i.e., calls and returns, respectively.

variety of tasks and computational needs. We believe, therefore, that using these efficient and verifiably correct interpreters is preferable in safety-critical, high-assurance contexts. In addition, these interpreters can form the backbone of secure, trusted infrastructure that we can rely upon when new just-in-time compiler bugs are exploited in the wild.

Summing up, this paper makes the following contributions:

- We present a formalization of self-optimizing bytecode interpreters for dynamically typed programming languages. In particular, we formalized advanced type feedback via inline caching and its extension to allow the manipulation of data in native-machine representation. Our formalization abstracts over the specific dynamic language and can be instantiated for many concrete languages.
- We prove both speculative optimizations to be sound and investigate their completeness. In addition, we show exemplary optimizing compilation passes, prove their soundness, and discuss their completeness.

Our work was developed using the Isabelle/HOL proof assistant. The theory files amount to around 4700 lines of source text and are publicly available in the *Archive of Formal Proofs* [11]. The *AFP* is continuously updated to track Isabelle's evolution, ensuring compatibility of the archived formalizations with future Isabelle versions.[1]

2 Background

Overhead of Dynamic Typing Figure 1 shows, on the left-hand side, the slightly simplified implementation of the add operation in JavaScriptCore, WebKit's JavaScript implementation, which is the open source version of Apple's Safari web browser. The dynamically-typed add operation resolves concrete type assignments according to the expected frequency. First, JavaScriptCore delegates to C++'s addition operator when both operands, v1 and v2, are numeric (lines 2 and 3 in Figure 1). Second, JavaScriptCore performs string concatenation, including coercion of the second operand, when the first operand is a string (lines 5–10 in Figure 1). Third, JavaScriptCore delegates implementation to jsAddSlowCase in all other cases (line 13), which is deemed "pretty uncommon" in the actual and original source code comment on line 12.

Figure 1 shows, on the right-hand side, the control flow including required branches for different operand type assignments. When both operands have type integer (Int in Figure 1, right-hand side, left column), control-flow takes the first branch and returns. When the first operand is a string (string in Figure 1, right-hand side, middle column), control flow requires at least one branch for testing against integers, plus a second branch if the second argument is not a string and needs to be coerced before returning the concatenated string. In all other cases, indicated by Object type assignments in the right column of Figure 1, the operation execution is delegated to yet another function, jsAddSlowCase, which requires two branches to determine less likely type assignments.

[1] Our formalization will be part of the next public *AFP* release, expected at the beginning of 2021. Before this release, use revision cb82935ea66a of the *AFP* development repository available at https://foss.heptapod.net/isa-afp/afp-devel.

From a performance perspective, the implementation of the add operator in Figure 1 indicates the performance penalties when type assignment expectations are not met. Consider a frequently executed, tight loop with a single string concatenation:

```
result = "";
for (i = 0; i < 100000; i++) {
   result += i;
}
```

In this example, the add operation will incur four branches to concatenate strings for *all iterations*. These branches are, however, redundant, as the type assignments of the operands for that specific occurrence of the add operation are invariant. If the string operands case were ranked first, then none of these branches were required, with the downside that now integer operands would suffer from the surplus type checks.

The effect of suboptimal static type-encoding in operation implementations of dynamic languages, as illustrated by the example above, has been known for decades. In 1982, Baden analyzed Smalltalk code and discovered what he termed a "dynamic locality of type usage." [3] In their landmark paper from 1984, Deutsch and Schiffman described what was to become one, if not the most, important optimization techniques to address this problem: *inline caching* [12]. In its original form, inline caching means that the virtual machine directly overwrites the target address of a call instruction in memory. So instead of calling the default routine that checks the types of all parameters—e.g., the type-generic function shown in Figure 1—one would overwrite the address of the call instruction to type-dependent function, prefixed with so-called guards, i.e., type checks to ensure that the expected types were passed. As a result, a subsequent execution of the same instruction will "short-circuit" the type checks and merely guard against expected types.

Overhead of Boxed Data Objects Boxed objects wrap primitive data types, such as numbers or characters. These primitive data usually can be manipulated using efficient native-machine operations and data representations. "Boxing" primitive data involves replacing the data item with a reference to an object representing the primitive data item. The resulting boxed object can, therefore, not be directly manipulated: assume that two numbers are in boxed object representation, then a simple machine addition, would add their addresses, instead of their numeric values. To manipulate boxed objects, their wrapped primitive data need to be "unboxed" first.

Boxing and unboxing requires surplus computation: to access the wrapped data, the computer must resolve the data references in the boxed objects. Additional operations, most often related to automatic memory management, must be taken into consideration as well. In Python, for example, each push operation that puts data onto the operand stack needs to adjust the object's reference count. Native-machine data, on the other hand, need not be reference counted, as they exist on their own in binary representation and need no automatic memory management.

With unboxing, data locality is improved, as the indirection via the boxed object wrapper is eliminated. Automatic memory management operations are reduced, as these operations are only required to manage boxed objects. Overall memory consumption can be reduced, because fewer objects are required. Automatic memory management techniques can be adjusted to take this into account. This effect is most pronounced on immediate memory management techniques such as reference counting.

On the other hand, boxed objects can be easily stored in the heap, and all other operations can refer to them in a uniform way using references or addresses. Boxed objects, furthermore, simplify the implementation of custom object and type systems.

3 Overview of the Formalization

Our formalization has three parts, each concerned with a separate programming language.

Dyn (Section 4) is a standard stack-based interpreter for dynamic languages, it provides a baseline for optimizations. The features provided by Dyn are intentionally kept minimal, but include the most representative features found in existing virtual machine interpreters for dynamic languages: operand stack manipulation, dynamic memory manipulation, built-in operations, conditional jumps, and (possibly recursive) function calls.

Inca (Section 5) extends Dyn with a speculative optimization known as inline caching. This type-based optimization is embedded directly in the semantics and, thus, performed automatically at run time. If the encountered types of an inlined operation match our speculation, the optimization is said to be a *hit*. Otherwise, the optimization is said to be a *miss* and must be rolled back. To ensure the soundness of this speculative optimization, we define a relation between unoptimized Dyn and optimized Inca programs, and prove that it is a bisimilarity, meaning that the compiled program has the same behavior as the unoptimized one and vice versa. In addition, we provide a simple compilation scheme and prove its soundness and completeness.

Ubx (Section 6) extends Inca with operations to manipulate unboxed, native-machine data. This optimization is also type-based but proceeds in two stages. First, an optimization pass rewrites the program *ahead of time* by substituting some type-generic instructions with type-specific alternatives that directly manipulate unboxed data. Second, the semantics is extended to perform the minimum number of checks at *run time* to ensure that the type-specific, optimized instructions rewritten in the first stage operate on the expected types and roll the optimization back if needed. Again, the soundness of this optimization is based on a bisimilarity relation, this time between unoptimized Inca programs and optimized Ubx programs. We provide an exemplary compilation scheme, too, based on a simple static analysis, and prove its soundness. We finish by discussing the incompleteness of this compilation scheme and some possible way forward.

We stove to keep the languages highly general by abstracting over a variety of implementation considerations. The most important abstraction is concerned with built-in operations. Instead of fixing a small set of these operations (such as arithmetic and Boolean operations) and optimizing them, we instead define an algebra of operations. For any operation of the algebra's carrier set, we can (i) determine the operation's arity,[2] and (ii) evaluate the operation on the given arguments. The semantics of all three languages, therefore, needs only to ensure that operations receive the correct number of arguments, and manipulate their results accordingly. By construction, this technique ensures that our formalization supports all operations, and we can mostly avoid arguing "without loss of generality."

To optimize these abstract operations, we progressively introduce more ways to manipulate the operations and check for speculative optimization opportunities. Thereby we are forced to state our formalization's assumptions.

Notation In the following paper, different typefaces or colors are used to identify different concepts. A *'blue* color, prefixed with a backtick, is used for abstract types and a *green* color from their abstract operations, which we will call parameters from now on to distinguish them from the built-in operations of the discussed languages. In contrast, a `monospace` typeface is used for concrete functions, defined either in this formalization or in the Isabelle/HOL standard library.

4 Dyn: Stack-Based Interpreter for Dynamically Typed Languages

The Dyn language corresponds to a simple, stack-based bytecode interpreter to execute a dynamically typed programming language. Figure 2 shows the syntax and dynamic state of Dyn.

4.1 Syntax and Semantics

Identifiers The identifiers for variables and functions are members of the abstract types *'var* and *'fun*, respectively.

Values The manipulated values belong to the abstract type *'dyn*. Dyn's semantics uses two disjoint subsets to decide whether values are true or false. Let x be a value, *IsTrue* x identifies the former, and *IsFalse* x the latter. This semantics is not affected by providing support for more types. Formally:

```
locale dynval =
  fixes
    IsTrue :: 'dyn ⇒ bool and
    IsFalse :: 'dyn ⇒ bool
  assumes
    ¬ (IsTrue x ∧ IsFalse x)
```

instr	::=	`Push` *'dyn* | `Pop` |	stack manipulation
		`Load` *'var* | `Store` *'var* |	memory manipulation
		`Op` *'op* |	operations on data
		`CJump` *nat* |	conditional jump
		`Call` *'fun*	function call
fundef	::=	`Fundef` $instr^*$ *nat*	function definition
prog	::=	`Prog` *'fenv* *'henv* *'fun*	program definition
frame	::=	`Frame` *'fun* *nat* $'dyn^*$	stack frame
state	::=	⟨*'fenv*, *'henv*, $frame^+$⟩	program state

Figure 2. The static syntax and dynamic state of Dyn.

In Isabelle, a *locale* [4] is a (possibly heterogeneous) algebra over some abstract types, with parameters, and subject to some assumptions. In the previous example, *'dyn* is the abstract type, *IsTrue* and *IsFalse* are the parameters, and the last line represents our assumption. A locale may then be later instantiated by providing concrete arguments for the types and parameters, and then discharging the proof obligations corresponding to the assumptions. All theorems proven for the locale are then automatically specialized for the concrete arguments. As a sanity check, all locales defined in this formalization were instantiated with suitable examples to ensure that the assumptions are consistent.

Operations The built-in operations are members of the type *'op* of the locale *nary_operations*. Let *op* be an operation, *Arity op* evaluates to *op*'s arity. *Op op xs* evaluates *op* on provided arguments *xs*; it is defined if and only if $|xs| = Arity\ op$. Formally:

```
locale nary_operations =
  fixes
    Op :: 'op ⇒ 'dyn list ⇒ 'dyn and
    Arity :: 'op ⇒ nat
```

Environments The environments are partial mappings from keys to values. In this paper, the important operations are *Get e k*, which retrieves the value associated with key k from the environment e, and *Add e k v*, which binds the key k with the value v in the environment e, overriding any prior bindings. Formally:

```
locale env =
  fixes
    Get :: 'env ⇒ 'key ⇒ 'val option and
    Add :: 'env ⇒ 'key ⇒ 'val ⇒ 'env and ...
  assumes
```

[2] All operations always return exactly one result. An operation with an arity of zero is equivalent to a constant.

$$\longrightarrow_{\textsc{Dyn}}\text{-}\textsc{Push}\ \frac{\mathit{FunGet}\ F\ f = \mathtt{Some}\ fd \qquad pc < |\mathtt{body}\ fd| \qquad \mathtt{body}\ fd\ !\ pc = \mathtt{Push}\ d}{\langle F,\ H,\ \mathtt{Frame}\ f\ pc\ \Sigma \cdot st\rangle \ \longrightarrow_{\textsc{Dyn}}\ \langle F,\ H,\ \mathtt{Frame}\ f\ (1+pc)\ (d \cdot \Sigma) \cdot st\rangle}$$

$$\longrightarrow_{\textsc{Dyn}}\text{-}\textsc{Pop}\ \frac{\mathit{FunGet}\ F\ f = \mathtt{Some}\ fd \qquad pc < |\mathtt{body}\ fd| \qquad \mathtt{body}\ fd\ !\ pc = \mathtt{Pop}}{\langle F,\ H,\ \mathtt{Frame}\ f\ pc\ (d \cdot \Sigma) \cdot st\rangle \ \longrightarrow_{\textsc{Dyn}}\ \langle F,\ H,\ \mathtt{Frame}\ f\ (1+pc)\ \Sigma \cdot st\rangle}$$

$$\longrightarrow_{\textsc{Dyn}}\text{-}\textsc{Load}\ \frac{\mathit{FunGet}\ F\ f = \mathtt{Some}\ fd \qquad pc < |\mathtt{body}\ fd| \qquad \mathtt{body}\ fd\ !\ pc = \mathtt{Load}\ x \qquad \mathit{MemGet}\ H\ (x, d_1) = \mathtt{Some}\ d_2}{\langle F,\ H,\ \mathtt{Frame}\ f\ pc\ (d_1 \cdot \Sigma) \cdot st\rangle \ \longrightarrow_{\textsc{Dyn}}\ \langle F,\ H,\ \mathtt{Frame}\ f\ (1+pc)\ (d_2 \cdot \Sigma) \cdot st\rangle}$$

$$\longrightarrow_{\textsc{Dyn}}\text{-}\textsc{Store}\ \frac{\mathit{FunGet}\ F\ f = \mathtt{Some}\ fd \qquad pc < |\mathtt{body}\ fd| \qquad \mathtt{body}\ fd\ !\ pc = \mathtt{Store}\ x \qquad H' = \mathit{MemAdd}\ H\ (x, d_1)\ d_2}{\langle F,\ H,\ \mathtt{Frame}\ f\ pc\ (d_1 \cdot d_2 \cdot \Sigma) \cdot st\rangle \ \longrightarrow_{\textsc{Dyn}}\ \langle F,\ H',\ \mathtt{Frame}\ f\ (1+pc)\ \Sigma \cdot st\rangle}$$

$$\longrightarrow_{\textsc{Dyn}}\text{-}\textsc{Op}\ \frac{\mathit{FunGet}\ F\ f = \mathtt{Some}\ fd \qquad pc < |\mathtt{body}\ fd| \qquad \mathtt{body}\ fd\ !\ pc = \mathtt{Op}\ op \qquad ar = \mathit{Arith}\ op \qquad ar \le |\Sigma| \qquad \mathit{Op}\ op\ (\mathtt{take}\ ar\ \Sigma) = d}{\langle F,\ H,\ \mathtt{Frame}\ f\ pc\ \Sigma \cdot st\rangle \ \longrightarrow_{\textsc{Dyn}}\ \langle F,\ H,\ \mathtt{Frame}\ f\ (1+pc)\ (d \cdot \mathtt{drop}\ ar\ \Sigma) \cdot st\rangle}$$

$$\longrightarrow_{\textsc{Dyn}}\text{-}\textsc{CJump-True}\ \frac{\mathit{FunGet}\ F\ f = \mathtt{Some}\ fd \qquad pc < |\mathtt{body}\ fd| \qquad \mathtt{body}\ fd\ !\ pc = \mathtt{CJump}\ n \qquad \mathit{IsTrue}\ d}{\langle F,\ H,\ \mathtt{Frame}\ f\ pc\ (d \cdot \Sigma) \cdot st\rangle \ \longrightarrow_{\textsc{Dyn}}\ \langle F,\ H,\ \mathtt{Frame}\ f\ n\ \Sigma \cdot st\rangle}$$

$$\longrightarrow_{\textsc{Dyn}}\text{-}\textsc{CJump-False}\ \frac{\mathit{FunGet}\ F\ f = \mathtt{Some}\ fd \qquad pc < |\mathtt{body}\ fd| \qquad \mathtt{body}\ fd\ !\ pc = \mathtt{CJump}\ n \qquad \mathit{IsFalse}\ d}{\langle F,\ H,\ \mathtt{Frame}\ f\ pc\ (d \cdot \Sigma) \cdot st\rangle \ \longrightarrow_{\textsc{Dyn}}\ \langle F,\ H,\ \mathtt{Frame}\ f\ (1+pc)\ \Sigma \cdot st\rangle}$$

$$\longrightarrow_{\textsc{Dyn}}\text{-}\textsc{Fun-Call}\ \frac{\mathit{FunGet}\ F\ f = \mathtt{Some}\ fd \qquad pc < |\mathtt{body}\ fd| \qquad \mathtt{body}\ fd\ !\ pc = \mathtt{Call}\ g \qquad \mathit{FunGet}\ F\ g = \mathtt{Some}\ gd \qquad ar = \mathit{Arity}\ gd \qquad ar \le |\Sigma|}{\langle F,\ H,\ \mathtt{Frame}\ f\ pc\ \Sigma \cdot st\rangle \ \longrightarrow_{\textsc{Dyn}}\ \langle F,\ H,\ \mathtt{Frame}\ g\ 0\ (\mathtt{take}\ ar\ \Sigma) \cdot \mathtt{Frame}\ f\ pc\ \Sigma \cdot st\rangle}$$

$$\longrightarrow_{\textsc{Dyn}}\text{-}\textsc{Fun-End}\ \frac{\mathit{FunGet}\ F\ g = \mathtt{Some}\ gd \qquad pc_g = |\mathtt{body}\ gd| \qquad ar = \mathit{Arity}\ gd \qquad ar \le |\Sigma_f|}{\langle F,\ H,\ \mathtt{Frame}\ g\ pc_g\ \Sigma_g \cdot \mathtt{Frame}\ f\ pc_f\ \Sigma_f \cdot st\rangle \ \longrightarrow_{\textsc{Dyn}}\ \langle F,\ H,\ \mathtt{Frame}\ f\ (1+pc_f)\ (\Sigma_g\ @\ \mathtt{drop}\ ar\ \Sigma_f) \cdot st\rangle}$$

Figure 3. The ($\longrightarrow_{\textsc{Dyn}}$) transition relation for Dyn.

$$\mathit{Get}\ (\mathit{Add}\ e\ k\ v)\ k = \mathtt{Some}\ v\ \textbf{and}$$
$$k_1 \neq k_2 \Longrightarrow \mathit{Get}\ (\mathit{Add}\ e\ k_1\ v)\ k_2 = \mathit{Get}\ e\ k_2\ \textbf{and}\ \ldots$$

We use two environments, one for function definitions (types *'fenv*, *'fun*, and *fundef*) and another one to model dynamic memory (types *'henv*, *'var* × *'dyn*, *'dyn*). The parameters are prefixed with *Fun* and *Mem*, respectively.

Static representation *Instructions* belong to one of the following categories: manipulation of operand stack, manipulation of dynamic memory, built-in operations, conditional jumps, and function calls. *Function definitions* contain a list of instructions and the function's arity. *Programs* contain an environment for functions, an initial memory, and an initializing function.

Dynamic states *Stack frames* contain the identifier of the current function, a program counter relative to the beginning of the function, and a (possibly empty) operand stack. *Program states* contain an environment for functions, an initial memory, and a non-empty call stack.

Loading and initial states The binary $\mathtt{load}_{\textsc{Dyn}}$ relation associates the static representation of a program to an initial dynamic program state. More precisely, the relation initializes the program state, obtains the initializing function from the program, and transfers control to this function.

Final states The predicate $\mathtt{final}_{\textsc{Dyn}}$ identifies final states the ones having a call stack with a single stack frame, where the program counter points beyond the last instruction.

Operational semantics The operational semantics is defined by the small-step transition relation $\longrightarrow_{\textsc{Dyn}}$ between program states (Figure 3). Most instructions' semantics corresponds to well-known, standard behavior. The dynamic memory is partitioned by variable names, which are statically encoded in the load and store instructions, and each partition may contain any number of dynamic values, which are indexed by a dynamic value taken from the operand stack.

The rule $\longrightarrow_{\textsc{Dyn}}$-Op assumes that there are enough arguments on the operand stack before evaluating the operation. This assumption ensures that the function *Op op* is defined for the list of operands $\mathtt{take}\ ar\ \Sigma$.

Similarly, the rule $\longrightarrow_{\textsc{Dyn}}$-Fun-Call assumes that the number of operands equals the arity of the called function. A new stack frame is created, the arguments copied to the new stack frame's operand stack. Note that a function may call itsefl recursively.

The rule $\longrightarrow_{\textsc{Dyn}}$-Fun-End proceeds in two steps. First, the remaining values on the called function's operand stack are interpreted as its result and its stack frame is discarded. Second, the arguments on top of the calling function's operand stack are replaced by the called function's result and the program counter is incremented.

The rule $\rightarrow_{\text{Dyn}}$-CJump-True transfers the control flow to a position relative to the beginning of the function. Note that execution gets stuck if a jump condition represents neither true nor false.

5 Inca: Inline Caching

The Inca language extends Dyn with a single instruction for inline caching of operations.

5.1 Syntax and Semantics

The syntax of Inca is a proper superset of Dyn's syntax. The only addition is an instruction to inline operations (Figure 4).

Inlined operations The built-in inlined operations are members of the type *'opinl* of the locale *nary_operations_inl*. Figure 5 illustrates the relationship between the sets *'op* and *'opinl*. An operation from *'op* may be mapped to any number (including none) of inlined operations in *'opinl* with *Inl*, which gives the most specific inlined operation for concrete operand types. This mapping may be inverted with Inl^{-1}.

A typical implementation of the *Inl* function may start with a case analysis of the operation followed by a linear search for the most specific inlined function. Depending on the cardinality of *'op* and *'opinl*, this may be time consuming and should be avoided when possible. When we are evaluating an inline operation, it is more efficient to leverage the "dynamic locality of type usage" by using *IsInl* to ensure that the expected operand types and the actual operand types match.

Finally, *InlOp* can be used to evaluate inline operations with given arguments. It is defined if and only if *Op* is defined for the corresponding operation and given arguments. In that case, the inlined and the normal operations must always produce the same results. Formally:

$$
\begin{array}{l}
\textbf{locale}\ \mathit{nary\_operations\_inl} = \mathit{nary\_operations}\ +\\
\quad \textbf{fixes}\\
\qquad \mathit{InlOp} :: \mathit{'opinl} \Rightarrow \mathit{'dyn}\ \text{list} \Rightarrow \mathit{'dyn}\ \textbf{and}\\
\qquad \mathit{Inl} :: \mathit{'op} \Rightarrow \mathit{'dyn}\ \text{list} \Rightarrow \mathit{'opinl}\ \text{option}\ \textbf{and}\\
\qquad \mathit{Inl}^{-1} :: \mathit{'opinl} \Rightarrow \mathit{'op}\ \textbf{and}\\
\qquad \mathit{IsInl} :: \mathit{'opinl} \Rightarrow \mathit{'dyn}\ \text{list} \Rightarrow \text{bool}\\
\quad \textbf{assumes}\\
\qquad \mathit{Inl}\ op\ xs = \mathtt{Some}\ op_{inl} \Longrightarrow \mathit{Inl}^{-1}\ op_{inl} = op\ \textbf{and}\\
\qquad \mathit{Inl}\ op\ xs = \mathtt{Some}\ op_{inl} \Longrightarrow \mathit{IsInl}\ op_{inl}\ xs\ \textbf{and}\\
\qquad |xs| = \mathit{Arity}\ (\mathit{Inl}^{-1}\ op_{inl}) \Longrightarrow\\
\qquad\quad \mathit{InlOp}\ op_{inl}\ xs = \mathit{Op}\ (\mathit{Inl}^{-1}\ op_{inl})\ xs
\end{array}
$$

Semantics Inca's dynamic representation, its loading relation $\mathtt{load}_{\text{Inca}}$, and its set of final states (identified by the predicate $\mathtt{final}_{\text{Inca}}$) are all the same as their Dyn counterparts. We modify the transition relation by adding three new rules and modifying the existing rule $\rightarrow_{\text{Dyn}}$-Op to $\rightarrow_{\text{Inca}}$-Op (Figure 6).

$$
\begin{array}{lll}
\mathit{instr}\ ::=\ \cdots\ | & & \text{instructions from Dyn}\\
 & \mathtt{OpInl}\ \mathit{'opinl} & \text{inlined operations on data}
\end{array}
$$

Figure 4. The static syntax of Inca's instructions.

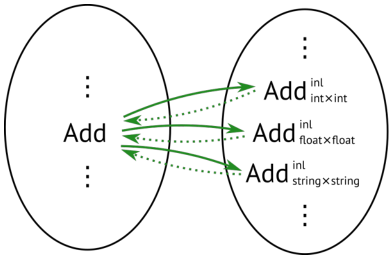

Figure 5. The relationship between the set of values from *'op* (left) and *'opinl* (right). Solid arrows represent calls to *Inl* and dotted arrows represent calls to Inl^{-1}.

When executing an operation (`Op`), *Inl* is used to check if an inlined operation exists for the supplied arguments. If no such inlined operation exists (Rule $\rightarrow_{\text{Inca}}$-Op), then the operation is evaluated with *Op*, and execution continues as in Dyn. If such an inlined operation exists (Rule $\rightarrow_{\text{Inca}}$-Op-Inl), then two things take place. First, we evaluate the operation with *InlOp*. Second, we cache the search for an optimized inline operation by replacing the `Op` instruction with an optimized `OpInl` instruction in the function definition (rewrite). As a result, any subsequent execution then "short-circuits" the check for an inlined operation.

When executing an inlined operation (`OpInl`), the efficient predicate *IsInl* is used to test whether it is still appropriate for the supplied arguments. If they are, then execution continues as expected using an optimized function (Rule $\rightarrow_{\text{Inca}}$-Op-Inl-Hit). Otherwise, we undo the optimization by replacing the optimized instruction with the generic, unoptimized instruction in the function definition (Rule $\rightarrow_{\text{Inca}}$-Op-Inl-Miss). Whether we use *Op* or *InlOp* is irrelevant, since they are semantically equivalent and because it is unknown which one would be more efficient.

5.2 Bisimulation Dyn-Inca

A Dyn and an Inca program that simulate each other differ only in the codomain of their functions environments. The dynamic memories, the call stacks, and the domains of the function environments are identical. Given two corresponding function definitions from Dyn and Inca, they may only differ by the potential use of inline operations.

The simulation relation $\overset{\text{D-I}}{\sim}$ thus inspects all corresponding instructions and checks whether an inlined operation maps to its corresponding regular operation. We use the inverse function Inl^{-1} to map an inlined to its corresponding regular operation.

$$\rightarrow_{\text{INCA}}\text{-OP}\ \frac{\begin{array}{c} \mathit{FunGet}\ F\ f = \texttt{Some}\ \mathit{fd} \qquad \mathit{pc} < |\texttt{body}\ \mathit{fd}| \qquad \texttt{body}\ \mathit{fd}\ !\ \mathit{pc} = \texttt{Op}\ \mathit{op} \\ \mathit{ar} = \mathit{Arith}\ \mathit{op} \qquad \mathit{ar} \leq |\Sigma| \qquad \mathit{Inl}\ \mathit{op}\ (\texttt{take}\ \mathit{ar}\ \Sigma) = \texttt{None} \qquad \mathit{Op}\ \mathit{op}\ (\texttt{take}\ \mathit{ar}\ \Sigma) = d \end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ \mathit{pc}\ \Sigma \cdot \mathit{st}\rangle \ \rightarrow_{\text{INCA}}\ \langle F,\ H,\ \texttt{Frame}\ f\ (1 + \mathit{pc})\ (d \cdot \texttt{drop}\ \mathit{ar}\ \Sigma) \cdot \mathit{st}\rangle}$$

$$\rightarrow_{\text{INCA}}\text{-OP-INL}\ \frac{\begin{array}{c} \mathit{FunGet}\ F\ f = \texttt{Some}\ \mathit{fd} \qquad \mathit{pc} < |\texttt{body}\ \mathit{fd}| \qquad \texttt{body}\ \mathit{fd}\ !\ \mathit{pc} = \texttt{Op}\ \mathit{op} \\ \mathit{ar} = \mathit{Arith}\ \mathit{op} \qquad \mathit{ar} \leq |\Sigma| \qquad \mathit{Inl}\ \mathit{op}\ (\texttt{take}\ \mathit{ar}\ \Sigma) = \texttt{Some}\ \mathit{op}_{\mathit{inl}} \qquad \mathit{InlOp}\ \mathit{op}_{\mathit{inl}}\ (\texttt{take}\ \mathit{ar}\ \Sigma) = d \\ F' = \mathit{FunAdd}\ F\ f\ (\texttt{rewrite}\ \mathit{fd}\ \mathit{pc}\ (\texttt{OpInl}\ \mathit{op}_{\mathit{inl}})) \end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ \mathit{pc}\ \Sigma \cdot \mathit{st}\rangle \ \rightarrow_{\text{INCA}}\ \langle F',\ H,\ \texttt{Frame}\ f\ (1 + \mathit{pc})\ (d \cdot \texttt{drop}\ \mathit{ar}\ \Sigma) \cdot \mathit{st}\rangle}$$

$$\rightarrow_{\text{INCA}}\text{-OP-INL-HIT}\ \frac{\begin{array}{c} \mathit{FunGet}\ F\ f = \texttt{Some}\ \mathit{fd} \qquad \mathit{pc} < |\texttt{body}\ \mathit{fd}| \qquad \texttt{body}\ \mathit{fd}\ !\ \mathit{pc} = \texttt{OpInl}\ \mathit{op}_{\mathit{inl}} \\ \mathit{op} = \mathit{Inl}^{-1}\ \mathit{op}_{\mathit{inl}} \qquad \mathit{ar} = \mathit{Arith}\ \mathit{op} \qquad \mathit{ar} \leq |\Sigma| \qquad \mathit{IsInl}\ \mathit{op}_{\mathit{inl}}\ (\texttt{take}\ \mathit{ar}\ \Sigma) \qquad \mathit{InlOp}\ \mathit{op}_{\mathit{inl}}\ (\texttt{take}\ \mathit{ar}\ \Sigma) = d \end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ \mathit{pc}\ \Sigma \cdot \mathit{st}\rangle \ \rightarrow_{\text{INCA}}\ \langle F,\ H,\ \texttt{Frame}\ f\ (1 + \mathit{pc})\ (d \cdot \texttt{drop}\ \mathit{ar}\ \Sigma) \cdot \mathit{st}\rangle}$$

$$\rightarrow_{\text{INCA}}\text{-OP-INL-MISS}\ \frac{\begin{array}{c} \mathit{FunGet}\ F\ f = \texttt{Some}\ \mathit{fd} \qquad \mathit{pc} < |\texttt{body}\ \mathit{fd}| \qquad \texttt{body}\ \mathit{fd}\ !\ \mathit{pc} = \texttt{OpInl}\ \mathit{op}_{\mathit{inl}} \\ \mathit{op} = \mathit{Inl}^{-1}\ \mathit{op}_{\mathit{inl}} \qquad \mathit{ar} = \mathit{Arith}\ \mathit{op} \qquad \mathit{ar} \leq |\Sigma| \qquad \neg\ \mathit{IsInl}\ \mathit{op}_{\mathit{inl}}\ (\texttt{take}\ \mathit{ar}\ \Sigma) \qquad \mathit{InlOp}\ \mathit{op}_{\mathit{inl}}\ (\texttt{take}\ \mathit{ar}\ \Sigma) = d \\ F' = \mathit{FunAdd}\ F\ f\ (\texttt{rewrite}\ \mathit{fd}\ \mathit{pc}\ (\texttt{Op}\ \mathit{op})) \end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ \mathit{pc}\ \Sigma \cdot \mathit{st}\rangle \ \rightarrow_{\text{INCA}}\ \langle F',\ H,\ \texttt{Frame}\ f\ (1 + \mathit{pc})\ (d \cdot \texttt{drop}\ \mathit{ar}\ \Sigma) \cdot \mathit{st}\rangle}$$

Figure 6. The subset of the ($\rightarrow_{\text{INCA}}$) transition relation that differs from ($\rightarrow_{\text{DYN}}$).

Following the path of CompCert, we proved the following lemmas to show that $\overset{\text{D-I}}{\sim}$ is a bisimulation, i.e., that two similar programs have similar behavior.

Lemma 1 (Forward simulation). *If $s_1 \ \rightarrow_{\mathit{DYN}}\ s_1'$ and $s_1 \overset{\text{D-I}}{\sim} s_2$, then there exists a state s_2' such that $s_2 \ \rightarrow_{\mathit{INCA}}\ s_2'$ and $s_1' \overset{\text{D-I}}{\sim} s_2'$.*

Lemma 2 (Forward matching final states). *If $s_1 \overset{\text{D-I}}{\sim} s_2$ and $\texttt{final}_{\mathit{DYN}}\ s_1$, then $\texttt{final}_{\mathit{INCA}}\ s_2$.*

Lemma 3 (Backward simulation). *If $s_2 \ \rightarrow_{\mathit{INCA}}\ s_2'$ and $s_1 \overset{\text{D-I}}{\sim} s_2$, then there exists a state s_1' such that $s_1 \ \rightarrow_{\mathit{DYN}}\ s_1'$ and $s_1' \overset{\text{D-I}}{\sim} s_2'$.*

Lemma 4 (Backward matching final states). *If $s_1 \overset{\text{D-I}}{\sim} s_2$ and $\texttt{final}_{\mathit{INCA}}\ s_2$, then $\texttt{final}_{\mathit{DYN}}\ s_1$.*

5.3 Compilation from Dyn to Inca

Dyn’s function definitions can be compiled by mapping all instructions to their equivalent in Inca. The compilation function of full programs can then simply compile all function definitions of the program.

We proved that compiled programs simulate their uncompiled counterparts.

Lemma 5 (Compiled matching states). *If $\texttt{compile}\ p_1 = \texttt{Some}\ p_2$ and $\texttt{load}_{\mathit{DYN}}\ p_1\ s_1$, then there exists a state s_2 such that $\texttt{load}_{\mathit{INCA}}\ p_2\ s_2$ and $s_1 \overset{\text{D-I}}{\sim} s_2$.*

Building on the VeriComp framework for verified compilation [10], lemmas 1 to 5 imply that the successful execution of a compiled Inca program exhibits the identical behavior as the execution of the original Dyn program. Formally:

Theorem 1 (Soundness of compilation). *Let the infix relation $\Downarrow$ pair a program to its run-time behavior and the infix relation $\approx$ be an equivalence relation between behaviors. If $\texttt{compile}\ p_1 = \texttt{Some}\ p_2$, and $p_2 \Downarrow b_2$, and b_2 does not go wrong, then there exists a behavior b_1 such that $p_1 \Downarrow b_1$ and $b_1 \approx b_2$.*

Furthermore, compilation is complete for all loadable Dyn programs.

Theorem 2 (Completness of compilation). *If $\texttt{load}_{\mathit{DYN}}\ p_1\ s_1$, then there exists a program p_2 and state s_2 such that $\texttt{compile}\ p_1 = \texttt{Some}\ p_2$, and $\texttt{load}_{\mathit{INCA}}\ p_2\ s_2$, and $s_1 \overset{\text{D-I}}{\sim} s_2$.*

6 Ubx: Operations on Unboxed Data

The Ubx language adds the concept of manipulating unboxed data representations to Inca.

6.1 Syntax and Semantics

The syntax of Ubx is a proper superset of Inca’s syntax (Figure 7).

Values The values manipulated through the operand stack may either be boxed or unboxed. In principle, any fixed number of unboxed types may be supported but, Isabelle/HOL not supporting abstractions over arbitrary numbers of types, we abstract over two unboxed types ($'\mathit{ubx}_1$ and $'\mathit{ubx}_2$) and have to argue without loss of generality.

Because the operand stack may only contain values of a uniform type, we define the tagged union *ubx* with three constructors: UbxDyn represents a boxed value while both $\texttt{UbxUbx}_1$ and $\texttt{UbxUbx}_2$ represent unboxed values. We extract a value stored in an *ubx* by casting it to the desired type. Casting (i) checks that the *ubx* value is tagged with the

ubx	::=	UbxDyn *'dyn* |	boxed dynamic value
		$\mathtt{UbxUbx}_1$ *'ubx₁* |	unboxed value 1
		$\mathtt{UbxUbx}_2$ *'ubx₂* |	unboxed value 2
type	::=	$\mathtt{Ubx}_1$ | $\mathtt{Ubx}_2$	unboxed types
instr	::=	$\cdots$ |	instructions from Inca
		$\mathtt{PushUbx}_1$ *'ubx₁* |	stack manipulation
		$\mathtt{PushUbx}_2$ *'ubx₂* |	of unboxed data
		LoadUbx *type 'var* |	memory manipulation
		StoreUbx *type 'var* |	of unboxed data
		OpUbx *'opubx*	unboxed operations
frame	::=	Frame *'fun nat* ubx^*	stack frame

Figure 7. An excerpt of Ubx's static syntax and dynamic state of Ubx.

expected constructor for the given type, and (ii) returns the unboxed value.

fun $\mathtt{cast}_{\mathtt{Dyn}}$ :: *ubx* ⇒ *'dyn option* **where**
  $\mathtt{cast}_{\mathtt{Dyn}}$ (UbxDyn *d*) = Some *d*
  $\mathtt{cast}_{\mathtt{Dyn}}$ _ = None

The functions $\mathtt{cast}_{\mathtt{Ubx}_1}$ and $\mathtt{cast}_{\mathtt{Ubx}_2}$ are analog but return values of type *'ubx₁* and *'ubx₂*, respectively. Our formalization proves that casts are always successful and an implementation of this optimization would be free to omit.

The boxing and unboxing operations are abstracted over in the locale *unboxedval*. Let *d* be a dynamic value and *u* an unboxed value of type *'ubx₁*, $Unbox_1\ x$ = Some *u* successfully extracts the native-machine value *u*, and $Box_1\ u$ boxes it back to *d*. Unboxing may fail by evaluating to None when the provided dynamic value is not of the expected type. The same holds for *'ubx₂*, and extends to any other supported unboxed type. Formally:

locale *unboxedval* = *dynval* +
  fixes
    Box_1 :: *'ubx₁* ⇒ *'dyn* **and**
    $Unbox_1$ :: *'dyn* ⇒ *'ubx₁* option **and**
    Box_2 :: *'ubx₂* ⇒ *'dyn* **and**
    $Unbox_2$ :: *'dyn* ⇒ *'ubx₂* option
  assumes
    $Unbox_1\ d = \mathrm{Some}\ u_1 \Longrightarrow Box_1\ u_1 = d$ **and**
    $Unbox_2\ d = \mathrm{Some}\ u_2 \Longrightarrow Box_2\ u_2 = d$

In order to uniformly manipulate *ubx* when boxing and unboxing, we define the type *type* which has one constructor per unboxed type, i.e., $\mathtt{Ubx}_1$ is associated with *'ubx₁* and $\mathtt{Ubx}_2$ is associated with *'ubx₂*. The generic cast_box function (i) casts unboxed values, and (ii) immediately boxes them to a dynamic value.

fun cast_box :: *type* ⇒ *ubx* ⇒ *'dyn option* **where**
  cast_box $\mathtt{Ubx}_1$ = map_option Box_1 ∘ $\mathtt{cast}_{\mathtt{Ubx}_1}$
  cast_box $\mathtt{Ubx}_2$ = map_option Box_2 ∘ $\mathtt{cast}_{\mathtt{Ubx}_2}$

Conversely, the generic function unbox unboxes *'dyn* values to some specified type.

fun unbox :: *type* ⇒ *'dyn* ⇒ *ubx option* **where**
  unbox $\mathtt{Ubx}_1$ = map_option $\mathtt{UbxUbx}_1$ ∘ $Unbox_1$
  unbox $\mathtt{Ubx}_2$ = map_option $\mathtt{UbxUbx}_2$ ∘ $Unbox_2$

Finally, a *ubx* value may be boxed and normalized to *'dyn*.

fun norm :: *ubx* ⇒ *'dyn* **where**
  norm (UbxDyn *d*) = *d*
  norm ($\mathtt{UbxUbx}_1\ u_1$) = $Box_1\ u_1$
  norm ($\mathtt{UbxUbx}_2\ u_2$) = $Box_2\ u_2$

Instructions One new instruction per unboxed type pushes an unboxed constant onto the operand stack. Two generic instructions allow loading unboxed values from and storing them in memory. Finally, one instruction manipulates unboxed, native-machine data. The number of new instructions to support n unboxed types is thus $n + 3$.

Operations on unboxed data The built-in operations on unboxed data are members of the type *'opubx* of the locale *nary_operations_ubx*. Let op_{ubx} be an operation on unboxed data and *xs* be a list of values of type *ubx*, *UbxOp* op_{ubx} *xs* uses some efficient machine-native instructions to operate directly on the given unboxed arguments. In contrast to *Op* and *OpInl* which, when given the correct number of arguments, always succeed to calculate a result, *OpUbx* may fail by returning None when evaluated on unboxed values of the wrong type.

An inlined operation (*'opinl*) is mapped to an operation on unboxed data (*'opubx*) with the *Ubx* function. But instead of relying on the dynamic type-information extracted from the actual *'dyn* arguments at run time, it relies on statically known type information. Each argument either has a boxed type (Some τ for some τ :: *type*), or an unboxed, dynamic type (None). The mapping is inverted with Ubx^{-1}.

Finally, *TypeOf* op_{ubx} evaluates to the type of the operation op_{ubx} encoded as a pair: the first element is the domain and the second element is the codomain. The type of an operation must be compatible with *Arity*, *Ubx*, and *UbxOp*. Formally:

locale *nary_operations_ubx* =
    nary_operations_inl +
    unboxedval +
  fixes
    UbxOp :: *'opubx* ⇒ *ubx* list ⇒ *ubx* option **and**
    Ubx :: *'opinl* ⇒ *type* list ⇒ *'opubx* option **and**
    Ubx^{-1} :: *'opubx* ⇒ *'opinl* **and**
    TypeOf :: *'opubx* ⇒ *type* option list × *type* option
  assumes
    $Ubx\ op_{inl}\ ts = \mathrm{Some}\ op_{ubx} \Longrightarrow Ubx^{-1}\ op_{ubx} = op_{inl}$ **and**
    $UbxOp\ op_{ubx}\ xs = \mathrm{Some}\ y \Longrightarrow$

$$
\begin{aligned}
&\quad \mathit{InlOp}\ (\mathit{Ubx}^{-1}\ op_{ubx})\ (\texttt{map}\ \texttt{norm}\ xs) = \texttt{norm}\ y \ \textbf{and}\\
&\mathit{UbxOp}\ op_{ubx}\ xs = \texttt{Some}\ y \Longrightarrow\\
&\quad \mathit{Inl}\ (\mathit{Inl}^{-1}\ (\mathit{Ubx}^{-1}\ op_{ubx}))\ (\texttt{map}\ \texttt{norm}\ xs) =\\
&\quad \texttt{Some}\ (\mathit{Ubx}^{-1}\ op_{ubx})\ \textbf{and}\\
&\mathit{Arity}\ (\mathit{Inl}^{-1}\ (\mathit{Ubx}^{-1}\ op_{ubx})) = |\texttt{fst}\ (\mathit{TypeOf}\ op_{ubx})|\ \textbf{and}\\
&\mathit{Ubx}\ op_{inl}\ ts = \texttt{Some}\ op_{ubx} \Longrightarrow\\
&\quad \texttt{fst}\ (\mathit{TypeOf}\ op_{ubx}) = ts\ \textbf{and}\\
&\mathit{TypeOf}\ op_{ubx} = (\texttt{map}\ \texttt{typeof}\ xs, \tau) \Longrightarrow\\
&\quad \exists x.\ \mathit{UbxOp}\ op_{ubx}\ xs = \texttt{Some}\ x\ \land \texttt{typeof}\ x = \tau\ \textbf{and}\\
&\mathit{UbxOp}\ op_{ubx}\ xs = \texttt{Some}\ x \Longrightarrow\\
&\quad \mathit{TypeOf}\ op_{ubx} = (\texttt{map}\ \texttt{typeof}\ xs, \texttt{typeof}\ x)
\end{aligned}
$$

Semantics We extend Inca's transition relation to also support *ubx* (Figure 8).

All rules to push constants onto the stack now use the appropriate constructor from *ubx*.

The rules for loading values from the dynamic memory distinguish three cases: (i) a dynamic value is loaded and pushed directly on the operand stack; (ii) a dynamic value is loaded, successfully unboxed, and pushed on the operand stack; and (iii) a dynamic value is loaded, the unboxing fails, and the function is generalized to cancel the Ubx optimization. All three rules start by (a) popping a value from the operand stack, and (b) casting it to a dynamic value, which is then used to index the dynamic memory.

In rule $\rightarrow_{\text{Ubx}}$-Load-Ubx-Miss, the unboxing fails because the dynamic value loaded from memory has a different type than what was expected when optimizing the program. Subsequent instructions expecting data in their native-machine representation cannot execute sensibly and must be generalized to cope with dynamic values. This generalization process applies to both the function definition and the call stack.

First, the function `generalize` generalizes the function definition by mapping all Ubx instructions to their Inca counterparts, e.g., $\texttt{PushUbx}_1$ to `Push`. For `OpInl` instructions, Ubx^{-1} identifies the corresponding *'opinl* operation.

Second, we need to update the operand stack to ensure that all elements use the boxed representation. If a tagged union contains an operand in unboxed data representation, these operands would not be accepted by the newly generalized instructions. To address this, we use the type information stored in the tagged union to box the object and replace the element with another tagged union (`UbxDyn`) representing this newly boxed object. The operand stack of the current stack frame must be boxed, but so do the operand stacks of all other active stack frames of the same function. Because each stack frame only stores the identifier of the function, and each execution step retrieves the instruction from the function definition, all active function invocations will start to use the generalized instructions. The function `box_stack` does this by recursively traversing the call stack and generalizing the operand stack of all stack frame for function f; all other stack frames are left untouched.

The rules for storing values in memory all cast the operand on the top of the stack to the expected type and box it before storing it in memory. No rules are needed to handle the case that an unboxed type does not match its expected type. The bisimulation relation proves that such a situation can never occur.

The rules for evaluating regular and inlined operations require minimal adaptation: they must first cast their operands to dynamic values before evaluation. Again, no rule is required to handle an invalid cast, as our proof shows that such situations can never occur. The new rule $\rightarrow_{\text{Ubx}}$-Op-Ubx does not need to perform any cast as it operates directly on unboxed data.

Similarly, the rules for conditional jumps and function call require minimal adaptation; they now cast their operand to a dynamic boolean value.

The rules $\rightarrow_{\text{Ubx}}$-Pop, and $\rightarrow_{\text{Ubx}}$-Fun-End do not need to change because they are polymorphic, i.e., they perform the same operation irrespective of the operand types they manipulate.

6.2 Bisimulation Inca-Ubx

The validity of a sequence of Ubx instructions can be statically verified by an abstract interpretation that calculates a form of strongest postcondition, i.e., the arity and types of values on the operand stack following the execution of the sequence. This means that, if a function is given the right number of boxed arguments, then it will successfully execute and return values of the computed types. The strongest postcondition of an instruction takes a stack of types as input and calculates the stack of types resulting from executing that instruction.[3]

Two corresponding program states from Inca and Ubx simulate each other (expressed by the $\overset{\text{I-U}}{\sim}$ binary relation) if they have the same dynamic memory, if both their function environments and call stacks are similar, and if an abstract interpretation of all function definitions succeeds.

Two function environments are similar if they have the same domain and if, given a function definition in Ubx and in Inca, the Ubx function definition generalizes to the Inca function definition.

Call stacks are similar if (i) they have the same height; (ii) two corresponding stack frames refer to the same function, have the same program counters, and Ubx's operand stack may be boxed to Inca's; (iii) the abstract interpretation of the function up to the current program counter matches with the operand types in the stack; (iv) the current instruction of all caller stack frames must be a call instruction to the callees' stack frames.

[3] Note that the simple analysis used in this formalization cannot handle uses of the `CJump` instruction and, thus, can only interpret linear functions. Using a more complete abstract interpretation to enable more interesting functions is left for future work.

$$\to_{\textsc{Ubx}}\text{-}\textsc{Push}\ \dfrac{\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{Push}\ d}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ \Sigma \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H,\ \texttt{Frame}\ f\ (1 + pc)\ (\texttt{UbxDyn}\ d \cdot \Sigma) \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Push-Ubx}_1\ \dfrac{\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{PushUbx}_1\ u_1}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ \Sigma \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H,\ \texttt{Frame}\ f\ (1 + pc)\ (\texttt{UbxUbx}_1\ u_1 \cdot \Sigma) \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Push-Ubx}_2\ \dfrac{\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{PushUbx}_2\ u_2}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ \Sigma \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H,\ \texttt{Frame}\ f\ (1 + pc)\ (\texttt{UbxUbx}_2\ u_2 \cdot \Sigma) \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Load}\ \dfrac{\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{Load}\ x \qquad \texttt{cast}_{\texttt{Dyn}}\ u = \texttt{Some}\ d_1 \qquad \mathit{MemGet}\ H\ (x, d_1) = \texttt{Some}\ d_2}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ (u \cdot \Sigma) \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H,\ \texttt{Frame}\ f\ (1 + pc)\ (\texttt{UbxDyn}\ d_2 \cdot \Sigma) \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Load-Ubx-Hit}\ \dfrac{\begin{array}{c}\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{LoadUbx}\ \tau\ x\\ \texttt{cast}_{\texttt{Dyn}}\ u_1 = \texttt{Some}\ d_1 \qquad \mathit{MemGet}\ H\ (x, d_1) = \texttt{Some}\ d_2 \qquad \texttt{unbox}\ \tau\ d_2 = \texttt{Some}\ u_2\end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ (u_1 \cdot \Sigma) \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H,\ \texttt{Frame}\ f\ (1 + pc)\ (u_2 \cdot \Sigma) \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Load-Ubx-Miss}\ \dfrac{\begin{array}{c}\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{LoadUbx}\ \tau\ x\\ \texttt{cast}_{\texttt{Dyn}}\ u_1 = \texttt{Some}\ d_1 \qquad \mathit{MemGet}\ H\ (x, d_1) = \texttt{Some}\ d_2 \qquad \texttt{unbox}\ \tau\ d_2 = \texttt{None} \qquad F' = \mathit{FunAdd}\ F\ f\ (\texttt{generalize}\ fd)\end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ (u_1 \cdot \Sigma) \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F',\ H,\ \texttt{box\_stack}\ f\ (\texttt{Frame}\ f\ (1 + pc)\ (\texttt{UbxDyn}\ d_2 \cdot \Sigma) \cdot st)\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Store}\ \dfrac{\begin{array}{c}\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{Store}\ x\\ \texttt{cast}_{\texttt{Dyn}}\ u_1 = \texttt{Some}\ d_1 \qquad \texttt{cast}_{\texttt{Dyn}}\ u_2 = \texttt{Some}\ d_2 \qquad H' = \mathit{MemAdd}\ H\ (x, d_1)\ d_2\end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ (u_1 \cdot u_2 \cdot \Sigma) \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H',\ \texttt{Frame}\ f\ (1 + pc)\ \Sigma \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Store-Ubx}\ \dfrac{\begin{array}{c}\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{StoreUbx}\ \tau\ x\\ \texttt{cast}_{\texttt{Dyn}}\ u_1 = \texttt{Some}\ d_1 \qquad \texttt{cast\_box}\ \ \tau\ u_2 = \texttt{Some}\ d_2 \qquad H' = \mathit{MemAdd}\ H\ (x, d_1)\ d_2\end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ (u_1 \cdot u_2 \cdot \Sigma) \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H',\ \texttt{Frame}\ f\ (1 + pc)\ \Sigma \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Op}\ \dfrac{\begin{array}{c}\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{Op}\ op\\ ar = \mathit{Arith}\ op \qquad ar \le |\Sigma| \qquad \texttt{traverse}\ \texttt{cast}_{\texttt{Dyn}}\ \ (\texttt{take}\ ar\ \Sigma) = \texttt{Some}\ \Sigma' \qquad \mathit{Inl}\ op\ \Sigma' = \texttt{None} \qquad \mathit{Op}\ op\ \Sigma' = d\end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ \Sigma \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H,\ \texttt{Frame}\ f\ (1 + pc)\ (\texttt{UbxDyn}\ d \cdot \texttt{drop}\ ar\ \Sigma) \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Op-Inl}\ \dfrac{\begin{array}{c}\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{Op}\ op\\ ar = \mathit{Arith}\ op \qquad ar \le |\Sigma| \qquad \texttt{traverse}\ \texttt{cast}_{\texttt{Dyn}}\ \ (\texttt{take}\ ar\ \Sigma) = \texttt{Some}\ \Sigma' \qquad \mathit{Inl}\ op\ \Sigma' = \texttt{Some}\ op_{inl} \qquad \mathit{InlOp}\ op_{inl}\ \Sigma' = d\\ F' = \mathit{FunAdd}\ F\ f\ (\texttt{rewrite}\ fd\ pc\ (\texttt{OpInl}\ op_{inl}))\end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ \Sigma \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F',\ H,\ \texttt{Frame}\ f\ (1 + pc)\ (\texttt{UbxDyn}\ d \cdot \texttt{drop}\ ar\ \Sigma) \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Op-Inl-Hit}\ \dfrac{\begin{array}{c}\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{OpInl}\ op_{inl}\\ op = \mathit{Inl}^{-1}\ op_{inl} \qquad ar = \mathit{Arith}\ op \qquad ar \le |\Sigma| \qquad \texttt{traverse}\ \texttt{cast}_{\texttt{Dyn}}\ \ (\texttt{take}\ ar\ \Sigma) = \texttt{Some}\ \Sigma'\\ \mathit{IsInl}\ op_{inl}\ \Sigma' \qquad \mathit{InlOp}\ op_{inl}\ \Sigma' = d\end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ \Sigma \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H,\ \texttt{Frame}\ f\ (1 + pc)\ (\texttt{UbxDyn}\ d \cdot \texttt{drop}\ ar\ \Sigma) \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Op-Inl-Miss}\ \dfrac{\begin{array}{c}\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{OpInl}\ op_{inl}\\ op = \mathit{Inl}^{-1}\ op_{inl} \qquad ar = \mathit{Arith}\ op \qquad ar \le |\Sigma| \qquad \texttt{traverse}\ \texttt{cast}_{\texttt{Dyn}}\ \ (\texttt{take}\ ar\ \Sigma) = \texttt{Some}\ \Sigma'\\ \neg\ \mathit{IsInl}\ op_{inl}\ \Sigma' \qquad \mathit{InlOp}\ op_{inl}\ \Sigma' = d \qquad F' = \mathit{FunAdd}\ F\ f\ (\texttt{rewrite}\ fd\ pc\ (\texttt{Op}\ op))\end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ \Sigma \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F',\ H,\ \texttt{Frame}\ f\ (1 + pc)\ (\texttt{UbxDyn}\ d \cdot \texttt{drop}\ ar\ \Sigma) \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Op-Ubx}\ \dfrac{\begin{array}{c}\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{OpUbx}\ op_{ubx}\\ op = \mathit{Inl}^{-1}\ (\mathit{Ubx}^{-1}\ op_{ubx}) \qquad ar = \mathit{Arith}\ op \qquad ar \le |\Sigma| \qquad \mathit{UbxOp}\ op_{ubx}\ (\texttt{take}\ ar\ \Sigma) = \texttt{Some}\ u\end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ \Sigma \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H,\ \texttt{Frame}\ f\ (1 + pc)\ (u \cdot \texttt{drop}\ ar\ \Sigma) \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{CJump-True}\ \dfrac{\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{CJump}\ n \qquad \texttt{cast}_{\texttt{Dyn}}\ u = \texttt{Some}\ d \qquad \mathit{IsTrue}\ d}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ u \cdot \Sigma \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H,\ \texttt{Frame}\ f\ n\ \Sigma \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{CJump-False}\ \dfrac{\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{CJump}\ n \qquad \texttt{cast}_{\texttt{Dyn}}\ u = \texttt{Some}\ d \qquad \mathit{IsFalse}\ d}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ u \cdot \Sigma \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H,\ \texttt{Frame}\ f\ 1 + pc\ \Sigma \cdot st\rangle}$$

$$\to_{\textsc{Ubx}}\text{-}\textsc{Fun-Call}\ \dfrac{\begin{array}{c}\mathit{FunGet}\ F\ f = \texttt{Some}\ fd \qquad pc < |\texttt{body}\ fd| \qquad \texttt{body}\ fd\ !\ pc = \texttt{Call}\ g\\ \mathit{FunGet}\ F\ g = \texttt{Some}\ gd \qquad ar = \mathit{Arity}\ gd \qquad ar \le |\Sigma| \qquad \texttt{list\_all}\ (\lambda u.\ \texttt{typeof}\ u = \texttt{None})\ (\texttt{take}\ ar\ \Sigma)\end{array}}{\langle F,\ H,\ \texttt{Frame}\ f\ pc\ \Sigma \cdot st\rangle\ \to_{\textsc{Ubx}}\ \langle F,\ H,\ \texttt{Frame}\ g\ 0\ (\texttt{take}\ ar\ \Sigma) \cdot \texttt{Frame}\ f\ pc\ \Sigma \cdot st\rangle}$$

Figure 8. The subset of the ($\to_{\textsc{Ubx}}$) transition relation that differs from ($\to_{\textsc{Inca}}$).

We proved that $\overset{\text{I-U}}{\sim}$ is a bisimulation.

Lemma 6 (Forward simulation). *If $s_1 \rightarrow_{\textsc{Inca}} s_1'$ and $s_1 \overset{I\text{-}U}{\sim} s_2$, then there exists a state s_2' such that $s_2 \rightarrow_{\textsc{Ubx}} s_2'$ and $s_1' \overset{I\text{-}U}{\sim} s_2'$.*

Lemma 7 (Forward matching final states). *If $s_1 \overset{I\text{-}U}{\sim} s_2$ and $\mathtt{final}_{\textsc{Inca}}\ s_1$, then $\mathtt{final}_{\textsc{Ubx}}\ s_2$.*

Lemma 8 (Backward simulation). *If $s_2 \rightarrow_{\textsc{Ubx}} s_2'$ and $s_1 \overset{I\text{-}U}{\sim} s_2$, then there exists a state s_1' such that $s_1 \rightarrow_{\textsc{Inca}} s_1'$ and $s_1' \overset{I\text{-}U}{\sim} s_2'$.*

Lemma 9 (Backward matching final states). *If $s_1 \overset{I\text{-}U}{\sim} s_2$ and $\mathtt{final}_{\textsc{Ubx}}\ s_2$, then $\mathtt{final}_{\textsc{Inca}}\ s_1$.*

6.3 Compilation from Inca to Ubx

The process of compiling has three steps.

1. Lift the program from Inca to Ubx.
2. Optimize the program by using as many Ubx instructions as possible.
3. Ensure that the result is valid with respect to the abstract interpretation.

The optimization pass is based on an oracle—an abstract function of type *'fun ⇒ nat ⇒ type option*—which, given the position of a `Load` instruction in a function, evaluates to the expected unboxed type of the loaded value. A variant of the abstract interpretation used for the simulation relation optimizes instructions in a linear pass based on the following type information.

1. All function parameters have boxed dynamic types.
2. The type produced by `Push` is provided by inspecting the constant.
3. The type produced by `Load` is provided by the oracle, or assumed to be a boxed dynamic type if the oracle evaluates to `None`.
4. The type consumed by `Store` is obtained from the abstract interpretation.
5. The types consumed and produced by `Op`, `OpInl`, and `Call` are always boxed dynamic and their number depends on the arity of the operation or function.
6. The types consumed and produced by `OpUbx` is obtained from *TypeOf*.

This information provided by the oracle could either be given directly by the programmer or be the result of automatic run-time instrumentation. In the second case, the virtual machine would first execute code in Inca mode and gather some statistics on encountered types, a stage usually referred to as *profiling*. When some heuristics indicate that a point of "dynamic locality of type usage" is reached, the program would then be compiled to Ubx, and the control flow diverted to Ubx's execution engine.

The accuracy of the oracle's predictions may increase or decrease run-time performance, but may never alter the semantics of the executed program. If a value loaded from memory does not match the oracle's prediction, rule $\rightarrow_{\textsc{Ubx}}$-Load-Ubx-Miss generalizes the function back to cope with boxed values before resuming the execution.

We proved that compiled, optimized programs simulate their uncompiled counterparts.

Lemma 10 (Compiled matching states). *If $\mathtt{compile}\ p_1 = \mathtt{Some}\ p_2$ and $\mathtt{load}_{\textsc{Inca}}\ p_1\ s_1$, then there exists a state s_2 such that $\mathtt{load}_{\textsc{Ubx}}\ p_2\ s_2$ and $s_1 \overset{I\text{-}U}{\sim} s_2$.*

Lemmas 6 to 10 imply that the successful execution of a compiled Ubx program exhibits identical behavior to the execution of the original Inca program. Formally:

Theorem 3 (Soundness of compilation). *Let the infix relation $\Downarrow$ pair a program to its run-time behavior and the infix relation $\approx$ be an equivalence relation between behaviors. If $\mathtt{compile}\ p_1 = \mathtt{Some}\ p_2$, and $p_2 \Downarrow b_2$, and b_2 does not go wrong, then there exists a behavior b_1 such that $p_1 \Downarrow b_1$ and $b_1 \approx b_2$.*

Compilation from Inca to Ubx is incomplete in the sense of Theorem 2 because (i) the abstract interpretation and optimization algorithm are too simplistic to handle jumps, and (ii) the hypothesis is too weak.

The former issue can be addressed by using a more sophisticated data-flow analysis instead of our one-pass linear analysis. The latter issue is that the process of loading does not guarantee a successful executions. To address this issue and prove completeness, the hypothesis needs to be strengthened, for example, by giving a typing judgment that guarantees a valid execution of the initial program.

7 Practical Perspective

A Brief History The original idea on how to optimize dynamically-typed programming languages with advanced type specialization in a purely interpretative setting is about ten years old by now. The senior author implemented a full-fledged prototype in CPython 3.3, reporting speedups by a factor of up to 5. At the same time, the prototype retained traditional interpreter benefits: simplicity and ease-of-implementation. Papers were submitted to ACM SIGPLAN PLDI 2013, SIGPLAN PLDI 2014, and ACM Transactions on Architecture and Code Optimization 2014, with usually positive feedback but no PC member championing the paper. Due to the important speedups enabled by the prototype, a series of talks were given in 2012: TU Wien, Universität Linz, IST Austria, and a talk at Mozilla.

Benefits of Formalization The full-fledged CPython prototype successfully passed all relevant unit tests and ran major Python applications, benchmarks, and frameworks. Although tests covered several ten thousands lines of Python programs and C code for libraries, some "Heisenbugs" occurred every now and then. Through the presented formalization, we were able to discern a new requirement that addressed the bug.

The new requirement—obvious in hindsight, but non-obvious before—is due to the deoptimization of Ubx optimized code. When deoptimizing a certain function `f`, a prior, yet incomplete call to function `f` may still be active on stack. Assume the prior stack frame of `f` was type-specialized to a specific type T and that the operand stack of the interpreter stack frame contained unboxed data of type T. If we deoptimize the newer stack frame of the present function invocation of `f`, then all unboxed data will be boxed again and stored in memory. Now, assume that during a following call of function `f`, it will be optimized again, but to a different type T'. The program continues, until it eventually continues to operate on the prior stack frame belonging to function `f`. The interpreter operand stack may now hold unboxed data of type T, but the optimized instructions will assume the data to be of type T'. Potential errors following from this situation are: (i) deoptimization may fail when the types T and T' differ; (ii) execution of native-machine operations may fail, when the data representation differs; (iii) (un-)boxing of data may fail, when we try to access native-machine data incorrectly.

The underlying problem is that there is only one optimized interpreter code image stored for each interpreted function. A Ubx function is, therefore, not able to infer potential changes to its code. A variety of techniques address this issue, e.g., deoptimizing *all* invocations of the optimized code, or keeping a version counter of the code image and check, that these are identical.

Evaluation Results Table 1 presents speedup factors relative to a baseline interpreter: CPython 3.3.2 using switch-dispatch for instruction dispatch. The PyPy3 measurements correspond to the then (2014) most recent version: 2.1 beta 1. At present, PyPy3 is out of beta and offers a better performance profile and better compatibility with C extensions.

We evaluated our full-fledged implementation using the following benchmarks. First, we use the following micro-benchmarks from the computer language benchmarks game [15]: binarytrees, mandelbrot, nbody, and spectralnorm. Second, we used a set of publicly available solutions to the first 50 Project Euler problems [1], where we selected programs that show a longer than average run-time (solutions to problems no. 27, 31, 39, and 50).

The benchmarks were run on an Intel Nehalem i7-920 running at a frequency of 2.67 GHz, on Linux kernel version 3.11.0-15 and `gcc` version 4.6.4. To minimize perturbations by third party systems, we took the following precautions. First, we disabled Intel's TurboBoost [25] feature to avoid frequency scaling based on unknown heuristics. Second, we used `nice -n -20` to minimize operating system scheduler effects. Third, we used 30 repetitions for each pairing of a benchmark with an interpreter to get stable results; we report the geometric mean of these repetitions, to account for outliers.

Table 1. Speedups of PyPy3 and Ubx-Prototype over the CPython baseline.

Benchmark	PyPy3	Ubx-Prototype
binarytrees	1.8225×	1.7081×
mandelbrot	0.9403×	2.0986×
nbody	1.5112×	3.7010×
spectralnorm	2.7900×	4.4012×
E27	3.8519×	2.3084×
E31	0.1460×	1.1393×
E39	1.9461×	4.9297×
E50	3.6531×	3.8018×
Geometric Mean	1.6984×	2.5367×

8 Related Work

To the best of our knowledge, there exists no prior work that is directly related to the formalization and verification of the speculative optimizations presented here. We therefore group the related work into the three most directly related groups of related work: (i) formalization and verification of translators, (ii) formalization and verification of dynamic languages, and (iii) just-in-time compiler optimizations.

8.1 Formalization and Verification of Translators

We combine the related work on compilers, just-in-time compilers, and interpreters and subsume all of them under the label "translators." From a history perspective, the correctness of translators has been an active research area since at least the 1980s. The topic of compiler correctness has, for instance, been examined in the European FP2 research program ProCoS [22]. The findings of ProCoS subsequently lead to a larger German research project called Verifix, which examined several aspects of compiler correctness [16, 17]. In the 2000s, a group of researchers in France pioneered the field by mechanizing correctness of an industrial-strength C compiler [5, 30, 38, 42–44]. In the 2010s, a mechanized formalization and verification of ML followed [28].

In 2006, Klein and Nipkow formalized Jinja, a unified model of a Java-like source language, virtual machine, and compiler [26, 27]. Lochbihler later added support for interleaved execution of threads with JinjaThreads [31–35]. In 2018, Watt mechanized the WebAssembly specification [46, 47].

In a similar vein, the verification of compile-time optimizations has received considerable attention from the research community. VellVM, for example, focused on verifying optimizations on the LLVM bitcode intermediate representation [52]. Tatlock and Lerner simplify the verification of optimizations in verified compilers by using SMT solvers to aid with the construction of verified translation validators [41] Prior research also focused on the formalization verification of intermediate representations, such as Java bytecode, without optimizations [29, 40].

In 2010, Myreen presents his work on the formalization and verification of just-in-time compilers [36], documenting some of the difficulties posed by self-modifying code. This paper is most directly related prior work, but addresses a different direction, namely, the formalization of just-in-time compilers. Our work, however, sidesteps the intricate difficulties of JIT compilers by focusing on optimizing interpreters instead. In 2017, Flückiger et al. investigate the correctness of speculative optimizations with dynamic deoptimiziation [14]. Since our virtual machine interpreters can be thought of as intermediate representations, the Inca language confirms the finding by Flückiger et al., namely, that reasoning about complex system interactions is a lot easier by embedding the proper information in it. However, Ubx goes further than Flückiger et al. by covering different data representation.

8.2 Formalization and Verification of Dynamic Languages

The formalization of dynamic languages in general, and JavaScript in particular, has been the subject of substantial prior work. In 2010, λ_{JS} presented the first executable, formal semantics of JavaScript [21]. By rewriting JavaScript surface syntax into equivalent Scheme code, JavaScript programs could be executed, with correctness and security guarantees depending on the underlying Scheme system. In 2013, λ_{π} applied a similar technique to provide a formal semantics for Python [37]. A comprehensive formalization and verification effort of JavaScript is the Coq-based project JSCert [6]. JSCert generates a verified JavaScript interpreter from its formalization.

While a formal semantics is an indispensable prerequisite for a correct and verified virtual machine, it addresses the desirable performance aspect insufficiently. To attain performance, a formalization of speculative optimizations is required, which is the key contribution of our paper.

8.3 Just-in-Time Compilers & Interpreters

Aycock gives a good overview of the history of just-in-time compilers up until the early 2000s [2]. Particularly relevant prior work is the original work by Deutsch and Schiffman, which introduced the seminal idea of inline caching [12]. Originally, their work on Smalltalk 80 systems considered so-called *monomorphic* inline caches, i.e., inline caches that hold at most one address. Hölzle, Chambers and Ungar subsequently extended these with so-called polymorphic inline caches, i.e., a combination of an inline cache and a stub to cache multiple target addresses, which is particularly relevant in highly polymorphic call sites [23, 24].

In 1996, Roemer et al. studied the performance of interpreters and found no specific evidence to identify hints [39]. In 2003, Ertl and Gregg investigated the performance of interpreters again and found evidence of the importance of branch predictors [13]. In 2009, Brunthaler analyzed the varying performance potential of interpreter optimizations and found that the interpreter abstraction level is the primary performance determinant for selecting interpreter optimizations [7]. In 2010, Brunthaler investigated the use of inline caching in a purely interpretative fashion, in contrast to its use in just-in-time compilers, and found speedups by a factor of up to 2 [8, 9]. In 2012, Würthinger generalized Brunthaler's bytecode interpreter optimizations to abstract-syntax tree interpreters [50], which subsequently became the cornerstone for the development of the Truffle/Graal virtual machine implementation efforts [48, 49]. In 2014, Wang et al. demonstrated the potential of combining advanced optimizations in the R programming language and reported speedups of up to 3.5 [45].

All prior work in this area reports important speedups, either through dynamic code generation in a classic just-in-time compiler setting, or by way of optimizing interpreters. The exclusive focus of prior work is on improving performance, or sometimes also reducing memory footprint. The aspect of formalization and verification, in particular to establish correctness, is notably absent.

9 Conclusion

We presented a formalization of virtual machine interpreters for dynamically typed programming languages. Our formalization define an interpreter supporting the most representative features found and used by many virtual machine interpreters for mainstream languages. We then methodically extend the virtual machine interpreter's instruction set and semantics to accommodate increasingly specialized and optimized instruction derivatives. These incrementally specialized derivatives eliminate much of the overhead frequently found in high abstraction-level virtual machines, such as those used by Python or JavaScript.

The optimized instruction derivatives, in particular, first eliminate the overhead of dynamic typing by inline caching a prior recorded type at its place. This recorded type information is subsequently used to expand the local knowledge of type usage in a specific region of the program, e.g., a loop, or a basic block. Once a suitable region of known types is determined, we can rewrite the whole sequence to eliminate the overhead of using boxed objects by using native-machine data representation instead.

Our formalization enables the proof of both, soundness and completeness, for speculative optimizations. Given a formal semantics of a dynamic language, and a suitable intermediate representation, our formalization provides a systematic way to (i) integrate speculative optimizations, and (ii) establish the correctness of the resulting system. We believe that our formalization provides a foundation for the verification of industrial-strength implementations. These implementations will benefit from our formalization's ability to pinpoint subtle errors and non-obvious requirements.

Acknowledgments

The authors thank Jasmin Blanchette and Johannes Kinder for invaluable feedback on earlier versions of this paper. This publication is part of the project Concordia, a project that has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 830927.

References

[1] S. Anand. 2011. Project Euler Solutions. Retrieved February 18th, 2014 from http://www.s-anand.net/euler.html

[2] John Aycock. 2003. A brief history of just-in-time. *Comput. Surveys* 35, 2 (2003), 97–113. https://doi.org/10.1145/857076.857077

[3] Scott B Baden. 1982. *High Performance Storage Reclamation in an Object-Based Memory System.* Technical Report. University of California, Berkeley, Berkeley, CA, USA.

[4] Clemens Ballarin. 2014. Locales: A Module System for Mathematical Theories. *Journal of Automated Reasoning* 52, 2 (01 Feb 2014), 123–153. https://doi.org/10.1007/s10817-013-9284-7

[5] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal Verification of a C Compiler Front-End. 460–475. https://doi.org/10.1007/11813040_31

[6] Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA). Association for Computing Machinery, New York, NY, USA, 87–100. https://doi.org/10.1145/2535838.2535876

[7] Stefan Brunthaler. 2009. Virtual-Machine Abstraction and Optimization Techniques. *Electronic Notes in Theoretical Computer Science* 253, 5 (2009), 3–14.

[8] Stefan Brunthaler. 2010. Efficient interpretation using quickening.

[9] Stefan Brunthaler. 2010. Inline caching meets quickening. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 6183 LNCS. 429–451.

[10] Martin Desharnais. 2020. A Generic Framework for Verified Compilers. *Archive of Formal Proofs* (Feb. 2020). https://isa-afp.org/entries/VeriComp.html, Formal proof development.

[11] Martin Desharnais. 2020. Inline Caching and Unboxing Optimization for Interpreters. *Archive of Formal Proofs* (Dec. 2020). https://isa-afp.org/entries/Interpreter_Optimizations.html, Formal proof development.

[12] L Peter Deutsch and Allan M Schiffman. 1984. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '84*. ACM Press, New York, New York, USA, 297–302. https://doi.org/10.1145/800017.800542

[13] M Anton Ertl and David Gregg. 2003. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-Level Parallelism* 5 (nov 2003), 1–25.

[14] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2017. Correctness of Speculative Optimizations with Dynamic Deoptimization. *Proc. ACM Program. Lang.* 2, POPL, Article 49 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158137

[15] Brent Fulgham. 2013. The Computer Language Benchmarks Game. Retrieved April 25th, 2013 from http://shootout.alioth.debian.org/

[16] Sabine Glesner, G. Goos, F. v. Henke, H. Langmaack, W. Goerigk, and W. Zimmermann. 2004. *Abschlussbericht Verifix.* Technical Report. Universitäten Karlsruhe, Kiel, Ulm.

[17] Gerhard Goos and Wolf Zimmermann. 1999. Verification of Compilers. In *Correct System Design: Recent Insights and Advances.* Lecture Notes in Computer Science, Vol. 1710. 201–230. https://doi.org/10.1007/3-540-48092-7_10

[18] Samuel Groß. 2020. JITSploitation I: A JIT Bug. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html Accessed: 2020-09-21.

[19] Samuel Groß. 2020. JITSploitation II: Getting Read/Write. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-two.html Accessed: 2020-09-21.

[20] Samuel Groß. 2020. JITSploitation III: Subverting Control Flow. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-three.html Accessed: 2020-09-21.

[21] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. 126–150. https://doi.org/10.1007/978-3-642-14107-2_7

[22] Michael G. Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog (Eds.). 2017. *Provably Correct Systems.* Springer. https://doi.org/10.1007/978-3-319-48628-4

[23] Urs Hölzle, Craig Chambers, and David M Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the 5th European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991 (ECOOP~'91) (Lecture Notes in Computer Science, Vol. 512/1991).* Springer-Verlag, 21–38.

[24] Urs Hölzle and David Ungar. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation - PLDI '94*. ACM Press, New York, New York, USA, 326–336. https://doi.org/10.1145/178243.178478

[25] Intel. 2012. Intel Turbo Boost Technology – On-Demand Processor Performance. Retrieved April 25th, 2013 from http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

[26] Gerwin Klein and Tobias Nipkow. 2005. Jinja is not Java. *Archive of Formal Proofs* (June 2005). https://isa-afp.org/entries/Jinja.html, Formal proof development.

[27] Gerwin Klein and Tobias Nipkow. 2006. A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.* 28, 4 (July 2006), 619–695. https://doi.org/10.1145/1146809.1146811

[28] Ramana Kumar, MO Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A verified implementation of ML. *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* 1, Section 3 (2014), 179–191. https://doi.org/10.1145/2535838.2535841

[29] Xavier Leroy. 2003. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning* 30, 3/4 (2003), 235–269. https://doi.org/10.1023/A:1025055424017

[30] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (dec 2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[31] Andreas Lochbihler. 2007. Jinja with Threads. *Archive of Formal Proofs* (Dec. 2007). https://isa-afp.org/entries/JinjaThreads.html, Formal proof development.

[32] Andreas Lochbihler. 2008. Type Safe Nondeterminism - A Formal Semantics of Java Threads. In *International Workshop on Foundations of Object-Oriented Languages (FOOL 2008)*. http://www.infsec.ethz.ch/people/andreloc/publications/lochbihler08fool.pdf

[33] Andreas Lochbihler. 2010. Verifying a Compiler for Java Threads. In *European Symposium on Programming (ESOP'10) (LNCS, Vol. 6012)*, A. D. Gordon (Ed.). Springer, 427–447. https://doi.org/10.1007/978-3-642-11957-6_23

[34] Andreas Lochbihler. 2012. Java and the Java Memory Model – a Unified, Machine-Checked Formalisation. In *Programming Languages and Systems (LNCS, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 497–517. https://doi.org/10.1007/978-3-642-28869-2_25

[35] Andreas Lochbihler and Lukas Bulwahn. 2011. Animating the Formalised Semantics of a Java-like Language. In *Interactive Theorem Proving (LNCS, Vol. 6898)*, Marko van Eekelen, Herman Geuvers, Julien Schmalz, and Freek Wiedijk (Eds.). Springer, 216–232. https://doi.org/10.1007/978-3-642-22863-6_17

[36] Magnus O Myreen. 2010. Verified just-in-time compiler on x86. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '10*. ACM Press, New York, New York, USA, 107. https://doi.org/10.1145/1706299.1706313

[37] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 217–232. https://doi.org/10.1145/2509136.2509536

[38] Silvain Rideau and Xavier Leroy. 2010. Validating Register Allocation and Spilling. 224–243. https://doi.org/10.1007/978-3-642-11970-5_13

[39] Theodore H Romer, Dennis Lee, Geoffrey M Voelker, Alec Wolman, Wayne A Wong, Jean-Loup Baer, Brian N Bershad, and Henry M Levy. 1996. The structure and performance of interpreters. In *Asplos*. ACM Press, 150–159.

[40] Robert F. Stärk, Joachim Schmid, and Egon Börger. 2001. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer.

[41] Zachary Tatlock and Sorin Lerner. 2010. Bringing Extensibility to Verified Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vol. 45. 111–121. https://doi.org/10.1145/1809028.1806611

[42] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal verification of translation validators. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '08*. ACM Press, New York, New York, USA, 17. https://doi.org/10.1145/1328438.1328444

[43] Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified validation of lazy code motion. 316–326. https://doi.org/10.1145/1542476.1542512

[44] Jean-Baptiste Tristan and Xavier Leroy. 2010. A simple, verified validator for software pipelining. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '10*. ACM Press, New York, New York, USA, 83. https://doi.org/10.1145/1706299.1706311

[45] Haichuan Wang, Peng Wu, and David Padua. 2014. Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-Level Specialization. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 295–305. https://doi.org/10.1145/2544137.2544153

[46] Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 53–65. https://doi.org/10.1145/3167082

[47] Conrad Watt. 2018. WebAssembly. *Archive of Formal Proofs* (April 2018). https://isa-afp.org/entries/WebAssembly.html, Formal proof development.

[48] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

[49] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[50] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587

[51] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2012. Finding and understanding bugs in C compilers. *ACM SIGPLAN Notices* 47, 6 (2012), 283. https://doi.org/10.1145/2345156.1993532

[52] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 427–440. https://doi.org/10.1145/2103656.2103709