Verified Interpreters for Dynamic Languages with Applications to the Nix Expression Language

RUTGER BROEKHOFF, Radboud University Nijmegen, The Netherlands
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands

To study the semantics of a programming language, it is useful to consider different specification forms—*e.g.*, a substitution-based small-step operational semantics and an environment-based interpreter—because they have mutually exclusive benefits. Developing these specifications and proving correspondences is challenging for 'dynamic'/'scripting' languages such as JavaScript, PHP and Bash. We study this challenge in the context of the Nix expression language, a dynamic language used in the eponymous package manager and operating system. Nix is a Turing-complete, untyped functional language designed for the manipulation of JSON-style attribute sets, with tricky features such as overloaded use of variables for lambda bindings and attribute members, subtle shadowing rules, a mixture of evaluation strategies, and tricky mechanisms for recursion.

We show that our techniques are applicable beyond Nix by starting from the call-by-name lambda calculus, which we extend to a core lambda calculus with dynamically computed variable names and dynamic binder names, and finally to Nix. Our key novelty is the use of a form of *deferred substitutions*, which enables us to give a concise substitution-based semantics for dynamic variable binding. We develop corresponding environment-based interpreters, which we prove to be sound and complete (for terminating, faulty and diverging programs) w.r.t. our operational semantics based on deferred substitutions.

We mechanize all our results in the Rocq prover and showcase a new feature of the Rocq-std++ library for representing syntax with maps in recursive positions. We use Rocq's extraction mechanism to turn our Nix interpreter into executable OCaml code, which we apply to the official Nix language tests. Altogether this gives rise to the most comprehensive formal semantics for the Nix expression language to date.

CCS Concepts: • **Theory of computation → Semantics and reasoning**.

Additional Key Words and Phrases: Interpreters, substitution, lambda calculus, Nix, Rocq

ACM Reference Format:
Rutger Broekhoff and Robbert Krebbers. 2025. Verified Interpreters for Dynamic Languages with Applications to the Nix Expression Language. *Proc. ACM Program. Lang.* 9, ICFP, Article 268 (August 2025), 30 pages. https://doi.org/10.1145/3747537

1 Introduction

Mechanized specifications of programming languages often come with multiple specification forms because these have mutually exclusive benefits. For example, mechanized specifications of C [12, 31, 35], LLVM [55], Java [38] and JavaScript [8] come with both an operational semantics and an interpreter/executable semantics. An operational semantics is a good fit for verification (of *e.g.*, type soundness, compilers, program logics) due to its mathematical/abstract nature. An interpreter is closer to a language implementation and hence inherently less mathematical/abstract, but has the benefit of enabling one to exercise the language specification on 'real-life' tests.

Another trade-off is whether to use substitution or environments. A substitution-based semantics is more concise (particularly because one does not have to model closures explicitly) but is

Authors' Contact Information: Rutger Broekhoff, rutger@fautchen.eu, Radboud University Nijmegen, The Netherlands; Robbert Krebbers, mail@robbertkrebbers.nl, Radboud University Nijmegen, The Netherlands.

ACM 2475-1421/2025/8-ART268
https://doi.org/10.1145/3747537

computationally inefficient. This trade-off becomes more evident for functional languages, and even more so for lazy/call-by-name languages where every expression is a thunk/closure in an environment-based setting. To obtain the best of both worlds, our goal is to verify soundness and completeness of environment-based interpreters w.r.t. substitution-based operational semantics.

We consider 'dynamic'/'scripting' languages, with a focus on the Nix expression language, which is a cornerstone of the eponymous package manager and operating system. Nix has a strong focus on reproducibility, so studying its semantics is a valuable goal. More broadly, Nix has many challenging features that have not (or only partly) been covered in prior work. Similar to other dynamic languages (such as JavaScript, PHP, and Bash), Nix has *dynamic binding*. But we also consider Nix's subtle shadowing rules and its mixture of evaluation strategies (shallow and deep). For some of these features it is not even clear how to write a concise operational semantics or interpreter in the very first place. To make sure that the semantics of evaluation strategies is correct, it is crucial to prove soundness and completeness for faulty and diverging programs, in addition to terminating programs. On top of that, we address the challenge of finding a representation of the syntax and semantics that is amenable for mechanization in a proof assistant.

Main challenge: Dynamic binding. Most scripting languages have constructs to dynamically compute variables names or to dynamically introduce variable bindings. Bash and PHP have `${e}`, which gives the value of the variable denoted by the string expression `e`. To see this construct in action, consider the PHP program `$y = "x"; $x = 10; echo ${$y}`, which prints `10`. Nix has `with r; e` (which is similar to `extract` in PHP, and `with` in non-strict JavaScript) to introduce a variable binding for each member of the attribute set `r` in the scope of the expression `e`. For example, take the following Nix program (we indicate when we consider examples in other languages):

```
let r = { x = 10; y = 12; }; in with r; x + y
```

Here, the variable `r` is bound to the JSON-style attribute set with members `x` and `y`. Using the `with` construct, these members are turned into variables that can be used in the expression `x + y`. Hence, this program prints `22`. The situation becomes more complicated when the computed variable names or attribute sets are the result of a function call. Consider:

```
with g {}; x + y
```

This program is only *closed* (*i.e.,* every variable has a corresponding binder) if the attribute set computed by `g` contains members `x` and `y`. This example thus shows that the basic property of closedness—which arguably every program should satisfy—is not statically checkable in dynamic languages. Similarly, one cannot statically check if a variable is *shadowed*:

```
let r = { x = 10; y = 12; }; in with r; with g {}; x + y
```

Whether `x` and `y` refer to the members of the attribute set `r` depends on the attribute set returned by `g`. In other words, these examples indicate that the choice of names is relevant for program execution. The choice of names is also relevant in the presence of the notorious `eval` function (in *e.g.,* JavaScript and PHP) using which one can interpret strings as *open* programs. For example, the outcome of the JavaScript program `y = 12; eval(g()); console.log(y)` depends on string returned by `g`. If `g` returns `"y = 10"`, the value of `y` would be overwritten.

The fact that variable names matter at runtime impacts the choice of a formal representation. Commonly used representations include De Bruijn indices [15], nominal syntax [21], higher-order abstract syntax [48] and locally nameless [13]. The strength of these representations is that they abstract away the concrete names of bound variables, but that immediately makes them unsuitable for dynamic languages. In the examples above we see that the names of variables are relevant, so they need to be treated as strings. Consequently, most formal semantics of dynamic languages

(such as JavaScript and PHP) either use a string-based variable representation with environments or avoid dynamic constructs such as `$` and `with` (§ 7.3). A notable exception is the substitution-based semantics of Nix by Dolstra [16], but that does not handle variable binding correctly (§ 7.2).

It is well-known that string-based variable representations can be tedious due to α-conversion and variable capture. However, since the result of a functional program should never contain any free variables, it is folklore that variable capture problems can be avoided by restricting reduction to closed programs [47, §11.3.2].[1] Unfortunately, in the presence of dynamic variable binding, this trick does not work because we cannot statically determine if expressions are closed (see `with g {}; x + y` above, where closedness depends on the result of `g`). This brings us to the question: *can we define a concise substitution-based semantics for dynamic languages that is sound and complete w.r.t. an environment-based interpreter, and scales to non-trivial language features?*

Solution to main challenge: A form of deferred substitutions. Our key insight is to "defer" substitutions in dynamic constructs such as `$` and `eval`. Substitution should not happen right away, but only at the moment the operand of `$` or `eval` has been fully reduced to a string literal. We do so by taking inspiration from the calculus of explicit substitutions [1], which reifies substitutions as explicit constructs in the expression syntax. But instead of allowing explicit substitutions to appear anywhere in the expression syntax, we only allow them to occur at dynamic constructs such as `$` and `eval`. More formally, in our operational semantics, we annotate $\${e}\overline{d}$ with a *deferred substitution* $\overline{d}$, which is a finite map from strings to expressions. In the source program all deferred substitutions are empty. When reducing a β-redex, a new mapping is added to all enclosed deferred substitutions. For example, $(\lambda\, x.\ \ldots \${e_1}\overline{d} \ldots)\ e_2$ is reduced to $\ldots \${e_1'}\overline{d}\langle x \coloneqq e_2\rangle \ldots$, where e_1' is the result of substituting e_2 for x in e_1. (We substitute x in e_1 to support nested lookups, *e.g.*, `${${"x"}}`.) Then finally when e in $\${e}\overline{d}$ is reduced to a string x, we look up x in the deferred substitution $\overline{d}$. That is, $\${x}\overline{d}$ is reduced to e for $x \coloneqq e \in \overline{d}$. (In an unpublished manuscript, Lippmeier [37] proposes a different form of deferred substitutions, with other applications, see § 7.1.)

We show that deferred substitutions enjoy some good properties. They correctly handle variable capture without the need for α-conversion and without a restriction to closed expressions, they naturally support variable shadowing in the presence of `with`, and they can easily be adjusted to support different combinations of language features (*e.g.*, `with`, `eval`, `$`). To show the correctness of deferred substitutions, we prove soundness and completeness of an environment-based interpreter w.r.t. them. We moreover prove that for the call-by-name lambda calculus, our semantics is sound and complete w.r.t. an ordinary substitution-based semantics for closed expressions.

Sub-challenge #1: Subtle shadowing rules. Many languages have subtle shadowing rules, whose semantics are difficult to specify. Consider the following examples in Nix:

```
let x = 10; in     let x = 12; in     x   # returns 12
with { x = 10; }; with { x = 12; }; x   # returns 12
let x = 10; in     with { x = 12; }; x   # returns 10
with { x = 10; }; let x = 12; in     x   # returns 12
```

Naively one might expect the third program to return 12, because `x` is shadowed. However, in Nix `let` has priority over `with`, regardless of whether the `let` is enclosed in the `with` or not. These rules are a well-known source of confusion [11, 43, 54], and the prior semantics of Nix by Dolstra [16] (which uses ordinary substitutions) got these rules wrong (§ 7.2).

Deferred substitutions are a good fit to model such shadowing rules because substitutions are deferred to the moment that variables are reduced. More formally, we extend deferred substitutions

[1] This trick is used in practice in the Rocq-based textbook Programming Language Foundations [49] and the Rocq-based verification tools Iris [28, 33] and CFML [14].

to track whether each variable expression mapping belongs to a `with` or a `let`. If a substitution for a `let` is performed after one for a `with`, we simply overwrite it in the deferred substitution.

Sub-challenge #2: Mixture of evaluation strategies. Many lazy languages have a mixture of evaluation strategies. Nix has `deepSeq e1 e2`, which if `e1` results in a list or attribute set, evaluates all elements/members recursively, and then proceeds with `e2`. Getting the semantics of terminating and faulty programs correct is surprisingly subtle. Consider:

```
Omega = (x: x x) (x: x x)
deepSeq { x = Omega; y = 0 0; } 10
```

There are two possible behaviors: either the program faults (when the faulty application `0 0` is evaluated first) or diverges (when the loop `Omega` is evaluated first). The behavior depends on the order in which the members of the attribute set are evaluated. In Nix, it appears that numbers are assigned to attribute members, and attribute sets are processed in a deterministic order based on that numbering. To model this behavior correctly, we parameterize our operational semantics and interpreter by a total order on names,[2] which neatly aligns with our use of strings for names.

The equality operator == (which is primitive in Nix) also performs a deep evaluation to compare its operands. Interestingly, its semantics w.r.t. divergence is different from `deepSeq`. Consider:

```
{ x = Omega; } == { x = Omega; y = 10; }
```

One might expect the operands of == to be evaluated deeply and the program to diverge, but it will actually terminate. What happens is that Nix will first compare the set of attribute names, and only if these compare equal, proceed recursively to compare the attribute values.

These examples clearly emphasize the need to prove soundness and completeness not just for terminating programs, but also for faulty and diverging programs. More precisely, we wish to prove that the substitution-based semantics terminates, faults or diverges iff the interpreter has that same behavior. Up to our knowledge, this correspondence has never been proved in a proof assistant for even a basic version of an interpreter for the call-by-name lambda calculus.

Sub-challenge #3: Recursion through finite maps in Rocq. To mechanize a language in a proof assistant, suitable data structures to represent the syntax are important. A key challenge for us is finding a representation of finite maps that can be used in nested positions and provides the right reasoning principles. We need that because variables in expressions contain deferred substitutions, which are finite maps from strings to expressions themselves. Finite maps also occur in nested position to model attribute sets and thunks/closures. We make use of the recently improved `gmap` data structure from the Rocq-std++ library by the second author [32], which allows us to write:

```
Inductive expr :=
  | EId (ds : gmap string expr) (ex : expr)    // ${ex}_ds
  | ...
```

The `gmap` data structure is based on the canonical binary trie data structure by Appel and Leroy [4], but generalized to arbitrary keys (we use `string`). Similar to the data structure by Appel and Leroy [4], `gmap` has a number of important features. First, it enjoys extensional equality, *i.e.,* maps are equal iff they have the same value for every key (without axioms like functional extensionality or proof irrelevance), which makes reasoning in Rocq much more concise (*e.g.,* no need for setoid rewriting). Second, it enables efficient computation (the lookup, insert an delete operations have logarithmic time complexity, both with `vm_compute` in Rocq and extraction to OCaml). We demonstrate that the data structure is well-suited to represent syntax with nested occurrences of maps and allows us to

[2] Alternatively one could make the operational semantics non-deterministic, but since the interpreter needs to pick a concrete evaluation order, that would result in a weaker soundness and completeness theorem, see § 4.4.

define recursive definitions (*i.e.,* `Fixpoints`) without Rocq's guardedness checker being a burden. Neither Rocq nor Rocq-std++ automatically provide the necessary induction principles for our proofs, but we show that these can be easily derived.

Contributions. We develop techniques for developing sound and complete interpreters for dynamic languages using a form of *deferred substitutions*, which we put to practice to develop the most comprehensive semantics of the Nix expression language to date. Concretely:

- As the baseline of our work, we develop a soundness and completeness proof for an interpreter for the call-by-name lambda calculus, which accounts for terminating, faulty, and diverging programs (§ 2). While this result is likely folklore, we believe we are the first to present a mechanized proof in a proof assistant.
- We develop a form of deferred substitutions to give a substitution-based operational semantics for dynamic languages, which we apply to a core language with versions of `with`, `$` and `eval` (§ 3). We prove soundness and completeness of an environment-based interpreter w.r.t. our operational semantics based on deferred substitutions.
- We put deferred substitutions to practice to develop a large-scale operational semantics and interpreter for Nix (§ 4). The results in this section extend the prior semantics of Nix by Dolstra [16] with additional features (*e.g.,* `deepSeq`, `__functor`, matching with recursive defaults, deep comparison operator semantics, IEEE floats based on the Flocq library [9] in Rocq) and mechanized proofs in a proof assistant. We discovered bugs in the prior semantics of Nix related to dynamic binding and divergence, see § 7.2.
- We use Rocq's extraction mechanism [36] to turn our Nix interpreter into executable OCaml code, which combined with a frontend (parser and elaborator) allows us to exercise the interpreter on the official Nix language tests (§ 5).
- We demonstrate how the new `gmap` data structure from the Rocq-std++ library can be used to represent syntax with nested recursion through finite maps (§ 6).

We conclude with related (§ 7) and future work (§ 8). The Rocq and OCaml source code for all sections can be found in our artifact [10]. Hyperlinks to the Rocq development are marked (🐓).

Limitations. Although we believe that our techniques are general purpose, our core focus is on the Nix language. Other dynamic languages (such as JavaScript, PHP or Bash) have an abundance of other subtle features, and it remains to be investigated how our techniques can be transferred.

Similar to Dolstra [16], we use call-by-name whereas the official Nix implementation is lazy (*i.e.,* uses sharing). This means that there are some cases where we are not lazy enough (*e.g.,* cycle detection in recursive attributes) or are too inefficient to execute certain test programs. We also omit Nix features for interaction with the file system (*e.g.,* paths) and package manager (*e.g.,* derivations).

2 The Call-by-Name Lambda Calculus

We present a substitution-based operational semantics (§ 2.1) and environment-based interpreter (§ 2.2) for the call-by-name lambda calculus, called LambdaLang, which will serve as a baseline to present our form of deferred substitutions (§ 3) and our semantics of Nix (§ 4). We provide a proof of soundness and completeness of the interpreter w.r.t. the operational semantics for terminating, faulty and diverging programs (§ 2.3). While the proof might be folklore, we believe that we are the first to spell out the details and mechanize it in a proof assistant.

2.1 Syntax and Operational Semantics

The syntax of LambdaLang is given in Figure 1. It is mostly standard with variables ($x \in \mathrm{Var}$), lambda abstraction ($\lambda\, x.\, e$) and application ($e_1\ e_2$), but extended with string literals ($s \in \mathrm{Str}$) as

Syntax:

$$\text{Expr} \ni d, e ::= s \in \text{Str} \mid x \in \text{Var} \mid \lambda\, x.\, e \mid e\ e$$

Operational semantics: $\boxed{e \to e'}$

$$\textsc{app}\ \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \beta\ (\lambda\, x.\, e_1)\ e_2 \to e_1[x := e_2]$$

Final expressions: $\boxed{\text{final}\ e}$

$$\text{final}\ s \qquad \text{final}\ (\lambda\, x.\, e)$$

Parallel substitution: $\boxed{e[\overline{d}] = e'}$

$$s[\overline{d}] := s$$

$$x[\overline{d}] := \begin{cases} e & \text{if } x := e \in \overline{d} \\ x & \text{otherwise} \end{cases}$$

$$(\lambda\, x.\, e)[\overline{d}] := \lambda\, x.\, e[\overline{d} \setminus \{x\}]$$

$$(e_1\ e_2)[\overline{d}] := e_1[\overline{d}]\ e_2[\overline{d}]$$

Interpreter: $\boxed{[\![e]\!]^E_\delta = r}$

$$[\![e]\!]^E_0 := \text{Timeout}$$

$$[\![s]\!]^E_\delta := \text{ret}\ s$$

$$[\![x]\!]^E_\delta := (\text{thu}_{E'}\ e) \leftarrow E\ x;\ [\![e]\!]^{E'}_{\delta-1}$$

$$[\![\lambda\, x.\, e]\!]^E_\delta := \text{ret}\ (\text{clo}_E\ x.\, e)$$

$$[\![e_1\ e_2]\!]^E_\delta := (\text{clo}_{E'}\ x.\, e') \leftarrow [\![e_1]\!]^E_{\delta-1};\ [\![e']\!]^{E'\langle x := \text{thu}_E\ e_2\rangle}_{\delta-1}$$

Data structures:

$$\text{Env} \ni E := \text{Var} \xrightarrow{\text{fin}} \text{Thunk}$$

$$\text{Thunk} \ni t ::= \text{thu}_E\ e$$

$$\text{Val} \ni v ::= s \mid \text{clo}_E\ x.\, e$$

Fig. 1. The operational semantics and interpreter for LambdaLang.

$$\text{Option}\ A \ni x^? ::= \text{None} \mid \text{Some}\ (x : A) \qquad \text{ret}\ (x : A) : \text{Res}\ A := \text{Done}\ (\text{Some}\ x)$$

$$\text{Res}\ A \ni r ::= \text{Timeout} \mid \text{Done}\ (x^? : \text{Option}\ A) \qquad \text{fail} : \text{Res}\ A := \text{Done None}$$

$$(r : \text{Res}\ A) \gg\!= (f : A \to \text{Res}\ B) : \text{Res}\ B := \begin{cases} f\ x & \text{if } r = \text{Done}\ (\text{Some}\ x) \\ \text{Done None} & \text{if } r = \text{Done None} \\ \text{Timeout} & \text{if } r = \text{Timeout} \end{cases}$$

Fig. 2. Options and the result monad.

primitive data. The semantics is given using a standard small-step operational semantics ($e \to e'$) (), which reduces the left-most outer-most β-redex. The judgment final e () describes normal forms that are valid results of a program, namely string literals and lambda abstractions.

In the definition of β-reduction we make use of parallel substitution $e[\overline{d}]$ (), where $\overline{d}$ is a finite map from variable names to expressions, rather than a single substitution. The notation $x := e \in \overline{d}$ denotes that $\overline{d}$ has a mapping from x to e, the notation $\overline{d}\langle x := e\rangle$ gives the map in which the key x is associated with e, and $\overline{d_1} \cup \overline{d_2}$ denotes the left-biased union of the maps $\overline{d_1}$ and $\overline{d_2}$.

Parallel substitutions make it easy to state auxiliary lemmas about our interpreters (Lemmas 2.2 and 2.4). Its definition needs care because variables are strings (*i.e.,* $\text{Var} := \text{Str}$). First, in the lambda case $(\lambda\, x.\, e)[\overline{d}] = \lambda\, x.\, e[\overline{d} \setminus \{x\}]$, we remove x from $\overline{d}$ to handle shadowing (we do not assume Barendregt [6]'s variable convention). Second, our parallel substitution is not capture avoiding, which is only correct if we restrict reduction to closed expressions, *i.e.,* we only consider "top-level" programs without free variables [47, §11.3.2]. With that restriction, the arguments of a β-redex are always closed, and thus parallel substitution $e[\overline{d}]$ is only applied if all expressions in $\overline{d}$ are closed. A counterexample is $(\lambda\, y.\, \lambda\, x.\, y)\ x$ "a" $\to (\lambda\, x.\, x)$ "a" $\to$ "a". Formally, all theorems in § 2.3 have the precondition closed e, where $\text{closed}_X\ e$ means $\text{FV}(e) \subseteq X$, and closed e is short for $\text{closed}_\emptyset\ e$.

2.2 Implementation of the Interpreter

Figure 1 gives an environment-based interpreter for LambdaLang (🚁). Our interpreter makes use of *partiality fuel* [2] to handle non-termination, and environments and thunks in the style of the Krivine [34] machine to model call-by-name evaluation.

The interpreter has type Expr → Env → ℕ → Res Val, where the ℕ argument is the *fuel value*. The fuel value provides a bound on the number of computation steps, allowing one to define the interpreter as a structurally recursive function in which each recursive call decreases the fuel.[3] The monad Res (Figure 2) models that the interpreter can either run out of fuel (Timeout), fault (fail), or return with the result of the program (ret x). We implicitly lift Option A into Res A using Done. The notation $p \leftarrow m_1; m_2$ (much like Haskell's 'do notation') should be read as $m_1 \gg\!= (\lambda p.\, m_2)$. When p is a pattern and the provided value does not match, fail is implicitly returned.

The key data structures of the interpreter are environments (Env), thunks (Thunk) and values (Val). An environment is a finite map from variable names (*i.e.,* strings) to thunks. A thunk $\mathrm{thu}_E\ e$ is a suspended computation e in an environment E, and is key to model call-by-name evaluation. Thunks are evident in the case for application $e_1\ e_2$, where e_2 is not evaluated directly, but $\mathrm{thu}_E\ e_2$ is inserted in the environment. Consequently, in the case of a variable x, the thunk $\mathrm{thu}_{E'}\ e$ for x is retrieved from the environment E, and the suspended computation e is evaluated in its corresponding environment E'. Values represent the results of the interpreter, they are either string literals s or closures $\mathrm{clo}_E\ x.\, e$. Similar to thunks, closures contain an environment E.

Already for the call-by-name lamda calculus, we see that a substitution-based semantics is simpler than an environment-based interpreter. The latter needs additional data structures—environments, thunks, and values—which are not needed in the former. In the definition for application ($[\![e_1\ e_2]\!]^E_\delta$), one has to be careful to use the right environment, making it more complicated than just β-reduction.

2.3 Soundness and Completeness

LambdaLang programs can have three kinds of behaviors: they can terminate with a value, can fault (*e.g.,* a wrong function application such as "foo" "bar"), or can diverge (*e.g.,* $(\lambda\, x.\, x\ x)\ (\lambda\, x.\, x\ x)$). In the operational semantics these correspond to a finite reduction to a final expression, a finite reduction to a non-final stuck expression, and an infinite reduction, respectively. In the interpreter these correspond to returning ret s for some fuel value, returning fail for some fuel value, and returning Timeout for any fuel value, respectively. The following theorems state that the interpreter is sound and complete w.r.t. the operational semantics for these behaviors.

Theorem 2.1. *The interpreter is sound and complete w.r.t. the operational semantics for:*

(1) *terminating programs, i.e.,* $(\exists \delta.\, [\![e]\!]^\emptyset_\delta = \mathrm{ret}\ s)$ *iff* $e \to^* s$ (🚁), *and*
(2) *faulty programs, i.e.,* $(\exists \delta.\, [\![e]\!]^\emptyset_\delta = \mathrm{fail})$ *iff* $(\exists e'.\, e \to^* e' \not\to \wedge \neg \mathrm{final}\ e')$ (🚁), *and*
(3) *diverging programs, i.e.,* $(\forall \delta.\, [\![e]\!]^\emptyset_\delta = \mathrm{Timeout})$ *iff* $(\forall e'.\, e \to^* e' \implies \mathrm{red}\ e')$ (🚁).

We let red e denote $\exists e'.\ e \to e'$. Item 1 is specialized to string results $s \in \mathrm{Str}$, but can be generalized to any final value (🚁), namely $(\exists w, \delta.\, [\![e]\!]^\emptyset_\delta = w \wedge |v| = |w|)$ iff $e \to^* |v|$ (we introduce $|\_|$ in the following; this statement does not hold without the $\exists$ for w since $|\_|$ is not injective). This result implies confluence up to normal forms, but not determinism in general, which is proved separately in Rocq (🚁). These remarks also apply to the languages in the other sections.

The left-to-right directions (soundness) of Items 1 and 2 rely on a helping lemma that generalizes over the environment E and combines the ret/fail cases by quantifying over an optional value $v^?$:

[3] To enable simple proofs by induction on the fuel value in Rocq, we follow the convention from Amin and Rompf [2] to decrease the fuel in every recursive call, even if the term is structurally smaller. Particularly, we decrease the fuel in the recursive call $[\![e_1]\!]$ in the application case whereas that is not needed for the recursion to be well-formed.

Lemma 2.2 (🏴). *If* $[\![e]\!]^E_\delta = \text{Done } v^?$, *then there exists some* e' *such that* $e(\!|E|\!) \to^* e'$ *and if* $v^?$ *is* Some v, *then* $|v| = e'$; *or if* $v^?$ *is* None, *then* $e' \not\to$ *and* $\neg\text{final } e'$.

This lemma is proved by induction over the fuel value δ. The lemma statement relies on lifting the parallel substitution to environments and the conversion from values to expressions:

$$|\text{thu}_E\, e| := e(\!|E|\!) \qquad |s| := s$$
$$e(\!|E|\!) := e[\{x := |t| \mid x := t \in E\}] \qquad |\text{clo}_E\, x.\, e| := \lambda\, x.\, e(\!|E \setminus \{x\}|\!)$$

Here, $|t|$ converts a thunk t into an expression (🏴), $e(\!|E|\!)$ performs a parallel substitution of an environment E in an expression e by converting all thunks to expressions (🏴), and finally $|v|$ converts a value v into an expression (🏴). The first two definitions are mutually recursive.

The right-to-left directions (completeness) of Items 1 and 2 of Theorem 2.1 are proved by induction over the multi-step reduction ($\to^*$). The base cases are trivial, for the inductive cases we show that the interpreter is preserved under reductions:

Lemma 2.3 (🏴). *If* $e_1 \to e_2$ *and* $[\![e_2]\!]^\emptyset_{\delta_2} = \text{Done } v_2^?$, *then there exist an optional value* $v_1^?$ *and a fuel value* δ_1 *such that* $[\![e_1]\!]^\emptyset_{\delta_1} = \text{Done } v_1^?$ *and* $|v_1^?| = |v_2^?|$.

We again quantify over optional values $v_1^?$ and $v_2^?$ to unify the ret and fail cases. The key complexity of this lemma is that the interpreter does not necessarily give the same value for e_1 and e_2. Consider $e_1 := (\lambda\, x.\, \lambda\, y.\, x)$ "a" and $e_2 := \lambda\, y.$ "a", for which the interpreter returns $\text{clo}_{x:=\text{thu}_\emptyset \text{ "a"}}\, y.\, x$ and $\text{clo}_\emptyset\, y.$ "a", respectively. We thus compare $v_1^?$ and $v_2^?$ by converting them to expressions. Throughout the proof of Lemma 2.3 we need to show that for inputs related up to conversion, the interpreter gives outputs related up to conversion:

Lemma 2.4 (🏴). *If* $e_1(\!|E_1|\!) = e_2(\!|E_2|\!)$ *and* $[\![e_1]\!]^{E_1}_{\delta_1} = \text{Done } v_1^?$, *then there exist an optional value* $v_2^?$ *and a fuel value* δ_2 *such that* $[\![e_2]\!]^{E_2}_{\delta_2} = \text{Done } v_2^?$ *and* $|v_1^?| = |v_2^?|$.

An important detail that we omitted is that all results only hold for closed expressions. Formally, Theorem 2.1 has precondition closed e (🏴). The presence of this precondition causes two problems. First, for dynamic languages such as Nix we cannot statically determine if a program is closed because of dynamically computed binder names (see § 1). Second, the closedness conditions induce an abundance of boilerplate in the mechanized proofs. We need to lift closedness to environments (🏴) and values (🏴), prove that substitution, the reduction relation and the interpreter preserve closedness, and so on. The preconditions for some lemmas therefore become very complicated, *e.g.*, Lemma 2.4 requires closed E_1 and closed E_2 and $\text{closed}_{\text{dom } E_1}\, e_1$ and $\text{closed}_{\text{dom } E_2}\, e_2$. Our solution based on a form of deferred substitutions that we will present in § 3 remedies both of these problems.

3 Deferred Substitutions

We present the basic ideas of deferred substitutions by defining a substitution-based semantics for a core language with dynamic features, called DynLang (§ 3.1). We implement a corresponding environment-based interpreter (§ 3.2). We prove soundness and completeness of our interpreter w.r.t. our semantics, and prove soundness and completeness of our semantics w.r.t. the ordinary semantics from § 2 when restricted to closed LambdaLang expressions (§ 3.3). Finally, we demonstrate the versatility of our form of deferred substitutions by describing some variations, in particular, a substitution-based semantics of a core language with `eval` (§ 3.4).

3.1 Syntax and Operational Semantics

The syntax and operational semantics (🏴) of DynLang is given in Figure 3. It extends LambdaLang with dynamically computed variable names (a core version of `$`) and dynamic binder introduction (a

Syntax:

$$\text{Expr} \ni d, e ::= s \in \text{Str} \mid \${e}_{\overline{d}} \mid \lambda\, e.\, e \mid e\ e$$

Operational semantics: $\boxed{e \longrightarrow e'}$

$$\text{ID-STR}\ \frac{s \coloneqq e \in \overline{d}}{\${s}_{\overline{d}} \longrightarrow e} \qquad \text{ID}\ \frac{e \longrightarrow e'}{\${e}_{\overline{d}} \longrightarrow \${e'}_{\overline{d}}}$$

$$\text{ABS}\ \frac{e_1 \longrightarrow e_1'}{\lambda\, e_1.\, e_2 \longrightarrow \lambda\, e_1'.\, e_2} \qquad \text{APP}\ \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}$$

$$\beta\ \ (\lambda\, s.\, e_1)\ e_2 \longrightarrow e_1[s \coloneqq e_2]$$

Parallel substitution: $\boxed{e[\overline{d}] = e'}$

$$\begin{aligned} s[\overline{d}] &\coloneqq s \\ (\${e}_{\overline{d'}})[\overline{d}] &\coloneqq \${e[\overline{d}]}_{\overline{d} \cup \overline{d'}} \\ (\lambda\, e_1.\, e_2)[\overline{d}] &\coloneqq \lambda\, e_1[\overline{d}].\, e_2[\overline{d}] \\ (e_1\ e_2)[\overline{d}] &\coloneqq e_1[\overline{d}]\ e_2[\overline{d}] \end{aligned}$$

Final expressions: $\boxed{\text{final}\ e}$

$$\text{final}\ s \qquad \text{final}\ (\lambda\, s.\, e)$$

Interpreter: $\boxed{[\![e]\!]^E_\delta = r}$

$$\begin{aligned} [\![e]\!]^E_0 &\coloneqq \text{Timeout} \\ [\![s]\!]^E_\delta &\coloneqq \text{ret}\ s \\ [\![\${e}]\!]^E_\delta &\coloneqq s \leftarrow [\![e]\!]^E_{\delta-1}; \\ &\qquad (\text{thu}_{E'}\ e') \leftarrow E\ s; \\ &\qquad [\![e']\!]^{E'}_{\delta-1} \\ [\![\lambda\, e_1.\, e_2]\!]^E_\delta &\coloneqq s \leftarrow [\![e_1]\!]^E_{\delta-1}; \\ &\qquad \text{ret}\ (\text{clo}_E\ s.\, e_2) \\ [\![e_1\ e_2]\!]^E_\delta &\coloneqq (\text{clo}_{E'}\ s.\, e') \leftarrow [\![e_1]\!]^E_{\delta-1}; \\ &\qquad [\![e']\!]^{E'\langle s \coloneqq \text{thu}_E\ e_2\rangle}_{\delta-1} \end{aligned}$$

Data structures:

$$\begin{aligned} \text{Env} \ni E &\coloneqq \text{Str} \xrightarrow{\text{fin}} \text{Thunk} \\ \text{Thunk} \ni t &::= \text{thu}_E\ e \\ \text{Val} \ni v &::= s \mid \text{clo}_E\ s.\, e \end{aligned}$$

Fig. 3. The operational semantics and interpreter for DynLang.

core version of `with`, in the sense that `with r; e` dynamically introduces a binding for every attribute in the attribute set resulting from `r`) by generalizing the constructs for variables ${e} and lambda abstractions $\lambda\, e_1.\, e_2$. The operand e of ${e} is an arbitrary expression that computes a string, and the result of ${e} is the value of the variable corresponding to the computed string. Similarly, the first operand e_1 of the generalized lambda abstraction $\lambda\, e_1.\, e_2$ is an arbitrary expression that computes a string, which is used as the name of the binder. Ordinary variables x are written as ${"x"}, and ordinary lambda abstractions $\lambda\, \text{x}.\, e$ as $\lambda\, \text{"x"}.\, e$. The identity function becomes $\lambda\, \text{"x"}.\, \${"x"}$. A more complicated example is $((\lambda\, \text{"x"}.\, \lambda\, \text{"y"}.\, \${\${"y"}})\ \text{"a"}\ \text{"x"})$. After reducing the β-redexes, ${"y"} reduces to "x", and ${${"y"}} thus reduces to the value of "x", *i.e.,* the string "a". Also consider $((\lambda\, \text{"x"}.\, \lambda\, \${"x"}.\, \${"y"})\ \text{"y"}\ \text{"a"})$, where $\lambda\, \${"x"}$ reduces to $\lambda\, \text{"y"}$, and the result of the whole program is therefore the string "a".

Our technique of deferred substitutions is inspired by the calculus of explicit substitutions [1] and related to a slightly different proposal by Lippmeier [37] (see § 7.1 for details), which reifies substitutions as an explicit constructor in the expression syntax. Instead of allowing explicit substitutions to appear anywhere in the expression syntax, we only let them occur at constructs that compute variables or refer to dynamically computed names. For DynLang this means we annotate ${e}$\overline{d}$ with a deferred substitution $\overline{d}$, which is a finite map from strings to expressions. We write ${e} instead of ${e}$\emptyset$ in case the substitution is empty.

All deferred substitutions in *source programs* (*i.e.,* programs written by a user) are empty, they only become non-empty as the result of reduction steps. Non-empty deferred substitutions are created by β-reduction, which reduces $(\lambda\, s.\, e_1)\ e_2$ to $e_1[s \coloneqq e_2]$. Here, $e[\overline{d}]$ performs parallel substitution (🐟), with two differences compared to the standard definition from § 2.1. First, instead

of replacing values ($x[\overline{d}] := e$ if $x := e \in \overline{d}$, as done in § 2.1), we add $\overline{d}$ to the deferred substitution in every variable $\${e'}_{\overline{d'}}$ in e. Second, we do not prevent shadowing in the lambda case $(\lambda\, e_1.\, e_2)[\overline{d}] := \lambda\, e_1[\overline{d}].\, e_2[\overline{d}]$ by removing the binding for e_1 from $\overline{d}$. We cannot do so because e_1 could be an arbitrary computation whose resulting string literal is not yet known, so we do not know which binding to remove from $\overline{d}$. Shadowing is instead handled by taking the left-biased union in the variable case, *i.e.,* $(\${e}_{\overline{d'}})[\overline{d}] := \${e[\overline{d}]}_{\overline{d} \cup \overline{d'}}$. In other words, parallel substitutions overwrite prior deferred substitutions. Consider the LambdaLang term $((\lambda\, x.\, \lambda\, x.\, x)$ "a" "b"$)$. Converted to DynLang, it reduces as $(\lambda$ "x"$.\, \lambda$ "x"$.\, \${$"x"$})$ "a" "b" $\to (\lambda$ "x"$.\, \${$"x"$}_{\text{x:="a"}})$ "b" $\to \${$"x"$}_{\text{x:="b"}} \to$ "b".

Deferred substitutions are used when the expression e in $\${e}_{\overline{d}}$ is reduced to a string literal s. Using ID-STR $\${s}_{\overline{d}}$ is reduced to e for $s := e \in \overline{d}$. Finally, the compatibility rules ID, ABS and APP allow reduction to happen in a call-by-name order.

We stress that there is no need for closedness conditions because deferred substitutions are well-behaved for programs with free variables, whereas ordinary non-capture avoiding substitutions are not. Consider the counterexample $((\lambda\, y.\, \lambda\, x.\, y)\ x$ "a"$)$ from § 2.1. When converted into DynLang it reduces as $(\lambda$ "y"$.\, \lambda$ "x"$.\, \${$"y"$})\ \${$"x"$}$ "a" $\to (\lambda$ "x"$.\, \${$"y"$}_{\text{y:=}\${\text{"x"}}})$ "a" $\to \${$"y"$}_{\text{x:="a",y:=}\${\text{"x"}}} \to \${$"x"$}$. Our semantics thus correctly gets stuck because the variable x is not bound, instead of reducing to a nonsensical result which the semantics from § 2.1 does.

3.2 Implementation of the Interpreter

Figure 3 gives an environment-based interpreter for DynLang (🏴‍☠️). The interpreter follows the same structure as the one for LambdaLang (§ 2.2) with the expected modifications for the new constructs. To evaluate $\${e}$, we evaluate e to a string literal, and look up the thunk in the environment E. Similarly, for $\lambda\, e_1.\, e_2$, we evaluate e_1 to a string literal, and return a closure. Like the semantics, the interpreter is well-behaved for programs with free variables. Reconsider the example $((\lambda\, y.\, \lambda\, x.\, y)\ x$ "a"$)$ from § 2.1 and 3.1. As expected, the interpreter fails (with sufficient fuel):

$$[\![(\lambda \text{ "y"}.\, \lambda \text{ "x"}.\, \${\text{"y"}})\ \${\text{"x"}}\ \text{"a"}]\!]^{\emptyset} = [\![(\lambda \text{ "x"}.\, \${\text{"y"}})\ \text{"a"}]\!]^{\text{y:=thu}_{\emptyset}\ \${\text{"x"}}} =$$
$$[\![\${\text{"y"}}]\!]^{\text{x:="a",y:=thu}_{\emptyset}\ \${\text{"x"}}} = [\![\${\text{"x"}}]\!]^{\emptyset} = \text{fail}$$

The interpreter is only defined on *source* programs, *i.e.,* programs without deferred substitutions, as there is no case $\${e}_{\overline{d}}$ for non-empty $\overline{d}$. This is well-formed because when given an expression with empty deferred substitutions as input, there are only recursive calls on expressions with empty deferred substitutions, and only expressions with empty deferred substitutions are inserted into the environment. In § 3.3, we generalize the interpreter to all expressions to carry out our proofs.

3.3 Soundness and Completeness

We establish soundness and completeness of the interpreter w.r.t. the operational semantics for terminating, faulty, and diverging programs. The main theorem is analogous to Theorem 2.1. It is worth noting that unlike the results in § 2.3, there is no need for closedness preconditions because deferred substitutions are well-behaved for programs with free variables.

The main theorem (🏴‍☠️) is proven using the same helping lemmas and definitions that we developed in § 2.3. There are two important observations. First, since the closedness conditions are gone, there is much less boilerplate when carrying out a mechanized proof in Rocq. Second, to state a variant of Lemma 2.3, which says that the interpreter is preserved under reductions (🏴‍☠️), we need to lift the interpreter from source programs to expressions with non-empty deferred substitutions (β-reduction creates deferred substitutions). This is done by extending the variable case:

$$[\![\${e}_{\overline{d}}]\!]^{E}_{\delta} := s \leftarrow [\![e]\!]^{E}_{\delta-1};\ (\text{thu}_{E'}\ e') \leftarrow (E \cup\!\!\!\cdot\ \{x := \text{thu}_{\emptyset}\ d \mid x := d \in \overline{d}\})\ s;\ [\![e']\!]^{E'}_{\delta-1}$$

We use the left-biased union to first lookup the variable s in the environment E, and if that fails, look it up in the deferred substitution $\overline{d}$. With the lifted version of the interpreter at hand, all remaining proofs are analogous to those in § 2.3 (but without closedness boilerplate).

To get additional confidence in our method, we prove that our semantics is sound and complete w.r.t. the ordinary substitution-based semantics from § 2 when restricted to closed LambdaLang expressions. To state this result, recall that every LambdaLang expression can be converted into DynLang by transforming variables x into ${"x"} and lambda abstractions $\lambda\,\mathrm{x}.\,e$ into $\lambda\,\text{"x"}.\,e$ (🏴).

Theorem 3.1. *The DynLang semantics is sound and complete w.r.t. the LambdaLang semantics for:*

(1) *terminating programs, i.e., for all closed* $e \in \mathsf{Expr}_{\mathsf{lam}}$ *we have* $e \to^*_{\mathsf{dyn}} s$ *iff* $e \to^*_{\mathsf{lam}} s$ (🏴), *and*
(2) *faulty programs, i.e., for all closed* $e \in \mathsf{Expr}_{\mathsf{lam}}$ *we have* $(\exists e'.\, e \to^*_{\mathsf{dyn}} e' \not\to_{\mathsf{dyn}} \wedge\, \neg\mathsf{final}_{\mathsf{dyn}}\, e')$ *iff* $(\exists e'.\, e \to^*_{\mathsf{lam}} e' \not\to_{\mathsf{lam}} \wedge\, \neg\mathsf{final}_{\mathsf{lam}}\, e')$ (🏴), *and*
(3) *diverging programs, i.e., for all closed* $e \in \mathsf{Expr}_{\mathsf{lam}}$ *we have* $(\forall e'.\, e \to^*_{\mathsf{dyn}} e' \implies \mathsf{red}_{\mathsf{dyn}}\, e')$ *iff* $(\forall e'.\, e \to^*_{\mathsf{lam}} e' \implies \mathsf{red}_{\mathsf{lam}}\, e')$ (🏴).

This theorem is derived from the soundness and completeness theorems of the interpreters for LambdaLang and DynLang, and the observation that these interpreters are nearly identical for any LambdaLang expression (only the fuel value might differ). Unlike Item 1 of Theorem 2.1 and Item 1 of Theorem 4.1, which generalize to all values, Item 1 of Theorem 3.1 does not generalize to all values, since DynLang and LambdaLang have different notions of values and expressions.

3.4 Variations of Deferred Substitutions

Deferred substitutions can easily be adjusted to different language features. We show that they can be simplified if one leaves out dynamic computation of variable names (*i.e.,* no `$`) and that they transfer to a semantics for a functional version of `eval`.

Without $. First consider a variation of DynLang that only supports dynamic introduction of binders (akin to `with` in Nix, see § 4.1), but not computing variable names:

$$\mathsf{Expr} \ni d, e ::= s \in \mathsf{Str} \mid x_{e^?} \mid \lambda\, e.\, e \mid e_1\ e_2$$

Since variables are static, the deferred substitution is no longer a finite map, but an option, *i.e.,* we let $e^? \in \mathsf{Option\ Expr}$. The rule ID-STR is adjusted to $x_{\mathrm{Some}\ e} \to e$, while x_{None} is stuck because it means the variable is unbound. As per convention, we assume that source programs contain only empty deferred substitutions (variables are of the form x_{None}). Parallel substitution is defined as:

$$x_{e^?}[\overline{d}] := \begin{cases} x_{\mathrm{Some}\ d} & \text{if } x := d \in \overline{d} \\ x_{e^?} & \text{otherwise} \end{cases}$$

Eval. Now consider EvalLang, a variation of the prior language with an `eval` construct (🏴):

$$\mathsf{Expr} \ni d, e ::= s \in \mathsf{Str} \mid x_{e^?} \mid \lambda\, e.\, e \mid e_1\ e_2 \mid \mathbf{eval}^{-}_{\overline{d}}\, e$$

The intuitive semantics of **eval** e is that it evaluates e to a string, then parses it as an expression, and executes that. For example, the result of **eval** `"(x: y: eval! y) \"a\" \"x\""` is the string "a". (Like Nix, `x: e` is syntax for lambda abstractions and `"` is escaped as `\"`, we use the exclamation mark to disambiguate `eval!` from the variable `eval`.) With deferred substitutions it is straightforward to give an operational semantics. The reduction rules (🏴) and case in the interpreter (🏴) for **eval** are:

EVAL-STR
$$\frac{\mathrm{parse}\ s = \mathrm{Some}\ e}{\mathbf{eval}^{-}_{\overline{d}}\ s \to e[\overline{d}]}$$

EVAL
$$\frac{e \to e'}{\mathbf{eval}^{-}_{\overline{d}}\ e \to \mathbf{eval}^{-}_{\overline{d}}\ e'}$$

$$[\![\mathbf{eval}\ e]\!]^E_\delta := s \leftarrow [\![e]\!]^E_{\delta-1};\ e' \leftarrow \mathrm{parse}\ s;\ [\![e']\!]^E_{\delta-1}$$

The deferred substitution $\overline{d}$ in $\mathbf{eval}_{\overline{d}}\, e$ allows us to defer substitution until the moment that the operand e has been reduced to a string and has been parsed. Proving soundness and completeness of the interpreter w.r.t. the operational semantics of EvalLang follows the same recipe as § 3.3. The main additional work (done in Rocq) is implementing the parse function (🐓).

4 A Semantics for the Nix Expression Language

We present our semantics of Nix using deferred substitutions (§ 4.2) and our interpreter (§ 4.3). We prove soundness and completeness of the interpreter w.r.t. the operational semantics (§ 4.4). We first give a brief introduction to the Nix language, highlighting its challenging features (§ 4.1).

4.1 Introduction to Nix

Attribute sets. A key feature of Nix are JSON-style attribute sets, which are constructed using the syntax `{ x1 = e1; ...; xn = en }`. The members of an attribute set `r` can be accessed using the selection operator `r.x`. For example, `let r = { x = 10; y = 12; }; in r.x` returns 10. The members of attribute sets are evaluated lazily, so `{ x = Omega; y = 2; }.y` returns 2.

What makes Nix's attribute sets special is that when prefixed with the `rec` keyword, the members can refer to each other, even (mutually) recursively. For example, `rec { y = x; x = 2; }.y` returns 2, and `rec { x = x; }.x` diverges. Recursive attribute sets become interesting when used in combination with functions. For example (`x: e` is a lambda abstraction and `!` is Boolean negation):

```
rec { f = x: if x == 0 then true else !(f (x - 1)); }.f n
```

This program returns `true` iff `n` is even. Attribute sets are allowed to have a special `"__functor"` member, which allows them to be used as a function:

```
{ "__functor" = r: x: if x == 0 then true else !(r (x - 1)); } n
```

Note that `r` is not the argument supplied to the record set (here, `n`), but the entire attribute set itself, allowing one to write recursive functions. This program thus also returns `true` iff `n` is even.

The `let`/`with` constructs. Let bindings in Nix are allowed to refer to each other, possibly mutually recursively. For example `let y = x; x = true; in y` returns `true`. Due to laziness, the program `let x = x; in true` also returns `true` as we do not use `x`. The `with r; e` construct adds all members of an attribute set `r` to the scope of the expression `e`. For example, `with { x = 10; y = 12; }; x` returns 10. The evaluation of the attribute set is lazy, so `with rec { x = x; }; true` returns `true` as we do not use `x`. As described in § 1, what makes `with` and `let` special is their subtle shadowing rules, namely a `let` binding has priority over `with` regardless of the order in which nesting occurs:

```
let x = 10; in    with { x = 12; }; x   # returns 10
with { x = 10; }; let x = 12; in    x   # returns 12
```

Variables bound by lambda abstractions have the same binding priority as let bindings.

Matching lambdas and recursion through defaults. Nix supports lambda abstractions that match on attribute sets. For example, `({ x, y }: x) { x = 10; y = 12; }` returns `10`. A matching lambda can either be *strict* (the members of attribute set and the bindings in the pattern should be the same) or *non-strict* (the attribute set is allowed to have more members than the pattern). Strict matching is the default, while non-strict matching is enabled by adding the `...` suffix to the matching pattern. For example, `({ x }: x) { x = 10; y = 12; }` faults because there is no binder for `y` in the pattern, while `({ x, ... }: x) { x = 10; y = 12; }` returns 10.

To deal with the situation where the pattern has more members than the attribute set, *defaults* can be given using the `? d` syntax. The default `d` is allowed to be an arbitrary expression that can even refer to other members in the pattern. For example, `({ x, y ? x }: y) { x = 10; }` returns `10`.

Defaults might be recursive, so `({ x ? x }: x) {}` diverges, while `({ x ? x }: true) {}` returns `true` because of laziness. The program `({ x ? y, y ? x }: x) r` diverges iff the attribute set `r` contains neither `x` nor `y`. Defaults can be functions, enabling (mutual) recursion:

```
({ f ? (x: if x == 0 then true else !(f (x - 1)))}: f) {} n
```

Similar to the prior recursive examples, this program returns `true` iff `n` is even.

Sequencing. Because Nix is lazy, a program like `let x = e1; in e2` only executes `e1` when `x` is used in `e2`. For example, `let x = Omega; in true` returns `true`. It is sometimes useful to 'force' a computation using the builtins `seq e1 e2` and `deepSeq e1 e2`, which will evaluate `e1` before `e2`. The difference between the two is that `deepSeq` evaluates attribute sets and lists in `e1` recursively, while `seq` only evaluates the outermost one. Both `seq Omega true` and `deepSeq Omega true` diverge. The program `seq rec { x = x; } true` returns `true` as the recursive attribute set is unfolded once, while `deepSeq rec { x = x; } true` diverges because the recursive unfolding of `rec { x = x; }` is infinite.

As described in § 1, what makes `deepSeq` interesting is the order of evaluation:

```
deepSeq { x = Omega; y = 0 0; } 10
```

This program either faults (when the faulty application `0 0` is evaluated first) or diverges (when the loop `Omega` is evaluated first). Both behaviors (*i.e.,* both evaluation orders) can be observed in the actual implementation of Nix. It appears that numbers are assigned to attribute members, and attribute sets are processed deterministically based on that numbering.

Operators. Nix provides many operators, *e.g.,* for arithmetic, comparison, merging attribute sets, and inspecting the type of an expression. An interesting aspect of these operators is their behavior w.r.t. faults and divergence. Binary operations are lazy in the second operand. For example, we know that attribute member selection is not defined on Booleans, so `true . Omega` fails, but `Omega . true` diverges because Nix tries to inspect the type of the first operand first. Similarly, `true == Omega` diverges because `==` is overloaded for any data type.

As described in § 1, the semantics of `==` is not obvious. When comparing two attribute sets, only when the domain of the attribute sets is the same, the attribute values are compared pairwise. So the following program terminates, returning `false`:

```
{ x = Omega; } == { x = Omega; y = 10; }
```

If the domains are the same, the behavior depends on the evaluation order. For example, using the actual implementation of Nix, the program `{ x = Omega; y = 12; } == { x = Omega; y = 10; }` can be observed to both return `false` and diverge.

Integers and floats. Nix supports 64-bit integers and floating-point numbers. Integer overflow results in a fault (unlike C, a fault causes an exception, not undefined behavior where any behavior is allowed). Floating-point numbers are IEEE 754, binary64, with platform-dependent quirks.

4.2 Syntax and Operational Semantics

Since Nix source programs contain redundancy in terms of language constructs, we formalize our semantics for a core language, called NixLang. In § 5 we present an elaborator that translates Nix source code into NixLang. Figure 4 gives the syntax (🐟) and operational semantics (🐟), which we explain below.

Attribute sets. Attribute sets in NixLang are of the form $\{x_1 \coloneqq \alpha_1, \ldots, x_n \coloneqq \alpha_n\}$, where each member α is either **nonrec** e or **rec** e (🐟). We keep track of the recursivity of each member instead of the attribute set as a whole, to easily support Nix's `inherit` keyword in our elaborator. This means that `{ x1 = e1; ...; xn = en }` is elaborated into $\{x_1 \coloneqq \textbf{nonrec}\ e_1, \ldots, x_n \coloneqq \textbf{nonrec}\ e_n\}$, and

Syntax:

$$\begin{array}{rl|rl}
\text{BaseLit} \ni b & ::= s \in \text{Str} \mid n \in \mathbb{Z} \mid q \in \mathbb{F}_{\text{IEEE}} \mid \textbf{null} \mid \textbf{true} \mid \textbf{false} & \text{Attr} \ni \alpha & ::= \tau\ e \\
\text{Matcher} \ni m & ::= \{\overline{e^?}\} \mid \{\overline{e^?},\ \ldots\} & \text{Subst} \ni \sigma & ::= k\ e \\
\text{BinOp} \ni \circledcirc & ::= == \mid < \mid \textbf{.} \mid + \mid \ldots & \text{Rec} \ni \tau & ::= \textbf{rec} \mid \textbf{nonrec} \\
\text{Expr} \ni d, e & ::= b \mid [\vec{e}] \mid \{\overline{\alpha}\} \mid x_{\sigma^?} \mid \lambda\, x.\, e \mid \lambda\, m.\, e \mid e\ e & \text{Kind} \ni k & ::= \textbf{abs} \mid \textbf{with} \\
& \quad \mid \textbf{let}/k\ e\ \textbf{in}\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid e \circledcirc e \mid \textbf{seq}/\mu\ e\ e & \text{Mode} \ni \mu & ::= \textbf{shallow} \mid \textbf{deep}
\end{array}$$

Operational semantics: $\boxed{e \to_\mu e'}$

$$\frac{\exists x, d.\, x \coloneqq \textbf{rec}\ d \in \overline{\alpha}}{\{\overline{\alpha}\} \to_\mu \{\text{unfold}_1\ \overline{\alpha}\}}\ \textsc{attr-rec} \qquad x_{\text{Some}\ (k\ e)} \to_\mu e\ \textsc{id-str} \qquad (\lambda\, x.\, e_1)\ e_2 \to_\mu e_1[x \coloneqq \textbf{abs}\ e_2]\ \beta$$

$$\frac{m \sim \overline{d} \rightsquigarrow \overline{\alpha}}{(\lambda\, m.\, e_1)\ \{\overline{\textbf{nonrec}\ d}\} \to_\mu e_1[\text{indirects}\ \overline{\alpha}]}\ \beta\textsc{-match} \qquad \frac{\text{"\_\_functor"} \coloneqq e_1 \in \overline{d}}{\{\overline{\textbf{nonrec}\ d}\}\ e_2 \to_\mu (e_1\ \{\overline{\textbf{nonrec}\ d}\})\ e_2}\ \textsc{functor}$$

$$\textbf{let}/k\ \{\overline{\textbf{nonrec}\ d}\}\ \textbf{in}\ e \to_\mu e[\{x \coloneqq k\ d \mid x \coloneqq d \in \overline{d}\}]\ \textsc{let-attr-attr}$$

$$\textbf{if true then}\ e_1\ \textbf{else}\ e_2 \to_\mu e_1\ \textsc{if-true} \qquad \textbf{if false then}\ e_1\ \textbf{else}\ e_2 \to_\mu e_2\ \textsc{if-false}$$

$$\frac{\text{final}_{\textbf{shallow}}\, e_1 \quad \text{final}_{\textbf{shallow}}\, e_2 \quad e_1 \Downarrow^{\circledcirc} \Phi \quad \Phi\ e_2\ e}{e_1 \circledcirc e_2 \to_\mu e}\ \textsc{bin-op} \qquad \frac{\text{final}_{\mu'}\ e_1}{\textbf{seq}/\mu'\ e_1\ e_2 \to_\mu e_2}\ \textsc{seq-final} \qquad \frac{e \to_{\mu'} e'}{K_\mu^{\mu'}[e] \to_\mu K_\mu^{\mu'}[e']}\ \textsc{ctx}$$

Evaluation contexts:

$$\begin{array}{rll}
K_{\textbf{deep}}^{\textbf{deep}} & ::= [\vec{e_1}, \square, \vec{e_2}] & \text{if } \forall e_1 \in \vec{e_1}.\, \text{final}_{\textbf{deep}}\ e_1 \\
& \mid \{\overline{\textbf{nonrec}\ d}, x \coloneqq \textbf{nonrec}\ \square\} & \text{if } \forall y \coloneqq d \in \overline{d}.\, \text{final}_{\textbf{deep}}\ d \vee x \sqsubset y \\
K_\mu^{\textbf{shallow}} & ::= \square\ e_2 \mid (\lambda\, m.\, e_1)\ \square \mid \textbf{let}/k\ \square\ \textbf{in}\ e_2 \mid \textbf{if}\ \square\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \square \circledcirc e_2 & \\
& \mid e_1 \circledcirc \square & \text{if } \text{final}_{\textbf{shallow}}\ e_1 \text{ and } \exists \Phi.\, e_1 \Downarrow^{\circledcirc} \Phi \\
K_\mu^{\mu'} & ::= \textbf{seq}/\mu'\ \square\ e_2 &
\end{array}$$

Final expressions: $\boxed{\text{final}_\mu\ e}$

$$\frac{b \in \mathbb{Z} \implies -2^{63} \le b < 2^{63}}{\text{final}_\mu\ b} \qquad \text{final}_{\textbf{shallow}}\ [\vec{e}] \qquad \frac{\forall e \in \vec{e}.\, \text{final}_{\textbf{deep}}\ e}{\text{final}_{\textbf{deep}}\ [\vec{e}]} \qquad \text{final}_{\textbf{shallow}}\ \{\overline{\textbf{nonrec}\ e}\}$$

$$\frac{\forall x \coloneqq e \in \overline{e}.\, \text{final}_{\textbf{deep}}\ e}{\text{final}_{\textbf{deep}}\ \{\overline{\textbf{nonrec}\ e}\}} \qquad \text{final}_\mu\ (\lambda\, x.\, e) \qquad \text{final}_\mu\ (\lambda\, m.\, e)$$

Auxiliaries:

$$\text{unfold}_1\ \overline{\alpha} \coloneqq \{x \coloneqq \textbf{nonrec}\ e \mid x \coloneqq \textbf{nonrec}\ e \in \overline{\alpha}\} \cup \{x \coloneqq \textbf{nonrec}\ e[\text{indirects}\ \overline{\alpha}] \mid x \coloneqq \textbf{rec}\ e \in \overline{\alpha}\}$$

$$\text{indirects}\ \overline{\alpha} \coloneqq \{x \coloneqq \textbf{abs}\ \{\overline{\alpha}\}\textbf{.}x \mid x \in \text{dom}\ \overline{\alpha}\}$$

Fig. 4. Syntax and operational semantics of NixLang.

Binary operator semantics: $\boxed{e_1 \Downarrow^{\circledcirc} (\Phi \subseteq \text{Expr} \times \text{Expr})}$

BINOP-ADD
$$n_1 \Downarrow^{+} \{(n_2, m) \mid m = n_1 + n_2 \text{ and } -2^{63} \leq m < 2^{63}\}$$

BINOP-SELECT-ATTR
$$\{\overline{\textbf{nonrec}\ e}\} \Downarrow^{\bullet} \{(s, e) \mid s \coloneqq e \in \overline{e}\}$$

BINOP-EQ-STR
$$s_1 \Downarrow^{==} \{(s_1, \textbf{true})\} \cup \{(s_2, \textbf{false}) \mid s_1 \neq s_2\}$$

BINOP-EQ-LIST
$$[\vec{e}] \Downarrow^{==} \{([\vec{d}], \textbf{false}) \mid \text{len}\ \vec{e} \neq \text{len}\ \vec{d}\} \cup \{([\vec{d}], e_1 == d_1\ \&\&\ \cdots\ \&\&\ e_n == d_n) \mid \text{len}\ \vec{e} = \text{len}\ \vec{d} = n\}$$

BINOP-EQ-ATTR
$$\begin{aligned}\{\overline{\textbf{nonrec}\ e}\} \Downarrow^{==} \{(\{\overline{\textbf{nonrec}\ d}\}, \textbf{false}) \mid \text{dom}\ \overline{e} \neq \text{dom}\ \overline{d}\} \cup \\ \{(\{\overline{\textbf{nonrec}\ d}\}, e_{x_1} == d_{x_1}\ \&\&\ \cdots\ \&\&\ e_{x_n} == d_{x_n}) \mid \\ \text{dom}\ \overline{e} = \text{dom}\ \overline{d} \text{ and } x_1 \sqsubset \ldots \sqsubset x_n \text{ cover dom } \overline{e}\}\end{aligned}$$

Argument matching: $\boxed{m \sim \overline{d} \rightsquigarrow \overline{\alpha}}$

$$\{\emptyset, \ldots\} \sim \overline{d} \rightsquigarrow \emptyset$$

$$\frac{\{\overline{e^?}, \ldots\} \sim \overline{d} \rightsquigarrow \overline{\alpha} \qquad x \notin \text{dom}\ \overline{e^?} \qquad x \notin \text{dom}\ \overline{d}}{\{\overline{e^?}\langle x \coloneqq e^?\rangle, \ldots\} \sim \overline{d}\langle x \coloneqq d\rangle \rightsquigarrow \overline{\alpha}\langle x \coloneqq \textbf{nonrec}\ d\rangle}$$

$$\frac{\{\overline{e^?}, \ldots\} \sim \overline{d} \rightsquigarrow \overline{\alpha} \qquad \text{dom}\ \overline{d} \subseteq \text{dom}\ \overline{e^?}}{\{\overline{e^?}\} \sim \overline{d} \rightsquigarrow \overline{\alpha}} \qquad \frac{\{\overline{e^?}, \ldots\} \sim \overline{d} \rightsquigarrow \overline{\alpha} \qquad x \notin \text{dom}\ \overline{e^?} \qquad x \notin \text{dom}\ \overline{d}}{\{\overline{e^?}\langle x \coloneqq \text{Some}\ e\rangle, \ldots\} \sim \overline{d} \rightsquigarrow \overline{\alpha}\langle x \coloneqq \textbf{rec}\ e\rangle}$$

Fig. 5. Matching semantics and excerpt of the operator semantics of NixLang.

`rec { x1 = e1; ...; xn = en }` is elaborated into $\{x_1 \coloneqq \textbf{rec}\ e_1, \ldots, x_n \coloneqq \textbf{rec}\ e_n\}$. Like Dolstra [16] we let the operational semantics expand recursive attribute sets into non-recursive ones by unfolding them one level. For instance $\{x \coloneqq \textbf{rec}\ x\}$ reduces to $\{x \coloneqq \textbf{nonrec}\ (\{x \coloneqq \textbf{rec}\ x\}\,.\,x)\}$ (omitting details about deferred substitutions) using ATTR-REC (🏴). The rule FUNCTOR (🏴) makes sure that an application of an attribute set is reduced to an application of the `__functor` member.

Substitution and the `let`/`with` constructs. Since Nix does not feature a construct to refer to dynamically computed variables, NixLang employs the approach of deferred substitutions from § 3.4 where variables $x_{\sigma^?}$ are annotated with an option $\sigma^?$ instead of a finite map $\overline{\sigma}$. To account for the shadowing rules of `let` and `with`, we do not let $\sigma^?$ be just an optional expression (like § 3.4), but we let it be an optional pair $k\ d \in \text{Subst}$. The *kind* k (🏴) tracks whether the last substitution came from a `let` binding/lambda abstraction ($k = \textbf{abs}$) or a `with` binding ($k = \textbf{with}$). Similarly, parallel substitutions now take a finite map $\overline{\sigma} \in \text{Str} \xrightarrow{\text{fin}} \text{Subst}$ from strings to expression/kind pairs. An excerpt of the definition of parallel substitution (🏴) is:

$$x_{\sigma^?}[\overline{\varsigma}] \coloneqq \begin{cases} x_{\text{Some (abs } d)} & \text{if } x \coloneqq \textbf{with}\ e \in \overline{\varsigma} \text{ and } \sigma^? = \text{Some (abs } d) \\ x_{\text{Some } (k\ e)} & \text{if } x \coloneqq k\ e \in \overline{\varsigma} \\ x_{\sigma^?} & \text{otherwise} \end{cases}$$

$$(\lambda x.\, e)[\overline{\varsigma}] \coloneqq \lambda x.\, e[\overline{\varsigma}]$$

There are three cases for the variable $x_{\sigma^?}$. The first ensures that **with** does not shallow **abs**. The second ensures that the new binding is used otherwise, both if $\sigma^?$ is Some (shadowing) or None (the variable still being free). The third accounts for the variable not being in the deferred substitution $\overline{\varsigma}$. Parallel substitution for lambda abstractions and the rules β (🏴) and ID-STR (🏴) are similar to § 3.

NixLang has a generalized `let`/`with` construct **let**/k d **in** e where k is a kind (**abs** or **with**) (🏴). Thus `let x1 = d1; ...; xn = dn; in e` is elaborated into **let**/**abs** $\{x_1 \coloneqq \textbf{rec}\ d_1, \ldots, x_n \coloneqq \textbf{rec}\ d_n\}$ **in** e

and `with d; e` is elaborated into **let**/**with** d **in** e. Our generalized **let** construct allows for a uniform reduction rule LET-ATTR-ATTR (), where k is used in the parallel substitution $\{x \coloneqq k\ d \mid x \coloneqq d \in \overline{d}\}$.

Matching lambdas and recursion through defaults. NixLang has both strict $\lambda\ \{\overline{e?}\}\ .\ e$ and non-strict $\lambda\ \{\overline{e?},\ \ldots\}\ .\ e$ matching lambdas. We let $\overline{e?} \in \mathrm{Str} \xrightarrow{\mathrm{fin}} \mathrm{Option\ Expr}$, where None mappings account for bindings without a default, and Some mappings account for bindings with a default. For instance, `{ x, y ? x }: y` is elaborated into $\lambda\ \{x \coloneqq \mathrm{None}, y \coloneqq \mathrm{Some}\ x\}\ .\ y$.

The rule β-MATCH () gives an operational semantics to matching lambdas. It makes use of the *matching relation* $m \sim \overline{d} \rightsquigarrow \overline{\alpha}$ (). This relation is inspired by Dolstra and Löh [17], but extended to support recursion through defaults by transforming a matcher m and arguments $\overline{d}$ into a *recursive* attribute set $\overline{\alpha}$, which we then substitute with via an indirection. Attributes in $\overline{d}$ matched against by m appear as non-recursive attributes in $\overline{\alpha}$. When an argument with a default value is given in m but no matching attribute exists in $\overline{d}$, the default value appears as a recursive attribute in $\overline{\alpha}$. For example, we have $\{x \coloneqq \mathrm{None}, y \coloneqq \mathrm{Some}\ x\} \sim \{x \coloneqq 10\} \rightsquigarrow \{x \coloneqq \textbf{nonrec}\ 10, y \coloneqq \textbf{rec}\ x\}$.

Sequencing. NixLang has a generalized sequencing operator **seq**/$\mu\ e_1\ e_2$ that is equipped with a mode μ (), which is either **shallow** (for Nix's `seq`) or **deep** (for Nix's `deepSeq`). In our operational semantics, we parameterize the reduction relation $\rightarrow_\mu$ () and the $\mathrm{final}_\mu\ e$ () predicate with a mode μ. The idea is that some reduction steps only happen in **deep** mode, and similarly fewer expressions in **deep** mode are final. Concretely, the members $\overline{\alpha}$ of attribute sets $\{\overline{\alpha}\}$ and elements $\vec{e}$ of lists $[\vec{e}]$ are only reduced in **deep** mode, and consequently attribute sets $\{\overline{\alpha}\}$ and lists $[\vec{e}]$ are only final in **deep** mode when all members $\overline{\alpha}$ and elements $\vec{e}$ are recursively final.

The rule SEQ-FINAL () ensures that we only reduce **seq**/$\mu\ e_1\ e_2$ to e_2 once the expression e_1 is final for mode μ. To let reduction occur in the first operand of **seq** (and other operators) we use evaluation contexts [19]. We index evaluation contexts $K_\mu^{\mu'}$ () with an input μ and output μ' mode, similar to how evaluation contexts in CompCertC [35] are indexed by an l-value and r-value kind. The rule CTX () expresses that $K_\mu^{\mu'}[e]$ can take a step in μ mode if e can take a step in μ' mode. The evaluation context $[\vec{e_1}, \square, \vec{e_2}]$ for deep evaluation of lists requires all expressions in $\vec{e_1}$ to be final, ensuring that reduction goes in left-to-right direction. For example, take **seq**/**deep** $[1+2,\ \Omega]$ **true** $\rightarrow_{\textbf{shallow}}$ **seq**/**deep** $[3, \Omega]$ **true** $\rightarrow_{\textbf{shallow}}$ **seq**/**deep** $[3, \Omega]$ **true** $\rightarrow_{\textbf{shallow}} \cdots$, where $\Omega \coloneqq (\lambda\, x.\, x\ x)\ (\lambda\, x.\, x\ x)$. The **seq**/**deep** forces us to deeply evaluate $[1+2, \Omega]$ (which must be evaluated in order). But since Ω loops, the deep evaluation of the list and thereby the entire program also loop.

The evaluation context $\{\textbf{nonrec}\ \overline{d}, x \coloneqq \textbf{nonrec}\ \square\}$ for deep evaluation of attribute sets is more complicated. Recall `deepSeq { x = Omega; y = 0 0; } true`, which either diverges or faults depending on the order on names. To account for this behavior, we parameterize our semantics (and interpreter) by a strict order $(\sqsubset) \subseteq \mathrm{Str} \times \mathrm{Str}$ on strings, which is used to select which member is evaluated first.

Operators. Recall that Nix's binary operations are lazy in their second operand. We therefore should only allow e_2 to take a reduction step in $e_1 \circledcirc e_2$ at the moment that e_1 is final (*i.e.,* fully reduced) and we know that e_1 is a valid operand for the operator $\circledcirc$. To account for this laziness, we give a semantics to binary operators using the judgment $e_1 \Downarrow^{\circledcirc} \Phi$ (). If e_1 is an *invalid input* for the first operand of $\circledcirc$, then $e_1 \Downarrow^{\circledcirc} \Phi$ simply does not hold. For example, **true** $\Downarrow^{\bullet} \Phi$ does not hold for any relation Φ because attribute selection (.) is not defined on Booleans. If e_1 is a *valid input* for the first operand of $\circledcirc$, then $e_1 \Downarrow^{\circledcirc} \Phi$ gives a binary relation $\Phi \subseteq \mathrm{Expr} \times \mathrm{Expr}$ that assigns outputs to inputs for the second operand. For example, BINOP-SELECT-ATTR () assigns the relation $\{(s, e) \mid s \coloneqq e \in \overline{e}\}$ to the selection operator (.) on an attribute set $\{\overline{\textbf{nonrec}\ e}\}$. The use of the judgment becomes most clear from BIN-OP (), where $e_1 \circledcirc e_2$ reduces to e if there exists a relation Φ with $e_1 \Downarrow^{\circledcirc} \Phi$ and $\Phi\ e_2\ e$. In the evaluation context $e_1 \circledcirc \square$ for binary operators, we require there to exist a Φ with $e_1 \Downarrow^{\circledcirc} \Phi$ so that reduction in the second operand only happens if the first operand is a valid input.

Recall that the equality operator == compares lists and attribute sets recursively. Inspired by the Nix implementation [45, `src/libexpr/eval.cc`, `EvalState::eqValues`], the rules BINOP-EQ-LIST (🏴) and BINOP-EQ-ATTR (🏴) expand the comparison operators on lists and attribute sets into a series of equalities on their members/elements, conjoined with the lazy && operator. Note that if the lengths/domains are different, these rules immediately give **false**. As usual, care has to be taken due to the order attribute members. The rule BINOP-EQ-ATTR therefore generates a series of equalities based on the order ($\sqsubset$) $\subseteq$ Str $\times$ Str by which our semantics is parameterized.

Integers and floats. To model 64-bit signed integers, we check that integers are in bounds after every binary operation, see *e.g.,* BINOP-ADD. We use the Flocq library [9] to support 64-bit binary IEEE 754 floats. Flocq is highly configurable, with many settings for *e.g.,* NaNs and rounding, so we tried our best to make these settings match with the Nix implementation (🏴). Nix has implicit casts from integers to floats, which are handled in NixLang by overloading the semantics for binary operators, *i.e.,* by adding rules for int/float and float/int inputs to $e_1 \Downarrow^{\circledcirc} \Phi$ (rules not shown here).

4.3 Implementation of the Interpreter

The environment-based interpreter for NixLang is shown in Figure 6. By convention, all interpreter functions that take a fuel parameter δ time out when $\delta = 0$. Similarly, functions fail when no case applies or a pattern in 'do notation' does not match. We discuss the most important aspects of the interpreter in the following.

Data structures. Entries in the environment contain a *kind*, which tracks whether a variable binding belongs to a `let` construct/lambda abstraction (kind **abs**) or a `with` construct (kind **with**). The merge operator on environments, which is used in the interpretation of the **let** construct, ensures that **with** bindings cannot shadow **abs** bindings:

$$E_1 \sqcup E_2 \coloneqq \{x \coloneqq (k, t) \in E_1 \mid k = \textbf{abs} \vee x \coloneqq (\textbf{abs}, \_) \notin E_2\} \cup E_2$$

Compared to the interpreters in § 2 and 3, the Thunk data structure needs to extended (🏴). Thunks can either be a forced value (forced v), a suspended computation ($\text{thu}_E\, e$), or the selection of a recursive attribute set ($\text{ind}_E\, \overline{\alpha_t}.x$). We need to explicitly consider forced values to support `__functor`. Let us consider `{ __functor = r: x: e1; } e2`. Here, the interpreter first evaluates the attribute set (which could be the result of an arbitrary computation) to a value. That value then needs to be bound to the variable `r` in the environment used for the interpretation of `e1`.

The Val structure also needs to be extended (🏴). Values can be base literals (b), closures ($\text{clo}_E\, x.\, e$ and $\text{clo}_E\, m.\, e$, for ordinary and matching lambda, respectively), lists ($[\vec{t}]$), or attribute sets ($\{\bar{t}\}$). The elements $\vec{t}$ of list values and the members $\bar{t}$ of attribute values are thunks because of laziness.

Mutually-recursive definition of the interpreter functions. The interpreter for expressions $[\![e]\!]^E_\delta$ (🏴) has the same signature as the simple interpreters from § 2 and 3. Aside from the complexities of Nix, the conceptional differences can be found in the cases for variables (x), application ($e_1\ e_2$) and `deepseq` (**seq**/**deep**), for which we use three additional functions, which are defined in a mutually-recursive manner with the interpreter itself.

The interpreter for thunks $\mathcal{T}[\![t]\!]_\delta$ (🏴) forces a thunk t into a value. Its primary use is in the variable case (x). If the thunk is already forced (forced v), it is a no-op. If the thunk is a suspended computation ($\text{thu}_E\, e$), it recursively calls the interpreter on e. If the thunk is the selection of a recursive attribute set ($\text{ind}_E\, \overline{\alpha_t}.x$), it looks up the member x in $\overline{\alpha_t}$ and recursively calls the interpreter.

The interpreter for applications $\mathcal{A}[\![v\ @\ t]\!]_\delta$ (🏴) computes the result of the application $v\ t$. We use a separate function because unlike the interpreters in § 2 and 3, Nix has multiple constructs that can be used as functions (and thus appear as the first operand of an application). The first operand of an

Interpreter: $\boxed{[\![e]\!]^E_\delta = r}$

$$
\begin{aligned}
[\![b]\!]^E_\delta &:= \text{guard (base\_lit\_ok } b);\ \text{ret } b \\
[\![[\vec{e}]]\!]^E_\delta &:= \text{ret } [\text{thu}_E\ e \mid e \in \vec{e}] \\
[\![\{\overline{\alpha}\}]\!]^E_\delta &:= \text{let } \overline{\alpha_t} = \{y := \textbf{nonrec}\ (\text{thu}_E\ e) \mid y := \textbf{nonrec}\ e \in \overline{\alpha}\} \cup \\
&\qquad\qquad \{y := \textbf{rec}\ e \mid y := \textbf{rec}\ e \in \overline{\alpha}\} \text{ in} \\
&\quad \text{ret } (\{y := \text{thu}_E\ e \mid y := \textbf{nonrec}\ e \in \overline{\alpha}\} \cup \\
&\qquad \{y := \text{thu}_{\text{indirects\_env } E\ \overline{\alpha_t}}\ e \mid y := \textbf{rec}\ e \in \overline{\alpha}\}) \\
[\![x]\!]^E_\delta &:= t \leftarrow E\ x;\ \mathcal{T}[\![t]\!]_{\delta-1} \\
[\![\lambda\, x.\, e]\!]^E_\delta &:= \text{clo}_E\ x.\, e \\
[\![\lambda\, m.\, e]\!]^E_\delta &:= \text{clo}_E\ m.\, e \\
[\![e_1\ e_2]\!]^E_\delta &:= v_1 \leftarrow [\![e_1]\!]^E_{\delta-1};\ \mathcal{A}[\![v_1 \mathbin{@} \text{thu}_E\ e_2]\!]_{\delta-1} \\
[\![\textbf{let}/k\ e_1\ \textbf{in}\ e_2]\!]^E_\delta &:= \{\vec{t}\} \leftarrow [\![e_1]\!]^E_{\delta-1};\ [\![e_2]\!]^{\{y:=(k,t)\mid y:=t\in\vec{t}\}\sqcup E}_{\delta-1} \\
\left[\!\!\left[\begin{array}{l}\textbf{if } d \textbf{ then } e_1 \\ \textbf{else } e_2\end{array}\right]\!\!\right]^E_\delta &:= b \leftarrow [\![d]\!]^E_{\delta-1};
\begin{cases} [\![e_1]\!]^E_{\delta-1} & \text{if } b = \textbf{true} \\ [\![e_2]\!]^E_{\delta-1} & \text{if } b = \textbf{false} \end{cases} \\
[\![e_1 \circledcirc e_2]\!]^E_\delta &:= v_1 \leftarrow [\![e_1]\!]^E_{\delta-1};\ f \leftarrow \mathcal{B}[\![v_1]\!]^{\circledcirc};\ v_2 \leftarrow [\![e_2]\!]^E_{\delta-1};\ t_2 \leftarrow f\ v_2;\ \mathcal{T}[\![t_2]\!]_{\delta-1} \\
[\![\textbf{seq}/\mu\ e_1\ e_2]\!]^E_\delta &:= \begin{cases} v_1 \leftarrow [\![e_1]\!]^E_{\delta-1};\ \text{force\_deep}_{\delta-1}\ v_1;\ [\![e_2]\!]^E_{\delta-1} & \text{if } \mu = \textbf{deep} \\ v_1 \leftarrow [\![e_1]\!]^E_{\delta-1};\ [\![e_2]\!]^E_{\delta-1} & \text{if } \mu = \textbf{shallow} \end{cases} \\
\text{force\_deep}_\delta\ v &:= \begin{cases} \vec{w} \leftarrow \text{list.mapM force\_thunk}_\delta\ \vec{t};\ [\text{forced } w \mid w \in \vec{w}] & \text{if } v = [\vec{t}] \\ \overline{w} \leftarrow \text{map.mapM force\_thunk}_\delta\ \overline{t};\ \{x := \text{forced } w \mid x := w \in \overline{w}\} & \text{if } v = \{\overline{t}\} \\ v & \text{otherwise} \end{cases} \\
\text{force\_thunk}_\delta\ t &:= v \leftarrow \mathcal{T}[\![t]\!]_{\delta-1};\ \text{force\_deep}_{\delta-1}\ v
\end{aligned}
$$

Interpreter (thunks): $\boxed{\mathcal{T}[\![t]\!]_\delta = r}$

$$
\begin{aligned}
\mathcal{T}[\![\text{forced } v]\!]_\delta &:= v \\
\mathcal{T}[\![\text{thu}_E\ e]\!]_\delta &:= [\![e]\!]^E_{\delta-1} \\
\mathcal{T}[\![\text{ind}_E\ \overline{\alpha_t}.x]\!]_\delta &:= \alpha_t \leftarrow \overline{\alpha_t}\ x;
\begin{cases} [\![e]\!]^{\text{indirects\_env } E\ \overline{\alpha_t}}_{\delta-1} & \text{if } \alpha_t = \textbf{rec}\ e \\ \mathcal{T}[\![t]\!]_{\delta-1} & \text{if } \alpha_t = \textbf{nonrec}\ t \end{cases}
\end{aligned}
$$

Interpreter (application): $\boxed{\mathcal{A}[\![v \mathbin{@} t]\!]_\delta = r}$

$$
\begin{aligned}
\mathcal{A}[\![\text{clo}_E\ x.\, e \mathbin{@} t_2]\!]_\delta &:= [\![e]\!]^{E\langle x:=(\textbf{abs},t_2)\rangle}_{\delta-1} \\
\mathcal{A}[\![\text{clo}_E\ m.\, e \mathbin{@} t_2]\!]_\delta &:= \{\overline{t}\} \leftarrow \mathcal{T}[\![t_2]\!]_{\delta-1};\ \overline{\alpha_t} \leftarrow \text{match } \overline{t}\ m;\ [\![e]\!]^{\text{indirects\_env } E\ \overline{\alpha_t}}_{\delta-1} \\
\mathcal{A}[\![\{\overline{t}\} \mathbin{@} t_2]\!]_\delta &:= t \leftarrow \overline{t}\ \text{"\_\_functor"};\ v_t \leftarrow \mathcal{T}[\![t]\!]_{\delta-1}; \\
&\qquad v \leftarrow \mathcal{A}[\![v_t \mathbin{@} \text{forced } \{\overline{t}\}]\!]_{\delta-1};\ \mathcal{A}[\![v \mathbin{@} t_2]\!]_{\delta-1}
\end{aligned}
$$

Data structures:

$$
\begin{aligned}
&\text{Env} \ni E := \text{Str} \xrightarrow{\text{fin}} \text{Kind} \times \text{Thunk} \qquad \text{Val} \ni v ::= b \mid \text{clo}_E\ x.\, e \mid \text{clo}_E\ m.\, e \mid [\vec{t}] \mid \{\overline{t}\} \\
&\text{Thunk} \ni t ::= \text{forced}\, v \mid \text{thu}_E\ e \mid \text{ind}_E\ \overline{\alpha_t}.x \qquad \text{TAttr} \ni \alpha_t ::= \textbf{rec}\ e \mid \textbf{nonrec}\ t
\end{aligned}
$$

Fig. 6. The interpreter for NixLang. (The base cases for Timeout/fail are elided.)

application can either be an ordinary closure ($\text{clo}_E\, x.\, e$), a matching closure ($\text{clo}_E\, m.\, e$), or an attribute set with a `__functor` member ($\{\overline{t}\}$). The interpreter for applications has a non-trivial recursive structure because `__functor` members can be nested, *i.e.,* `{ __functor = { __functor = ...; }; }`.

The function force_deep$_\delta\, v$ (🚁) is used in the interpretation of **seq**/**deep**, and recursively forces all thunked list elements and attribute members in v. The function makes use of the well-known monadic combinator list.mapM : $(A \to \text{Res}\ B) \to \text{List}\ A \to \text{Res}\ (\text{List}\ B)$, which maps a monadic action in left-to-right direction over a list. The function map.mapM is similar, and uses the order $(\sqsubset) \subseteq \text{Str} \times \text{Str}$ by which our semantics is parameterized.

Operators and matching. In the interpretation of binary operators ($e_1 \circledcirc e_2$) we call the interpreter for binary operators $\mathcal{B}[\![v_1]\!]^{\circledcirc}$ (🚁) on the result value v_1 of e_1. This interpreter returns an Option (Val → Option Val) to account for the fact that operators are lazy in their first operand. It returns None if the operator $\circledcirc$ does not support the first operand v_1, or Some f where f is a function that takes the value of the second operand and produces the result of the operator. In the interpretation of matching closures ($\text{clo}_E\, m.\, e$) we call match (🚁), which is an algorithmic version of the matching relation $m \sim \overline{d} \rightsquigarrow \overline{\alpha}$ that is used in the operational semantics.

Recursive attribute sets and defaults. To support recursive attribute sets there is a fair amount of map surgery in the interpretation of attribute sets ($\{\overline{\alpha}\}$). Recursive attributes need to be encoded as thunks whose environment is extended with entries for the recursive selections, for which we use the following variant of the indirects function from the operational semantics on TAttr (🚁):

$$\text{indirects\_env}\ E\ \overline{\alpha_t} \coloneqq \{x \coloneqq \textbf{abs}\ (\text{ind}_E\ \overline{\alpha_t}.x) \mid x \in \text{dom}\ \overline{\alpha_t}\} \cup\!\!\!\cdot\ E$$

Like the indirects function from the operational semantics (Figure 4), this function brings the attributes of a recursive attribute set into scope, but produces an environment instead of a substitution. We use the indirect attribute selection thunk constructor $\text{ind}_E\ \overline{\alpha_t}.x$ to keep track of the environment E of the attributes $\overline{\alpha_t}$.

Comparison with the substitution-based semantics. Similarly to the call-by-name lambda calculus, the substitution-based semantics is more concise than the environment-based interpreter. This lack of conciseness is exacerbated—the definition of thunks becomes more complicated, and the interpreter consists of several mutually defined parts.

Aside from the use of substitutions or environments, there are also other differences. We set up our development so that the definitions for the operational semantics are inductive relations and the interpreter uses computable (monadic) functions. These definitions are often subtly different because they operate on expressions (operational semantics) or values/thunks (interpreter). Most prominent are matching and binary operators. The inductive definition of matching ($m \sim \overline{d} \rightsquigarrow \overline{\alpha}$) is very simple, whereas the computable function involves subtle map surgery. The handling of deep/shallow evaluation is also different, the operational semantics uses (kinded) evaluation contexts whereas the interpreter relies on force_deep being called correctly.

4.4 Soundness and Completeness

To state the main soundness and completeness theorem for both deep and shallow reduction, we lift the interpreter to a variant that takes the mode μ as an additional argument (🚁):

$$[\![e]\!]_\delta^{E,\mu} \coloneqq \begin{cases} v \leftarrow [\![e]\!]_\delta^E;\ \text{force\_deep}_\delta\ v & \text{if } \mu = \textbf{deep} \\ [\![e]\!]_\delta^E & \text{if } \mu = \textbf{shallow} \end{cases}$$

With this definition at hand, the main theorem has the same shape as the ones in § 2 and 3:

Theorem 4.1. *The NixLang interpreter is sound and complete w.r.t. the operational semantics for:*

(1) *terminating programs, i.e.,* $(\exists \delta.\, [\![e]\!]^{\emptyset,\mu}_{\delta} = \text{ret } s)$ *iff* $e \to^*_\mu s$ (), *and*
(2) *faulty programs, i.e.,* $(\exists \delta.\, [\![e]\!]^{\emptyset,\mu}_{\delta} = \text{fail})$ *iff* $(\exists e'.\, e \to^*_\mu e' \not\to_\mu \wedge \neg \text{final}_\mu\, e')$ (), *and*
(3) *diverging programs, i.e.,* $(\forall \delta.\, [\![e]\!]^{\emptyset,\mu}_{\delta} = \text{Timeout})$ *iff* $(\forall e'.\, e \to^*_\mu e' \implies \text{red}_\mu\, e')$ ().

Our proof involves similar helper lemmas as in § 2 and 3, but due to the additional features of NixLang, we need some additional ingredients. Since the interpreter is defined using a number of mutually-recursive functions, we prove variants of Lemmas 2.2 and 2.4 by mutual induction.

Lemma 4.2 (). *If* $[\![e]\!]^{E,\mu}_{\delta} = \text{Done } v^?$, *then there exists some* e' *such that* $e(\!|E|\!) \to^*_\mu e'$ *and if* $v^?$ *is* Some v, *then* $|v| = e'$; *or if* $v^?$ *is* None, *then* $e' \not\to_\mu$ *and* $\neg \text{final}_\mu\, e'$.

This lemma follows by proving the following properties mutually by induction on δ ():

(1) If $[\![e]\!]^E_\delta = \text{Done } v^?$, then there exists some e' such that $e(\!|E|\!) \to^*_{\textbf{shallow}} e'$ and if $v^?$ is Some v, then $|v| = e'$; or if $v^?$ is None, then $e' \not\to_{\textbf{shallow}}$ and $\neg \text{final}_{\textbf{shallow}}\, e'$.
(2) If $\mathcal{T}[\![t]\!]_\delta = \text{Done } v^?$, then there exists some e' such that $|t| \to^*_{\textbf{shallow}} e'$ and if $v^?$ is Some v, then $|v| = e'$; or if $v^?$ is None, then $e' \not\to_{\textbf{shallow}}$ and $\neg \text{final}_{\textbf{shallow}}\, e'$.
(3) If $\mathcal{A}[\![w \mathbin{@} t]\!]_\delta = \text{Done } v^?$, then there exists some e' such that $|w|\ |t| \to^*_{\textbf{shallow}} e'$ and if $v^?$ is Some v, then $|v| = e'$; or if $v^?$ is None, then $e' \not\to_{\textbf{shallow}}$ and $\neg \text{final}_{\textbf{shallow}}\, e'$.
(4) If $\text{force\_deep}_\delta\ w = \text{Done } v^?$, then there exists some e' such that $|w| \to^*_{\textbf{deep}} e'$ and if $v^?$ is Some v, then $|v| = e'$; or if $v^?$ is None, then $e' \not\to_{\textbf{deep}}$ and $\neg \text{final}_{\textbf{deep}}\, e'$.

Lemma 4.3 (). *If* $e_1 \to_\mu e_2$ *and* $[\![e_2]\!]^{\emptyset,\mu}_{\delta_2} = \text{Done } v_2^?$, *then there exist an optional value* $v_1^?$ *and a fuel value* δ_1 *such that* $[\![e_1]\!]^{\emptyset,\mu}_{\delta_1} = \text{Done } v_1^?$ *and* $|v_1^?| = |v_2^?|$.

This lemma follows by proving the following properties mutually by induction on $(\to_\mu)$ ():

(1) If $e_1 \to_\mu e_2$ and $\text{final}_{\textbf{shallow}}\ e_2$ and $[\![e_2]\!]^\emptyset_{\delta_2} = \text{Done } v_2^?$, then there exist an optional value $v_1^?$ and a fuel value δ_1 such that $[\![\mu]\!]^\emptyset_{\delta_1} e_1 = \text{Done } v_1^?$ and $|v_1^?| = |v_2^?|$.
(2) If $|w_1| \to_{\textbf{deep}} |w_2|$ and $\text{force\_deep}_{\delta_2}\ w_2 = \text{Done } v_2^?$, then there exist an optional value $v_1^?$ and a fuel value δ_1 such that $\text{force\_deep}_{\delta_1}\ w_1 = \text{Done } v_1^?$ and $|v_1^?| = |v_2^?|$.

As part of the above proofs, we need versions of Lemma 2.4 for all components of the interpreter (): $[\![e]\!]^E_\delta$, $\mathcal{T}[\![t]\!]_\delta$, $\mathcal{A}[\![w \mathbin{@} t]\!]_\delta$, and $\text{force\_deep}_\delta\ w$. That is, we need to show that for inputs related up to conversion, the components of the interpreter give outputs related up to conversion. Again, these properties are proved by mutual induction on the fuel value. To handle the cases for `deepSeq` in the proofs, we need some lemmas about the function force_deep. We prove that the result of $\text{force\_deep}_\delta\ v$ is final for **deep** mode (), and conversely that if a value is final for **deep** mode, then the function force_deep gives a related outcome (). Finally, we first need to prove soundness and completeness lemmas for the interpretation of binary operators $\mathcal{B}[\![v]\!]^{\circledcirc}$ and the match function. The proofs are mostly straightforward, but involve a fair number of cases.

Non-deterministic semantics. Instead of parameterizing our operational semantics by an order $(\sqsubset) \subseteq \text{Str} \times \text{Str}$ that specifies in which order the members of attribute sets are evaluated, we could have made a non-deterministic version (but have not done so). Since the interpreter needs to make a concrete choice for the evaluation order, this means that Items 2 and 3 of our soundness and completeness theorem would have to be weakened. Item 1 would not need to be weakened because non-deterministic evaluation of attribute sets cannot influence the result of terminating programs.

5 Frontend and Evaluation

We describe our frontend that turns Nix source programs into NixLang programs, and how we have used it to evaluate our Nix semantics on the official Nix language tests.

Frontend. We use the parser by Korzunov [30] (written in OCaml) to turn Nix source files into an AST. We use Rocq's extraction mechanism [36] to turn the NixLang data structures and interpreter into OCaml code. Our elaborator (also written in OCaml) transforms the Nix AST into an NixLang AST. Finally, we use the pretty printer by Korzunov [30] (together with some glue code) to transform the outputs of our interpreter into textual output that can be compared against expected test outputs. Finally, we have implemented a number of the Nix builtins using Nix itself.

Noteworthy features of our elaborator. Nix allows one to define attribute names with *string interpolation*, *i.e.,* arbitrary expressions may appear as attribute members, as long as they evaluate to strings. Furthermore, these expressions may refer to other members when used in recursive attribute sets. For example, `rec { x = "foo"; ${x + "bar"} = 10; }` evaluates to `{ foobar = 10; x = "foo"; }`. The dynamic attributes are evaluated after the rest of the attribute set, and may not describe members that already appear. Normal attributes cannot refer to dynamic attributes in recursive attribute sets for this reason, and neither can dynamic attributes refer to each other. Dynamic attribute members are evaluated in the order in which they appear. We elaborate dynamic attribute members into a sequence of operations that 'insert' members in NixLang.

In Nix, recursive attribute sets and `let` bindings are allowed to contain `inherit x` declarations, *e.g.,* `rec { ... inherit x; ... }`, which include a variable `x` from the enclosing scope in the attribute set. We elaborate `inherit x` into a member (`x` ≔ **nonrec** `x`) in NixLang. There is also a version `inherit (e) x`, which inserts a binding `x = e.x`. Unlike `inherit x`, the recursivity of the inserted binding depends on the attribute set in which it appears. Surprisingly, it is also allowed to write `inherit ${"x"}`, but not `inherit ${x}`. We make sure to handle such edge cases.

Nix allows writing `{ x.a = 10; x = { b = 20; }; }` as sugar for `{ x = { a = 10; b = 20; }; }`. This syntax has many edge cases, especially in combination with recursive attribute sets and dynamic attributes. We have done our best to replicate these edge cases by studying the Nix language tests and pull requests on GitHub (*e.g.,* [26]). We handle these edge cases by implementing an extra elaboration step on the Nix AST that takes care of unfolding attribute paths before the elaboration from Nix to NixLang, so that the latter never encounters attribute paths with more than one element.

Evaluation on the official Nix language tests. Nix (version 2.25.0) comes with 182 test files, 108 are supported by NixLang, and for 103 of which we agree. For 5 tests we disagree:

- (2 tests) Our interpreter is strictly call-by-name, which is much less efficient than Nix's native implementation which is lazy (*i.e.,* uses term sharing).
- (1 test) The Nix interpreter uses term sharing and pointer equality to detect cycles, *e.g.,* `deepSeq (let x = { y = x; }; in x) true` terminates in Nix, but diverges in NixLang. The cases in which term sharing is able to detect loops appear very ad-hoc.
- (2 tests) In some corner cases the semantics of `with` is more lazy in Nix than NixLang. Given `with r; e`, it sometimes happens that `r` is not evaluated at all in Nix (in NixLang, `r` is always shallowly evaluated to an attribute set). For instance, `with Omega; true` returns `true` in Nix, but diverges in NixLang. Interestingly, minor variations such as `with Omega; x` and `with { x = 10; }; with Omega; x` diverge in both Nix and NixLang. It is unclear how to accurately capture this lazy semantics in a principled core language like NixLang.

These disagreements manifest in the tests timing out, either due to exhausting the fuel value (*i.e.,* they perform too many reduction steps) or when running for longer than one minute.

There are plenty of tests that are simply out of scope. We have implemented the builtins that are relatively generic, but 27 of the Nix language tests depend on builtins that are very specific, such as performing SHA256 hashing, conversion to/from JSON and XML, operations on version strings, Flake references, and looking up the location of a term in sources. More generally, our interpreter

Table 1. LOC for our Rocq development (not including the OCaml frontend, nor blank lines and comments).

	Op. sem. (+ proofs)	Interpreter (+ proofs)	Tests	Extra	Total
Shared	—	—	—	—	304
LambdaLang	30 + 22	35 + 561	—	—	648
DynLang	33 + 26	40 + 391	—	143	633
EvalLang	121 + 26	43 + 439	26	—	655
NixLang	472 + 624	326 + 2599	150	368	4539
Total					6779

lacks support for I/O, file system paths, retrieving source locations, derivations and file system paths. Another 47 tests that fall into one of these categories have been ignored.

Instrumenting our interpreter using Bisect_ppx [5] while running it against the Nix language tests gives us 91.77% coverage for the interpreter code extracted from Rocq.

Program logic. As a proof of concept, we define a weakest preconditions-based program logic for total correctness of NixLang programs in terms of the operational semantics (🏴‍☠️):

$$\mathsf{WP}_\mu\ e\ \{x.\, P\} \coloneqq \exists e'.\, e \to_\mu^*\ e' \wedge \mathsf{final}_\mu\ e' \wedge P[x \coloneqq e]$$

From the operational semantics, we derive structural rules for WP, *e.g.,* for application (🏴‍☠️):

$$\frac{\mathsf{WP}_{\mathbf{shallow}}\ e_1\ \{e_1'.\, \mathsf{WP}_\mu\ (e_1'\ e_2)\ \{x.\, P\}\}}{\mathsf{WP}_\mu\ (e_1\ e_2)\ \{x.\, P\}}$$

We prove total correctness of the recursive program from § 4.1 that determines whether a number is even using recursive attribute sets (🏴‍☠️), the "__functor" attribute (🏴‍☠️), and recursion through default arguments (🏴‍☠️). For all these variants we prove that they return `true` iff `n` is even (assuming that `n` evaluates to some valid integer). We can also reason about open programs, for example:

```
let x = 1; in with e; with { y = 2; }; x == y
```

We prove that this program returns false for any non-recursive attribute set `e` (🏴‍☠️). Although not very difficult, the verification of these programs is a bit tedious because we have not implemented tactic support in Rocq for applying the WP rules and simplifying the resulting programs.

6 Rocq Mechanization

We give an overview of our Rocq development and demonstrate how the new `gmap` data structure from the Rocq-std++ library [32] can be used to represent syntax with nested recursion through finite maps, particularly deferred substitutions.

Overview of the Rocq development. Table 1 shows an overview of our Rocq development. For every language, the interpreter proofs encompass a soundness and completeness theorem w.r.t. the operational semantics, of which we separately prove properties (such as determinism). Since all languages already differ in their syntax, little reuse is possible. The row 'Shared' concerns the monad Res (Figure 2), a tactic to simplify monad equations, and some utilities for finite maps. For DynLang, we have two main proofs besides those of the properties of the operational semantics. The first part (391 LOC) is the soundness and completeness proof for the interpreter w.r.t. the operational semantics. The second part (143 LOC, counted under 'Extra') corresponds to the equivalence proof between LambdaLang and DynLang for closed LambdaLang terms. Despite DynLang being more complicated than LambdaLang, we observe that the soundness and completeness proof is 70%

in size due to the lack of boilerplate related to closedness conditions. Note that the operational semantics for EvalLang is significantly bigger than DynLang due to the parser that is present there (and is shared with the interpreter), which comprises of about 83 LOC. The extra part for NixLang concerns the program logic proof of concept and some examples, as shown in § 5.

The Rocq mechanisation of each language closely follows the structure on paper. Most proofs involve induction on the fuel value. For NixLang, we often need to prove a number of variants of a theorem in a mutual fashion (see § 4.4), for which we use the `Fixpoint lem1 ... with lem2 ...` pattern in Rocq. Key to most proofs is simplifying the interpreter according to its definition. We aimed to set up our definitions so that `simpl` is well-behaved (which sometimes poses challenges, see the remark about `subst_env` below). We make little use of custom proof automation, with the exception of a simple tactic `simplify_res` to simplify equations in the monad Res, and a simple tactic `inv_step` to repeatedly perform inversion on the reduction relation.

Nested recursion through finite maps. Finite maps play a central role in our mechanization, *e.g.,* to represent parallel substitutions, attribute sets, patterns in matching lambda abstractions, and environments. What makes the use of finite maps even more interesting is that they often occur in nested recursive positions, in the sense that the constructors of a data type contain a finite map whose elements contain the data type itself. For instance, the variable constructor $\${e}_{\overline{d}}$ in expressions contains a deferred substitution $\overline{d}$, which is a finite maps from strings to expressions themselves. Finite maps also occur in nested position to model attribute sets and thunks.

To make mechanization of our results feasible, we use the recently improved `gmap` data structure from the Rocq-std++ library [32], which provides some important features. First, it allows us to define the desired syntax and data structures without complaints from Rocq's positivity checker. Second, it allows us to define mutually/nested recursive functions (such as parallel substitution and the conversion from thunks to expressions) without complaints from Rocq's guardedness checker. Third, it provides suitable reasoning principles, such as extensional Leibniz equality on maps, the ability to prove the right induction principles, and many operations and lemmas to deal with the map surgery for recursive attribute sets in Nix. Fourth, it provides reasonable performance allowing us to run the interpreter, both inside of Rocq (using `vm_compute`) and when extracted to OCaml. To showcase these features, let us consider the definition of environments and thunks in DynLang:

```
Inductive thunk :=
  Thunk { thunk_env : gmap string thunk; thunk_expr : expr }.
Notation env := (gmap string thunk).
```

The definitions of these data types are in one-to-one correspondence with Thunk $\ni t ::= \mathrm{thu}_E\, e$ and Env $\ni E \coloneqq \mathrm{Str} \xrightarrow{\mathrm{fin}} \mathrm{Thunk}$ in Figure 3. The functions $|t|$ (which converts a thunk t into an expression) and $e\langle\!|E|\!\rangle$ (which performs a parallel substitution of an environment E in e by converting all thunks to expressions) would ideally be written to exactly match the definitions in § 2.3:

```
Fixpoint thunk_to_expr (t : thunk) : expr :=
  subst_env (thunk_env t) (thunk_expr t)
with subst_env (E : env) : expr → expr := subst (thunk_to_expr <$> E).
```

Unfortunately, Rocq does not allow us to use the syntax for mutually recursive functions on nested inductive data structures. In the actual definition we therefore first define the helper `subst_env'`, which takes `thunk_to_expr` as an argument, and define `subst_env` as notation:

```
Definition subst_env' (thunk_to_expr : thunk → expr)
  (E : env) : expr → expr := subst (thunk_to_expr <$> E).
Fixpoint thunk_to_expr (t : thunk) : expr :=
  subst_env' thunk_to_expr (thunk_env t) (thunk_expr t).
```

```
Notation subst_env := (subst_env' thunk_to_expr).
```

(We use `Notation` to make the `simpl` tactic behave well. Consider `thunk_to_expr (Thunk E e)`, the `simpl` tactic reduces this to `subst_env E e`. If `subst_env` were a `Definition`, then `thunk_to_expr (Thunk E e)` would reduce to `subst_env' thunk_to_expr E e`. That is, `simpl` would not refold `subst_env`.)

The next step, after having defined the data types and functions on them, is to carry out some proofs. An essential feature of the `gmap` data structure is its support for extensional equality on maps. That is, we have that two maps are equal, if they are element-wise equal:

```
Lemma map_eq (m1 m2 : gmap K A) : (∀ i, m1 !! i = m2 !! i) → m1 = m2.
```

This property holds (without axioms) because `gmap` is based on binary tries in canonical form [4]. Extensional equality is important for lemmas about our functions, for example:

```
Lemma subst_env_union E1 E2 e :
  subst_env (E1 ∪ E2) e = subst_env E1 (subst_env E2 e).
```

Recall from Figure 3 that parallel substitution in DynLang performs a left-biased union in the variable constructors $\${e}_{\vec{d}}$ in the syntax. Proving the lemma requires us to show that the finite maps in the variable constructors are the same, which is achieved using `map_eq`. Another important reasoning principle is induction. The induction principle on environments (which we actually do not use in practice, but we use more complicated induction principles for NixLang) is:

```
Lemma env_ind (P : env → Prop) :
  (∀ E, map_Forall (λ i, P ∘ thunk_env) E → P E) →
  ∀ E : env, P E.
```

This induction principle says that in order to prove a property of environments, we can assume it holds for the environments in all thunks (the induction hypothesis). To state the induction hypothesis, we use the `map_Forall` combinator from Rocq-std++. Proving this induction principle requires a couple of lines of boilerplate, involving the definition of a 'size' function on thunks and environments. It would thus be nice to automatically generate these induction principles using *e.g.*, Rocq-Elpi [53] or Template-Rocq [3] in the future.

The `gmap` data structure is based on the canonical binary trie data structure by Appel and Leroy [4], and is fairly efficient as far as purely functional data structures in a proof assistant go. Operations such as lookup, insert, and deletion are logarithmic in the size of the key (byte-length of the string). The data structure is fast enough to run our interpreter in Rocq (using `vm_compute`) and via extraction to OCaml on non-trivial test cases.

Finally, we point out that these features scale to more complicated languages, such as NixLang. Environments, thunks and values are mutually dependent, and defined as follows in Rocq:

```
Inductive val :=
  | VLit (bl : base_lit) (Hbl : base_lit_ok bl)
  | VClo (x : string) (E : gmap string (kind * thunk)) (e : expr)
  | VCloMatch (E : gmap string (kind * thunk))
              (ms : gmap string (option expr))
              (strict : bool) (e : expr)
  | VList (ts : list thunk)
  | VAttr (ts : gmap string thunk)
with thunk :=
  | Forced (v : val) : thunk
  | Thunk (E : gmap string (kind * thunk)) (e : expr) : thunk
  | Indirect (x : string)
```

```
            (E : gmap string (kind * thunk))
            (tαs : gmap string (expr + thunk)).
Notation env := (gmap string (kind * thunk)).
```

To define the operational semantics, interpreter, and to carry out our proofs we need to perform a good amount of surgery on recursive attribute sets. Fortunately, Rocq-std++ comes with plenty of operations on finite maps and lemmas about those to that make that goal feasible. For instance, we need operators that transform keys and values element-wise, we need to consider the domain as a finite set, and need to perform merging and biased unions.

7 Related Work

7.1 Explicit Substitutions

Up to our knowledge, we are the first to use ideas from the calculus of explicit subsitutions [1] to model dynamic languages. Lippmeier [37] also modifies the calculus of explicit substitutions by limiting the substitutions to appear only at abstractions, whereas we limit them to variables. Both approaches avoid the need for α-conversion when reducing open programs, but our approach scales to dynamic features such as `$`, `eval`, and `with`. The applications are also different, Lippmeier applies his approach to a proof of progress and preservation of simply-typed lambda calculus, whereas we apply it to verified interpreters of dynamic languages.

7.2 Prior Semantics of Nix

The Nix language was originally developed by Dolstra [16] as part of his PhD thesis, in which he described the Nix package manager. An integral part of his PhD thesis is the Nix language, for which he focused on the design, semantics and implementation. Subsequent papers by Dolstra and Löh [17] and Dolstra et al. [18] used Nix as the basis of the Linux distribution NixOS, but also presented variations of the Nix language and its semantics. This line of work used a substitution-based operational semantics, and already covered some of the key features of Nix, in particular recursive attribute sets by unfolding them one level.

Compared to the aforementioned work by Dolstra and collaborators, our paper only focuses on the semantics of the Nix language instead of its applications, but covers a much larger set of language features. Notably, we provide the most complete support for matching lambda abstractions (Dolstra [16] supports only non-recursive defaults, Dolstra et al. [18] support non-strict matchers, but no paper supports the combination nor recursive defaults), and investigate features that were not supported by any of their papers, *e.g.*, `__functor`, `deepSeq`, deep equality of lists and attribute sets, and IEEE floats. We also provide mechanized results in a proof assistant and test our semantics against the official language tests through a verified interpreter.

We repair various bugs in their semantics. The first bug is related to shadowing of `let`/`with`. For example, `with { x = 10; }; with { x = 12; }; x` returns 10 in Dolstra et al., whereas the official reference interpreter returns 12. Through deferred substitutions, we ensure that shadowing is handled correctly. The second bug is that their semantics cannot distinguish between non-terminating and faulty programs. This is most evident in their rule for the selection operator (.):

$$\frac{e \to^* \{\overline{\alpha}\} \qquad x \coloneqq e' \in \overline{\alpha}}{e.x \to e'}$$

Due to the big-step premise, their semantics gives a stuck/faulty behavior instead of a diverging behavior to `Omega.x`. We are able to distinguish stuck and diverging behaviors by giving an operational semantics that is fully small-step.

Table 2. Comparison with other semantics of dynamic languages (?: not clear, P: partial).

	Maffeis et al.	λ_{JS}	S5	JSCert	$\mathbb{K}$PHP	KJS	JaVerT	JSkel	NixLang
Binding	SO(C)s	Subst.	ERs	ERs	Env.	ERs	ERs	ERs	Def. subst.
Mechanization	–	Redex/Rocq	OCaml/Rocq	Rocq	$\mathbb{K}$	$\mathbb{K}$	OCaml	Skel	Rocq
Program logic	✓	–	–	P	✓	✓	✓	–	P
Closures, `eval`, `with`	✓✓✓	✓–✓	✓✓✓	✓✓✓	––?	✓✓✓	✓P✓	✓–✓	✓N/A✓
Language tests	–	✓	✓	✓	✓	✓	✓	–	✓
Correspondence	N/A	–	✓	✓	N/A	N/A	✓	N/A	✓

7.3 Semantics of Other Dynamic Languages

There is an abundance of work on the semantics of dynamic languages such as JavaScript [8, 25, 29, 39, 46, 50–52], PHP [20], Bash [41], and Posix Shell [23]. Table 2 contains an overview of the most relevant related projects. The row 'Language tests' corresponds to testing against test262 for JS (or the Mozilla test suite for λ_{JS}), the Zend test suite for PHP, and the Nix language tests. The row 'Correspondence' indicates if a (formal) proof between different kinds of the semantics exists (for NixLang, we consider the correspondence between the operational semantics and interpreter).

With the exception of λ_{JS} [25] (mechanized in PLT Redex and later in Rocq [24]), none of these projects consider a truly substitution-based semantics. λ_{JS} supports `with` by elaboration of source JavaScript (ES3). Variable names that appear under `with` are changed so that they first perform a lookup into the object associated with the `with` statement before checking the wider scope. A similar approach is used for Nix [11] as part of the transpilation to Nickel, a configuration language with compatability for Nix. The desugaring relies on the observation that one can statically determine which variables are bound by `let` binders and lambda abstractions. Hence one can rewrite variables that are otherwise free to a sequential lookup from the innermost `with` up unto the outermost `with` to find a binding associated with that variable, or fail if it is not bound. This way, one also gets the lazy behavior of `with` (see the last two tests that fail in § 5), *e.g.*, `with Omega; with { x = 10; }; x` is roughly desugared into (the `?` operator tests if a member is present):

```
if { x = 10; } ? x then { x = 10; }.x else
if Omega ? x then Omega.x else abort "unbound variable"
```

Here, `Omega` will not be evaluated, and so this program terminates successfully. But clearly, there is a trade-off between desugaring `with` and giving a native semantics (as done by Dolstra [16] and us) because the desugaring can blow up the size of the source program significantly.

Since programs provided to `eval` must have access to the outer scope, some mechanism is required that keeps track of all variables in the surrounding scope. Naive substitution does not do this, so it is not too surprising that λ_{JS} does not provide support for `eval`. For the other JavaScript semantics considered, which practically make use of Environment Records (ERs), `eval` is comparatively easy to implement, since the ERs keep track of the entire scope in one place. With EvalLang in § 3.4, we have shown how we can recover support for `eval` using a form of deferred substitutions, albeit in a language with much lower complexity than JavaScript.

Maffeis et al. [39] were the first to define an operational semantics of JavaScript using pen and paper. Their semantics closely follow the ECMAScript standard, version 3 (ES3). This means that, instead of substitution, Scope Object Chains are employed, which are comparable to modern-day ERs. Later, Gardner et al. [22] gave a program logic based on this semantics.

S5 [50] uses a core language that is substitution based, but performs desugaring to abstract away all JavaScript variables into object lookups, akin to an ER semantics. It is therefore marked as having an ER semantics instead of substitution-based semantics in Table 2. Mechanization was

originally done in PLT Redex, and an interpreter was written in OCaml. For a slightly modified version of S5, a correspondence proof in Rocq between the interpreter and operational semantics and partial proof of the desugaring mechanism was given by Materzok [40].

JSCert [8] is a Rocq development that consists of two parts: a mechanized semantics for ES5 (JSCert) and a reference interpreter (JSRef). JSCert is described on a high level, such that it can easily be compared with the ES specification. An advantage of such a specification is that it can be used to verify (desugaring to) more concise semantics. The correspondence entails the soundness proof of the JSRef interpreter with respect to JSCert. In his PhD thesis, Bodin [7] presents an (incomplete) program logic based on JSCert.

$\mathbb{K}$PHP [20] is a semantics of PHP in the $\mathbb{K}$ framework. KJS [46] uses the same approach. The semantics corresponds closely to the original specification while getting, *e.g.,* concrete and symbolic execution for free. This helps with the trustworthiness of the formalization. Proving properties about programs is also possible using a kind of Hoare logic. However, one does not get the full flexibility that one would have when working with a proof assistant, for instance, to verify soundness and completeness of different language specifications.

JaVerT [51, 52] is a framework for verifying the correctness of JavaScript (ES5 strict) programs. It consists of two parts: compilation to JSIL (an intermediate language) and verification with a Hoare-style logic. The top-level JavaScript semantics is based on JSCert. The JSIL interpreter is written in OCaml. JaVerT has a partial pen-and-paper correctness proof for the JS-2-JSIL compiler [51, §6.1], shown by Naudžiūnienė [44]. JaVerT allows proving properties about JavaScript programs using a kind of Hoare logic. However, it does not seem to be possible to use a proof assistant like Rocq in combination with it, particularly to verify the meta theory. The `eval` construct is only supported in direct, strict mode, hence it is marked as 'partial' in Table 2.

JSkel [29] focuses on defining the semantics of JavaScript using skeletal semantics. Language specifications written in Skel, such as JSkel, can automatically be translated into both Rocq (for a formalization) and OCaml (for an interpreter). Convenient here is the single source of truth, compared to, *e.g.,* NixLang having two specifications and a correspondence proof.

8 Conclusions and Future Work

We presented a form of deferred substitutions to give concise substitution-based semantics for 'dynamic'/'scripting' languages. We proved soundness and completeness of environment-based interpreters w.r.t. deferred substitutions, and applied our results to give the most comprehensive semantics of Nix to date, which we evaluated on the official Nix language tests. In future work it would be interesting to use differential testing [42] to compare our interpreter more thoroughly with the official Nix implementation. It would also be useful to investigate a lazy semantics and interpreter of Nix, instead of a call-by-name one. A lazy interpreter is more efficient (due to term sharing), but it also opens the door to support some features from Nix that we are missing, *e.g.,* equality of functions and cycle detection. One could also investigate the lazy semantics of `with` that we discovered in two of the Nix language tests (§ 5). Finally, one could investigate whether deferred substitutions could be applied to other languages, *e.g.,* JavaScript, Bash or Makefile. An important question is how to deal with mutation. Perhaps one could use the same approach as S5 [50], Iris [28] and RustBelt [27], where variables are substituted for references on the heap.

Data Availability Statement

The Rocq development for the languages LambdaLang (§ 2), DynLang and EvalLang (§ 3) and NixLang (§ 4), the elaborator from Nix to NixLang (written in OCaml) and the code to exercise the Nix language tests on the NixLang interpreter extracted from Rocq to OCaml (§ 5) can all be found in Broekhoff and Krebbers [10].

Acknowledgments

We thank the anonymous reviewers for their suggestions. This work is supported in part by ERC grant COCONUT (grant no. 101171349), funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1991. Explicit Substitutions. *JFP* 1, 4 (1991), 375–416. doi:10.1017/S0956796800000186

[2] Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *POPL*. 666–679. doi:10.1145/3009837.3009866

[3] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *ITP (LNCS, Vol. 10895)*. 20–39. doi:10.1007/978-3-319-94821-8_2

[4] Andrew W. Appel and Xavier Leroy. 2023. Efficient Extensional Binary Tries. *JAR* 67, 1 (2023), 8. doi:10.1007/S10817-022-09655-X

[5] Anton Bachin. 2023. Bisect_ppx. https://github.com/aantron/bisect_ppx

[6] Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.

[7] Martin Bodin. 2016. *Certified semantics and analysis of JavaScript*. Ph. D. Dissertation. Université Rennes 1, France.

[8] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudžiūnienė, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised JavaScript specification. In *POPL*. 87–100. doi:10.1145/2535838.2535876

[9] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *ARITH*. 243–252. doi:10.1109/ARITH.2011.40

[10] Rutger Broekhoff and Robbert Krebbers. 2025. Artifact for "Verified interpreters for dynamic languages with applications to the Nix expression language". doi:10.5281/zenodo.15839106

[11] François Caddet. 2023. Nix with; with Nickel. https://tweag.io/blog/2023-01-24-nix-with-with-nickel/

[12] Brian Campbell. 2012. An Executable Semantics for CompCert C. In *CPP (LNCS, Vol. 7679)*. 60–75. doi:10.1007/978-3-642-35308-6_8

[13] Arthur Charguéraud. 2012. The Locally Nameless Representation. *JAR* 49, 3 (2012), 363–408. doi:10.1007/S10817-011-9225-2

[14] Arthur Charguéraud. 2020. Separation logic for sequential programs (functional pearl). *PACMPL* 4, ICFP (2020), 116:1–116:34. doi:10.1145/3408998

[15] Nicolaas Govert de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae*, Vol. 75. 381–392. doi:10.1016/1385-7258(72)90034-0

[16] Eelco Dolstra. 2006. *The purely functional software deployment model*. Ph. D. Dissertation. Utrecht University, Netherlands. http://dspace.library.uu.nl/handle/1874/7540

[17] Eelco Dolstra and Andres Löh. 2008. NixOS: A purely functional Linux distribution. In *ICFP*. 367–378. doi:10.1145/1411204.1411255

[18] Eelco Dolstra, Andres Löh, and Nicolas Pierron. 2010. NixOS: A purely functional Linux distribution. *JFP* 20, 5-6 (2010), 577–615. doi:10.1017/S0956796810000195

[19] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. 1987. A Syntactic Theory of Sequential Control. *TCS* 52 (1987), 205–237. doi:10.1016/0304-3975(87)90109-5

[20] Daniele Filaretti and Sergio Maffeis. 2014. An Executable Formal Semantics of PHP. In *ECOOP (LNCS, Vol. 8586)*. 567–592. doi:10.1007/978-3-662-44202-9_23

[21] Murdoch Gabbay and Andrew M. Pitts. 2002. A New Approach to Abstract Syntax with Variable Binding. *FAC* 13, 3-5 (2002), 341–363. doi:10.1007/S001650200016

[22] Philippa Gardner, Sergio Maffeis, and Gareth David Smith. 2012. Towards a program logic for JavaScript. In *POPL*. 31–44. doi:10.1145/2103656.2103663

[23] Michael Greenberg and Austin J. Blatt. 2020. Executable formal semantics for the POSIX shell. *PACMPL* 4, POPL (2020), 43:1–43:30. doi:10.1145/3371111

[24] Arjun Guha, Claudiu Saftoiu, Spiridon Eliopoulos, Benjamin Lerner, and Joe Gibbs Politz. 2013. The LambdaJS GitHub repository. https://github.com/brownplt/LambdaJS

[25] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP (LNCS, Vol. 6183)*. 126–150. doi:10.1007/978-3-642-14107-2_7

[26] Ryan Hendrickson. 2024. Nix Pull Request #11294, parser-state: fix attribute merging. https://github.com/NixOS/nix/pull/11294

[27] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34. doi:10.1145/3158154

[28] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018), e20. doi:10.1017/S0956796818000151

[29] Adam Khayam, Louis Noizet, and Alan Schmitt. 2021. JSkel: Towards a Formalization of JavaScript's Semantics. In *JFLA*. 95–116. https://inria.hal.science/hal-03509431

[30] Denis Korzunov. 2018. A Nix code formatter written in OCaml using Menhir and OCamllex. https://github.com/d2km/nixformat/

[31] Robbert Krebbers. 2015. *The C standard formalized in Coq*. Ph. D. Dissertation. Radboud University Nijmegen, Netherlands.

[32] Robbert Krebbers. 2023. Efficient, Extensional, and Generic Finite Maps in Coq-std++. https://coq-workshop.gitlab.io/2023/abstracts/coq2023_finmap-stdpp.pdf Extended abstract at "The Coq Workshop 2023", see also https://gitlab.mpi-sws.org/iris/stdpp/-/merge_requests/461.

[33] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. doi:10.1145/3009837.3009855

[34] Jean-Louis Krivine. 2007. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation* 20, 3 (2007), 199–207. doi:10.1007/S10990-007-9018-9

[35] Xavier Leroy. 2009. Formal verification of a realistic compiler. *CACM* 52, 7 (2009), 107–115. doi:10.1145/1538788.1538814

[36] Pierre Letouzey. 2002. A New Extraction for Coq. In *TYPES (LNCS, Vol. 2646)*. 200–219. doi:10.1007/3-540-39185-1_12

[37] Ben Lippmeier. 2016. Don't Substitute into Abstractions. https://benl.ouroborus.net/papers/2016-dsim/lambda-dsim-20160328.pdf Unpublished manuscript.

[38] Andreas Lochbihler and Lukas Bulwahn. 2011. Animating the Formalised Semantics of a Java-Like Language. In *ITP (LNCS, Vol. 6898)*. 216–232. doi:10.1007/978-3-642-22863-6_17

[39] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. In *APLAS (LNCS, Vol. 5356)*. 307–325. doi:10.1007/978-3-540-89330-1_22

[40] Marek Materzok. 2016. Certified Desugaring of JavaScript Programs using Coq. Presented at CoqPL'16. http://arthur.chargueraud.org/events/coqpl2016/CoqPL_2016_paper_3.pdf, archived at [https://web.archive.org/web/20220302171941/http://www.chargueraud.org/events/coqpl2016/CoqPL_2016_paper_3.pdf]

[41] Karl Mazurak and Steve Zdancewic. 2007. ABASH: finding bugs in bash scripts. In *PLAS*. 105–114. doi:10.1145/1255329.1255347

[42] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf, archived at [https://web.archive.org/web/20230306000947/https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf]

[43] Alexey Muranov. 2017. Nix issue #1361, Language feature proposal: exclusive 'with'. https://github.com/NixOS/nix/issues/1361

[44] Daiva Naudžiūnienė. 2018. *An infrastructure for tractable verification of JavaScript programs*. Ph. D. Dissertation. Imperial College London, UK. https://vtss.doc.ic.ac.uk/publications/Naudziuniene2018Infrastructure.pdf

[45] NixOS contributors. 2025. The Nix GitHub repository. https://github.com/NixOS/nix/tree/2.25.0

[46] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: a complete formal semantics of JavaScript. In *PLDI*. 346–356. doi:10.1145/2737924.2737991

[47] Simon L. Peyton Jones. 1987. *The Implementation of Functional Programming Languages*. Prentice-Hall.

[48] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *PLDI*. 199–208. doi:10.1145/53990.54010

[49] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. 2024. Programming Language Foundations. In *Software Foundations*, Benjamin C. Pierce (Ed.). https://softwarefoundations.cis.upenn.edu/plf-current/index.html

[50] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. 2012. A tested semantics for getters, setters, and eval in JavaScript. In *DLS*. 1–16. doi:10.1145/2384577.2384579

[51] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript verification toolchain. *PACMPL* 2, POPL (2018), 50:1–50:33. doi:10.1145/3158138

[52] José Fragoso Santos, Petar Maksimović, Gabriela Cunha Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: compositional symbolic execution for JavaScript. *PACMPL* 3, POPL (2019), 66:1–66:31. doi:10.1145/3290379

[53] Enrico Tassi. 2019. Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq. In *ITP (LIPIcs, Vol. 141)*. 29:1–29:18. doi:10.4230/LIPICS.ITP.2019.29
[54] Jude Taylor. 2015. Nix issue #490, Scoping is unintuitive. https://github.com/NixOS/nix/issues/490
[55] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*. 427–440. doi:10.1145/2103656.2103709

Received 2025-02-27; accepted 2025-06-27