Bachelor's Thesis Computing Science

Formally defining the semantics for the Nix expression language

Rutger Broekhoff
s1083777

June 28, 2024

First supervisor/assessor:
dr. Robbert Krebbers

Second assessor
prof. dr. Herman Geuvers

Radboud University

Abstract

Nix is a purely functional package manager and NixOS is a Linux distribution that builds on Nix. Both packages for Nix and system configurations for NixOS make use of the Nix expression language: a lazy functional programming language. The Nix language may seem like a simple extension of the classic λ-calculus at first, but it has interesting semantics for bindings.

Although reduction rules for the language were defined when Nix was introduced in 2006, they have become outdated. Nowadays, the official Nix interpreter is the single source of truth; it is the definition of how the language should function. This makes it difficult for alternative implementations of the language to be created, or for many implementations to coexist while retaining compatibility.

Based on earlier work on the Nix language, we define a new 'core' Nix language, Mininix. We define revised operational semantics and solve issues present in the older semantics. Most importantly, we are fully complete in our description of the language to prevent ambiguity about how the language should function.

Together with our Mininix semantics, we also provide an interpreter. The goal of this interpreter is to function as a reference point for other implementations of Mininix, so that these can use it to verify their own behavior. The interpreter is verified to be sound and complete with regard to the semantics of Mininix that we define.

All our work is mechanized using the Coq proof assistant.

This work is licensed under a Creative Commons "Attribution 4.0 International" license.

The artifacts of this work are licensed under the 3-clause BSD license (SPDX short identifier: BSD-3-Clause).

Contents

1 Introduction **3**

2 A tour of the Nix language **7**
2.1 Simple and composite values and expressions 7
2.2 Functions and recursion 8
2.3 Sources of bindings 10
2.4 Functors 13
2.5 Miscellaneous peculiarities 14

3 Defining a simplified version of the Nix language **16**
3.1 Grammar 16
3.2 The prelude 19
3.3 Semantics 19

4 Building a verified interpreter for Mininix **33**
4.1 Mechanizing shared components 33
4.2 Mechanizing the semantics 35
4.3 Writing an interpreter 37
4.4 Verifying the interpreter 40

5 Related work **45**
5.1 Previous Nix semantics 45
5.2 Missing features 49
5.3 Bindings and substitution 50
5.4 Non-academic work on Nix 53

6 Conclusions **54**
6.1 Future work 54

A Confluence of Mininix **60**

Chapter 1

Introduction

Package managers are prolific on Unix-based systems. Users use package managers to install, update and remove software from their systems. Package managers are typically aware of dependencies between software packages. This way, dependencies can automatically be installed when the user requests the installation of some program.

However, package managers traditionally have two problems: lack of reproducibility and 'dependency hell'. Reproducibility means that, regardless of when or on what system you build a package, the same artifacts result (*i.e.*, the resulting files are equal on a binary level). Dependency hell refers to different software packages having conflicting version constraints for software they depend on.

Reproducibility issues typically occur when the build environment is not sufficiently isolated or when inputs cannot be specified sufficiently precise. Dependency hell is practically unavoidable when software (primarily shared libraries) is installed on a system-wide level, as package managers typically do not allow two versions of the same shared library installed at the same time.

Nix, as first introduced by Dolstra, De Jonge, and Visser [12] in 2004, aims to solve these problems with three core ideas:

- Builds can be isolated to such an extent that all inputs used to build some package uniquely identify the output of the build process. This way, the build process is a deterministic partial function from build inputs to build artifacts.
- Instances of a package can be identified not by the version of the package, but by the hash of all inputs used to build said package.
- Packages do not need to be installed on system-wide level like `/usr/bin` or `/usr/lib`, but instead can all be installed in their own folder, of which the name is contains the hash of the package as described above. All these folders can be placed in a single 'store' folder.

The last point may seem to make it impossible for programs to depend on each other, as their dependencies are not in traditional folders such as `/usr/bin`

and `/usr/lib`. Very simply put: Nix primarily avoids this problem by patching packages so that they directly refer to their dependencies by their store path. With this construction, Nix avoids dependency hell. Dolstra, De Jonge, and Visser [12, Fig. 4] give a good illustration about how the store works and how programs refer to their dependencies.

NixOS is a Linux distribution that tightly integrates with the Nix package manager. System configuration is described using the same language used to describe packages in Nix. System configuration is declarative: instead of describing what actions should be taken to activate certain programs as system services or to install/remove software, the user specifies what they want the state of the system to be. NixOS takes care that this desired state is reached [13]. A version of the system configuration is called a ‘generation’. If desired, users can revert their system to an older generation or boot into an older system generation from the bootloader [26]. This makes it very easy to go back to an earlier working system configuration if a newer configuration breaks the system.

The Nix expression language. Both packages[1] and NixOS configurations are written in a domain-specific language: the Nix expression language, or Nix language for short. This is a lazy language that builds on the λ-calculus. The language avoids side effects as much as possible.

Although the Nix language may seem like ‘just another functional language’, it has interesting semantics for constructs that introduce bindings. The language has native support for attribute sets: partial maps from names to values. The bindings in the attribute set can be brought into scope using the `with` keyword, as we can see below. However, as we can see in the following example, shadowing is not trivial:

```
let x = 1; in let x = 2; in x         gives 2
with { x = 1; }; in let x = 2; in x   gives 2
let x = 1; in with { x = 2; }; x      gives 1
with { x = 1; }; with { x = 2; }; x   gives 2
```

Interestingly enough, `with` does seem to shadow earlier `with`s, but it does not seem to shadow let bindings. The listed results of these programs are the output that the official Nix interpreter gives, but earlier descriptions of the semantics of the language disagree on the fourth example. Most prominent and complete is the dissertation in which Dolstra describes Nix [11]. Here, however, a different syntax for the let construct is used. Dolstra and Löh [13] introduce the let-in construct as used above, but leave definition of the substitution function as an exercise for the reader. Combining the substitution function from Dolstra [11] and the reduction rules from Dolstra and Löh [13], we get that the last example should give `1` instead of `2`.

[1] Nix jargon: ‘derivations’. We refer to derivations as packages because this is a more well-known term that suffices for our purposes.

Problems. In short, the semantics as described by Dolstra [11] and Dolstra and Löh [13] suffer from the following problems:

1. They use mixed big-step and small-step reductions in the rules, which is not very conventional. Most importantly, this makes it impossible to distinguish errors from non-termination.

2. It is assumed that static analysis of the closedness of terms is possible. We discuss why this assumption is problematic in section 5.1.1.

3. The semantics of some language features are incompletely defined. For example, the way equality is defined is ambiguous. The way the inclusion construct (`with`) is defined is also either incomplete or does not match the way it works with the modern Nix interpreter anymore.

4. Other than the issues with these semantics, there is the general issue of time: many years have passed since the last paper with a description of the semantics of the Nix language was published [10]. During these years, the language has been extended.

5. Although this is in no way common, there is no formal relationships between any description of the semantics of the language and the Nix interpreter.

Meanwhile, the user base of the Nix package manager has also greatly grown in size. New implementations aiming to be fully compatible with Nix have spawned, such as Tvix [3] and HNix [27].

Although the Nix reference documentation does provide an updated description of most features of the language [15], it still lacks reduction rules. The Tvix project has started working on a more detailed language specification [2], but this does not seem to be finished at the time of writing.

This inconsistency and lack of specification makes it difficult for new implementations of the Nix language, let alone Nix in general, to be created in such a way that the semantics are and remain consistent across the implementations.

Solutions. We can solve problems 1–4 by defining new, strictly small-step semantics for Nix that does not make incorrect assumptions about closedness, and that is fully complete. To ensure that we leave nothing open for interpretation, we can formalize these semantics using a proof assistant.

Problem 5 can be solved by creating a verified interpreter for Nix, similarly to how there exist verified compilers for ML (CakeML [19]) and C (CompCert [20]). This interpreter does not have to be complex; it can be a near one-to-one translation of the reduction rules. This interpreter could then serve as a starting point for others looking to create more efficient interpreters for the language; it could be used to quickly verify whether one properly understands the language, and if another interpreter indeed matches the behavior of this reference interpreter, without having to construct a full derivation tree. We can mechanically verify that

this interpreter is indeed sound and correct according to the semantics that we also define.

Contributions. We aim to, at least partially, fill the gap between the last academic work on the Nix language, and the language as it stands today. We provide:

- A (brief) survey of the Nix language as it stands today.
- Revised and modernized semantics (using evaluation contexts [16]) for a smaller version of the Nix language, Mininix. These semantics are formalized and mechanically verified to be strongly confluent.
- An interpreter for Mininix that is mechanically verified to be sound and complete.

Outline. We start by exploring the Nix language in chapter 2. In chapter 3, we define the 'core' Nix language, Mininix. We sketch the confluence proof for Mininix in appendix A. We mechanize the semantics of Mininix, as defined in chapter 3, in chapter 4. In chapter 4, we also discuss how we wrote and verified the reference interpreter for Mininix. We discuss related work, including the relation of the semantics for Mininix with those described earlier by Dolstra et al., in chapter 5. We conclude and describe future work in chapter 6.

Artifact availability. Our formalization of the semantics and interpreter for Mininix using Coq is available on Zenodo [8].

Chapter 2

A tour of the Nix language

In this chapter, we go over the Nix language in a few parts. We discuss most important language features, from simple values (section 2.1) and recursion (section 2.2) to assertions and functors (section 2.4). Constructs which were not discussed in Dolstra [11] and Dolstra and Löh [13] (due to them not being part of the language yet) are also covered, namely:

- pattern-matching functions (section 2.3, p. 10),
- fallback in selection with attribute paths (section 2.3, p. 10), and
- callable attribute sets (functors) (section 2.4, p. 13).

We also discuss the inclusion (`with`) construct in detail, as it is similar to the let-in construct, but has some significant semantic differences.

2.1 Simple and composite values and expressions

Before we discuss functions and recursion, it is good to be aware of the different types of values which can be used in the Nix language. The following program displays these:

```
{                                                ①
  ex1.foo = true -> false;                       ②
  ex2 = 2 + 1;
  ex3 = "test" + "ing";
  ex4 = [ "fizz" "buzz" ];                       ③
  ex5 = { foo = 1; bar = 2; } // { bar = 3; };   ④
}.ex1.foo
```

Listing 1: Simple and composite values and expressions in the Nix language.

This program returns `false`. With the knowledge that ⟨`true -> false`⟩ should indeed be read as an implication, this is relatively trivial. Some points of interest:

① The top-level definition we see here is called an attribute set. It functions as a partial map from some string (the relevant field) to an expression. Fields are accessed using the selection (dot) operator. All bindings ⟨`x = e`⟩ in the attribute set must be semicolon-terminated. This also includes the last binding.

② Nested attribute set definitions may be defined using syntactic sugar. While we see ⟨`ex1.foo = true -> false`⟩ here, this should be read as ⟨`ex1 = { foo = true -> false; }`⟩.

③ While it may be natural to read ⟨`"fizz" "buzz"`⟩ as `"fizz"` applied to `"buzz"`, Nix has different rules inside lists. Any top-level expression in the list that is not semicolon-terminated is treated as an entry in the list. If we do want to write function application inside a list, we have to group the relevant expression by surrounding it with parentheses.

④ The `//` operator signifies a right-biased non-recursive union/merge of two attribute sets. In this case, this means that evaluating ⟨`ex5.foo`⟩ would give `1` and evaluating ⟨`ex5.bar`⟩ would give `3`.

2.2 Functions and recursion

As a first example, let us take a look at how we can define (recursive) functions in the Nix language. In listing 2, we can see a very basic Nix program.

```
let binToString = n:
      if n == 0
      then "0"
      else if n == 1
           then "1"
           else binToString (n / 2) + (if isEven n then "0" else "1");
    isEven = n: n != 1 && (n == 0 || isEven (n - 2));
    test = { x, y ? attrs.x, ... } @ attrs:
      "x: " + x + ", y: " + y + ", z: " + attrs.z or "(no z)";
 in test { x = binToString 6; }
```

Listing 2: Functions and recursion in the Nix language.

The output of this program looks like this:

```
x: 110, y: 110, z: (no z)
```

How should this program be read?

Functions. The Nix language is a functional programming language based on the λ-calculus. Simple functions are written as ⟨`x: e`⟩, corresponding to the common mathematical notation $\lambda x.e$.[1] There are also functions which pattern-match on their input.

At the top level, we have a let-in construct which binds three variables: `isEven`, `binToString` and `test`. The definitions of `isEven` and `binToString` are more or less trivial: they are defined as simple functions and have a single parameter, namely `n`.

In the `test` function, we do not see such a singular parameter name, as we do for `isEven` and `binToString`. Instead, the function `test` pattern-matches on its argument. Here is what the different parts of the pattern mean:

`{ x, y ? attrs.x, ... } @ attrs`: Match on an attribute set. (This is the only type of pattern matching that the Nix language currently supports.)

`{ x, y ? attrs.x, ... } @ attrs`: Bind the entirety of the passed attribute set to the variable `attrs`. (*I.e.*, attributes which are not matched against in the pattern will still be present in this attribute set.) Instead of writing ⟨`@ attrs`⟩ after the pattern, it is also allowed to write ⟨`attrs @`⟩ before the pattern. Having both is not allowed.

`{ x, y ? attrs.x, ... } @ attrs`: The attribute set passed as the argument to this function *must* contain the attribute `x`. Bind this to the variable `x`.

`{ x, y ? attrs.x, ... } @ attrs`: The attribute set passed as the argument to this function *may* contain the attribute `y`. If it does, bind the corresponding value to the variable `y`. If the passed attribute set does not contain `y`, then bind `attrs.x` to the variable `y`.

`{ x, y ? attrs.x, ... } @ attrs`: The attribute set passed as the argument to this function *may* contain other attributes than `x` and `y`. Note: If this ellipsis is left out, the Nix evaluator will throw an error when an attribute set is passed which contains attributes which are not explicitly matched against.

The default values for attributes can be expressed in terms of different arguments matched against, or in terms of the entire passed attribute set, *e.g.*, writing `attrs.x` when the entire attribute set is bound using ⟨`@ attrs`⟩.

Once again, consider the example pattern `{ x, y ? attrs.x, ... } @ attrs`. Here, the default value for `y` is expressed in terms of the entire passed attribute set, which is bound to `attrs`. Because this pattern matches against `x`, we could have written `x` instead of `attrs.x` for the default value of `y`.

Moreover, the default value for an attribute can refer to itself. The following program gives `20`:

```
({ n, double ? (x: if x == 0 then x else double (x - 1) + 2) } : double n)
  { n = 10; }
```

[1] In Nix, the shortest function that can be written is the identity function: `x: x`. Omitting the space causes the expression to be parsed as a string instead of as a function.

Recursion. While we just demonstrated a special case of recursion in the Nix language, recursion is available more generally. As is likely apparent from the code in listing 2, let bindings make all bound variables available recursively, in all associated definitions (right-hand sides). As we can see in `isEven` and `binToString`, this makes recursion possible. The order of the bindings is irrelevant. Although not demonstrated here, it is indeed also possible to define mutually recursive functions in this manner.

Other than these language-specific recursion, there is that the Nix language is based on the classic λ-calculus. It is also lazy, so we can use the fixed point combinator **Y** [7, Corollary 6.1.3] in the Nix language too:

Y ≡ `f: (x: f (x x)) (x: f (x x))`

For example, the following program calculates the 20$^{\text{th}}$ Fibonacci number:

```
Y (go: x: if x <= 1 then x else go (x - 1) + go (x - 2)) 20
```

Selection. In the definition of `test`, we write ⟨`attrs.z or "(no z)"`⟩. Concretely, this has the following meaning: try to select attribute `z` from `attrs`, but use the value `"(no z)"` if it does not exist. This also works for nested attribute sets: you may also write ⟨`attrs.z.foo or "(no z.foo)"`⟩—this will give `"(no z.foo)"`, both when `attrs` does not have the attribute `z` and when `attrs.z` does not have the attribute `foo`. This means that we cannot trivially rewrite ⟨`attrs.z.foo or "(no z.foo)"`⟩ to ⟨`(attrs.z).foo or "(no z.foo)"`⟩.

2.3 Sources of bindings

So far, we have already seen the let-in constructs and recursion between bindings in the let-in construct. We have also discussed attribute sets. However, we have not yet discussed recursive attribute sets and inclusion: two other ways that variables can be bound.

Recursive attribute sets. For an example of recursion and inheritance in recursive attribute sets, see listing 3. The output of the listed program is

```
"foobarfoobarfoobar"
```

```
let test1 = {
      foo = "foo";
      bar = "bar";
    };
    test2 = rec {
      inherit (test1) foo bar;
      foobar = foo + bar;
      x = n: if n > 0 then x (n - 1) + foobar else "";
    };
 in test2.x 3
```

Listing 3: Inheritance and recursion in recursive attribute sets.

In the Nix language, attribute sets (written as `{ ` x ` = ` e`; ...}`) are not recursive by default. This means that x and y are free variables in e_1 and e_2 iff they are free variables in `{ ` x ` = ` e_1`; ` y ` = ` e_2`; }`.

If we define a recursive attribute set, we can refer to the values of all attributes in the attribute set from all attribute values. We do so by prefixing the attribute set definition with the keyword `rec`. In listing 3, we can see that the value for `x` can call itself and that it can also access `foobar`. Mutual recursion, although not demonstrated here, is also possible.

Problems appear with recursive attribute sets when we want to define some attribute x to be equal to the value of some variable x that is already in scope. Selecting x from `rec { ` x ` = ` x`; }` would cause infinite recursion. For that reason, the Nix language has the `inherit` construct, which inserts a non-recursive binding in a recursive attribute set and/or let binding. So instead, we would have written ⟨`rec { inherit x; }`⟩ here. Multiple space-separated arguments (names) can be used with `inherit`. So ⟨`inherit` x_1 `...` x_n⟩ is equivalent to having a non-recursive binding $\langle x_i = x_i \rangle$ for every $i \in [1, n]$.

Formally, `inherit` is syntactic sugar. Nevertheless, using `inherit` is the only way that a user of the Nix language can insert non-recursive bindings in recursive attribute sets and let statements. See Dolstra [11, Sec. 4.3.2, p. 74] for more details.

It is also possible to define which attribute sets the inherited bindings should come from. The syntax for that is ⟨`inherit (`x`)` y_1 `...` y_n⟩. But *hic sunt dracones*: for recursive attribute sets, this inserts a *recursive* binding $\langle y_i = x.y_i \rangle$ for every $i \in [1, n]$. So the following program gives `1`:

```
let h = rec { inherit ({ y = z; }) y; z = 1; }; in h.y
```

For non-recursive attribute sets, ⟨`inherit (`x`)` y_1 `...` y_n⟩ can also be used, but it will simply insert non-recursive bindings here.

Inclusion. As discussed earlier, NixOS is a Linux distribution which is based on the Nix package manager and the Nix language. System configuration is declarative, expressed in the Nix language. As an example, let us take a part of such a simple NixOS system configuration, as shown in listing 4.

```
{ pkgs, ... }:                                ①
  let sl = import ./derivations/sl.nix;       ②
   in {
        environment.systemPackages =
          with pkgs;                          ③
            [
              pkgs.vim                        uses ①
              git                             uses ① via ③
              sl                              uses ②
            ];
      }
```

Listing 4: A part of a simple NixOS configuration.

We see two new constructs here: `import` and `with`.

`import` simply loads a Nix file as an expression and replaces the import statement with the loaded expression, after performing various checks (*e.g.*, whether the loaded expression contains pattern-matching functions with duplicated formal parameters) [11, Sec. 4.3.4, p. 80]. Here, Dolstra also claims that a check for closedness is performed, but we believe that this is not possible. See section 5.1.1 for more details.

`with` (also known as inclusion) binds attributes from its first argument in its second argument. *I.e.*, the program ⟨`with { x = 1; }; x`⟩ will return `1`. Although the inclusion construct might seem similar to the let-in construct, it has different semantics. We discuss it below.

In the listing, the three places where bindings are created (variables are brought into scope) are annotated with a number. Their uses are also annotated accordingly. We attempt to provide some intuition for the way these bindings work below.

With the configuration in listing 4, the user wanted to define a system in which all users can access Vim and Git. They also want all users to have access to their customized version of a program called SL, which they patched to add some feature.[2] They have written their own derivation (read: package) for this patched version of SL. However, `sl` is also in the official Nix package collection, Nixpkgs (as passed to the configuration with the argument `pkgs`). Which package will be selected?

The definition of `sl` at ② is kept; it is not shadowed by the `sl` package in `pkgs`. We could say that the language chooses to let the inclusion construct be weaker than the let-in construct. Why should the language work this way?

[2] SL(1) (Steam Locomotive) is a joke program written by Toyoda Masashi, intended to amuse/annoy users when they mistype the well-known Unix command `ls`. Instead of listing the contents of the current working directory, the program will display a train that slowly drives across the screen in ASCII art, and prevent the user from terminating it until the train has driven off the screen (unless configured otherwise). The program can display different types of trains; for the sake of this exercise, we will assume that the user has patched SL to support displaying the Dutch National Railways' (NS) ICM train.

Imagine a world in which `sl` is not yet in Nixpkgs (`pkgs`). The user wants to install SL, but does not want to go through the process of submitting a package. Instead, they simply write their own derivation and import it, adding it to the list of system packages. So far, so good. However, one day, a new Nixpkgs release is made—and this release *does* include the SL package! What happens now? As we know from before: the code retains its original meaning. Indeed, imagine if it were not to: the sole removal or addition of a package in Nixpkgs could result in Nix programs showing unexpected behavior. That would be highly undesirable.

With this example, we can now clearly understand what the `with` construct: it brings variables into scope which would have otherwise been free. Its bindings are therefore weaker than bindings generated by let constructs, function parameters and recursive attribute sets.

Still, there is one catch: lower inclusions generate bindings stronger than higher-up inclusions. So `with { x = 1; }; with { x = 2; }; x` gives 2. As we describe in section 5.1.1, this is where the official Nix interpreter does not agree with the semantics described by Dolstra [11].

2.4 Functors

So far, we have covered the most important basic constructs in the language. Still, there is one interesting constructs that has been left uncovered.

```
let divide = a: b: assert a >= 0 && b > 0;
      if a < b then 0 else divide (a - b) b + 1;
    divider = {
      __functor = self: x: self // {
        value = divide self.value x;
      };
    };
    mkDivider = value: divider // { inherit value; };
 in (mkDivider 100 5 4).value
```

Listing 5: Functors: repeated natural number division.

Attribute sets become callable when they have a special attribute, namely `__functor`. Such attribute sets are called *functors*. The value of the `__functor` attribute must be a function that (1) takes the value of the attribute set itself (usually aptly named `self`) and (2) returns another function. These requirements are quite lax, and indeed show no practical resemblance to functors in classical terms, as used in other functional programming languages.

As an example, take the program in listing 5. Here, we can see that `divider` is bound to an attribute set which contains this `__functor` attribute. Whenever `divider` (the attribute set) is called with x, it updates its own attribute `value` to be the current `value` divided by x. Recall that the $\langle // \rangle$ operator signifies the right-biased merge of two attribute sets. We would break repeated application of the

functor without the merge with `self`, because we otherwise discard the `__functor` attribute.

We do see that the attribute set bound to `divider` does not define a `value` attribute. Without this, invocation will fail as `self.value` is used in the `__functor` definition. The `mkDivider` function addresses this by taking a value and initializing the `divider` with it.

The expression ⟨`(mkDivider 100 5 4).value`⟩ denotes the following: "Take the number 100, divide it by 5 and then by 4, then return the result." Accordingly, this program will indeed return `5`.

In the custom division function `divide` (used here instead of the division operator ⟨`/`⟩ for illustratory purposes), we can see an assertion. The assert construct has the following syntax: ⟨`assert` *condition*`;` *result*⟩. When evaluating an assert statement, the condition is checked first. If the condition does not evaluate to a Boolean or if it evaluates to false, the program fails. Otherwise, the result is returned.

Setting up recursion using functors. Using the functor mechanism, we have a way to set up recursion without using any recursive bindings (*i.e.*, without any recursive attribute sets and let statements) or the fixed point combinator **Y**. We give an example in listing 6, in which the Fibonacci function is defined. The result of the listed program is `610`, the same as the 15th Fibonacci number.

```
{ __functor = self: f: f (self f); }
  (go: n: if n <= 1 then n else go (n - 1) + go (n - 2))
    15
```

Listing 6: Functors: the Fibonacci function without recursive bindings.

2.5 Miscellaneous peculiarities

`null`, `true` and `false` are not keywords. Instead, they are constants which are in scope by default. (More precisely, they are nullary primitive operations [11, Sec. 4.3.4, p. 81].) They can be shadowed in recursive attribute sets and let bindings. For example,

```
let true = 42; in true == 42
```

gives the Boolean value `true`. The inclusion construct cannot shadow these because the bindings for builtins are equally strong as bindings generated by the let construct and function parameters. *I.e.*,

```
with { true = 42; }; in true == 42
```

gives the Boolean value `false`.

Functions are not comparable. Comparing two functions will always return false, except when comparing two attribute sets / lists in which there is a reference to exactly the same function definition. In this case, the comparison of these functions will be true. So for example,

```
let id = x: x; in id == id
```

and

```
[(x: x)] == [(x: x)]
```

both give `false`, but

```
let id = x: x; in [ id ] == [ id ]
```

gives `true`. In the official Nix interpreter, this depends on pointer equality. That makes hard to implement. The Tvix project has put in effort to document this behavior [3, `tvix/docs/src/value-pointer-equality.md`].

Mininix, which we describe in the following section, does not share all this behavior. Instead, it always gives false when two functions are checked for equality. We discuss future work in section 6.1.

Overrides. Attributes of a recursive attribute set can be updated with a magic `__overrides` attribute. For example,

```
rec { __overrides = { x = 1; }; x = 2; y = x; }
```

gives `{ x = 1; y = 1; }` when strictly evaluated. Recursive attribute sets in which `__overrides` does not reduce to a non-recursive attribute set cause the program to fail, *i.e.*, the following program will cause an error:

```
rec { __overrides = 1; }
```

This feature is hardly documented and we only discovered its existence by reading the language tests for the Nix interpreter [14, `tests/functional/lang`]. Mininix lacks support for this feature.

Let-body. We have seen the let-in construct before, but it also exists in a different form:

```
let { double = x: if x == 0 then 0 else body (x - 1) + 2; body = double; } 10
```

gives 20. The `body` attribute determines the output of the let-body construct. The code shown above can simply also be rewritten to the following:

```
rec { double = x: if x == 0 then 0 else body (x - 1) + 2; body = double; }.body 10
```

This is also the syntax Dolstra uses in his PhD dissertation [11]. This way of writing let bindings is not very common anymore, but is still supported. To keep things simple, we do not support this syntax in Mininix. However, conversion is trivial and can be done statically, as the argument provided to `let` may be no other expression than an attribute set: **let** $\{\, \overrightarrow{b_r} \,\}$ can simply be rewritten to **rec** $\{\, \overrightarrow{b_r} \,\}$`.body`.

Chapter 3

Ex Nixo Mininix fit: Defining a simplified version of the Nix language

In this chapter, we describe Mininix, a slightly simplified version of the modern Nix language. We provide detailed descriptions of the semantics of all supported features, in order to leave no room for different interpretations of how the language should function. Section 3.1 discusses the grammar of Mininix. We very quickly cover the prelude in section 3.2. Section 3.3 covers the semantics of Mininix, including substitution.

Mininix should be seen as a 'core language' for Nix. In that sense, Mininix is not intended as a language for programmers to work with; instead, it should be viewed as a language that Nix programs can be converted to, provided that the programs do not use any features that Mininix does not support. In section 5.1, we extensively discuss the differences between our semantics and the work of Dolstra et al. in describing the semantics of the Nix language.

3.1 Grammar

In fig. 3.1, we can see the grammar for Mininix. Most of its grammar is very similar to that of Nix, and particularly similar to the grammar by Dolstra and Löh [13, Fig. 5], which it was largely based on.

As we can see, values are defined as a subset of expressions. Values are expressions that are in normal form. We can also see that there are a few constructs that do not appear in Nix programs, but that we do have in Mininix: **force** e, $\mathsf{closed}(e)$, and $\mathsf{placeholder}_x(e)$.

While **force** e may be necessary when translating Nix programs to Mininix, $\mathsf{closed}(e)$ and $\mathsf{placeholder}_x(e)$ are purely run-time constructs that should not be need to be used when translating Nix programs to Mininix or when writing Mininix

Identifier	x, y
String	s
Integer	$n \in \mathbb{Z}$
Binding	$b ::= x := e$
Recursive binding	$b_r ::= b \mid x :=_r e$
Binary operator	$\circledcirc ::=$
Comparison	$\mid = \mid <$
Arithmetic	$\mid + \mid - \mid /$
Update	$\mid //$
Argument matcher	$m ::=$
Mandatory	$\mid x, m$
Default	$\mid x \ ? \ e, m$
Any	$\mid \ldots$
Empty	$\mid \varepsilon$
Value	$u, v \in \mathit{Value} ::=$
Literal	$\mid \textbf{true} \mid \textbf{false}$
	$\mid \textbf{null} \mid n \mid s$
Function	$\mid x : e \mid \{ \, m \, \} : e$
Attribute set	$\mid \{ \, b^* \, \}$

Expressions	$d, e \in \mathit{Expr} ::=$
Value	$\mid v$
Identifier	$\mid x$
Attribute set	$\mid \textbf{rec} \ \{ \, b_r^* \, \}$
Let binding	$\mid \textbf{let} \ b_r^* \ \textbf{in} \ e$
Must select	$\mid e.x^+$
Try to select	$\mid e.x^+ \ \textbf{or} \ e$
Application	$\mid e \ e$
Conditional	$\mid \textbf{if} \ e \ \textbf{then} \ e \ \textbf{else} \ e$
Inclusion	$\mid \textbf{with} \ e; \ e$
Assertion	$\mid \textbf{assert} \ e; \ e$
Operator	$\mid e \circledcirc e$
Has attribute	$\mid e \ ? \ x$
Strictly evaluate	$\mid \textbf{force} \ e$
Closed[RT]	$\mid \mathrm{closed}(e)$
Placeholder[RT]	$\mid \mathrm{placeholder}_x(e)$

[RT]: purely a run-time construct.

Figure 3.1: Syntax for Mininix.

$$
\begin{aligned}
x \ @ \ \{ \, m \, \} : e &\triangleq x : (\{ \, m \, \} : e) \ x \quad \text{if } x \notin \mathrm{dom}(m) \\
\{ \, m \, \} \ @ \ x : e &\triangleq x : (\{ \, m \, \} : e) \ x \quad \text{if } x \notin \mathrm{dom}(m) \\
\textbf{inherit} \ x_1 \ldots x_n &\triangleq x_1 := x_1; \ \ldots; \ x_n := x_n \\
\textbf{inherit} \ (e) \ x_1 \ldots x_n &\triangleq x_1 := e.x_1; \ \ldots; \ x_n := e.x_n \\
\textbf{inherit}_r \ (e) \ x_1 \ldots x_n &\triangleq x_1 :=_r e.x_1; \ \ldots; \ x_n :=_r e.x_n \\
\mathrm{cast}_{\mathrm{bool}}(e) &\triangleq \textbf{if} \ e \ \textbf{then true else false} \\
e_1 \neq e_2 &\triangleq \textbf{if} \ e_1 = e_2 \ \textbf{then false else true} \\
e_1 \ || \ e_2 &\triangleq \textbf{if} \ e_1 \ \textbf{then true else} \ \mathrm{cast}_{\mathrm{bool}}(e_2) \\
e_1 \ \&\& \ e_2 &\triangleq \textbf{if} \ e_1 \ \textbf{then} \ \mathrm{cast}_{\mathrm{bool}}(e_2) \ \textbf{else false} \\
e_1 \Rightarrow e_2 &\triangleq \textbf{if} \ e_1 \ \textbf{then} \ \mathrm{cast}_{\mathrm{bool}}(e_2) \ \textbf{else true} \\
!e &\triangleq \textbf{if} \ e \ \textbf{then false else true}
\end{aligned}
$$

Figure 3.2: Macros.

programs in general. See section 3.3.1 for more details on **force**. We explain the use of closed(e) and placeholder$_x(e)$ in more depth in section 3.3.4.

Notation. We use:

b^* to indicate a semicolon-separated list of non-recursive bindings that may be empty and where no variable on the left-hand side of any binding is used more than once;

b_r^* for the same purpose, but now also including potentially recursive bindings;

x^+ to indicate a period-separated list of identifiers that *may not be empty.*

$\mathrm{dom}(m)$ as the keys which the matcher m (optionally) matches against.

Macros. In fig. 3.2, we can see macros for Mininix. These define the expansion of syntactic sugar for Mininix. We choose to define more syntactic sugar than Dolstra [11] because this reduces the amount of constructs in the language. A smaller grammar primarily makes it easier to prove properties about the language. Besides, most syntactic sugar can trivially be verified to be correct, albeit manually.

We will now explain a select few macros that could benefit from clarification.

- $\langle x \mathbin{@} \{\, m \,\} : e \rangle$ denotes a function that takes an attribute set. The passed attribute set is matched with m. All explicitly matched members of the passed attribute set are brought into scope. The entire passed attribute set is also bound to x. See section 2.2 for a more detailed, practical example.

  In Nix, the name x may not be shared with a variable matched against in m. This causes an error. We encode this restriction as a condition for this macro.

  We may instead write $\langle x \mathbin{@} \{\, m \,\} : e \rangle$ as follows: $\langle x : (\{\, m \,\} : e)\ x \rangle$. Now, we have a function that takes a parameter x, the attribute set. This ensures that the entire attribute set is bound to x. Then, we invoke another function, $\langle \{\, m \,\} : e \rangle$, with this attribute set. This function now perform pattern-matching on its argument, and ensures that all explicitly matched members are bound in e. This way, both all explicitly matched members are brought into scope for e, and the entire attribute set is also bound to x in e. So $\langle x \mathbin{@} \{\, m \,\} : e \rangle$ has the same behavior as $\langle x : (\{\, m \,\} : e)\ x \rangle$.

- $\langle \{\, m \,\} \mathbin{@} x : e \rangle$ has the same meaning as $\langle x \mathbin{@} \{\, m \,\} : e \rangle$.

- $\langle \textbf{inherit}\ x_1 \ldots x_n \rangle$ is only used in the context of bindings. That is, it can only be used inside a let-in construct or an attribute set. The Nix language provides it to allow users to insert a non-recursive binding in a recursive attribute set. See section 2.3 for rationale.

Mininix does not strictly require **inherit**, because bindings inside attribute sets are explicitly annotated with their recursiveness. However, for clarity, we still keep **inherit** as a macro, which simply reduces to a non-recursive binding (or multiple bindings). So writing $\{$ **inherit** $x\ y\ \}$ is the same as writing $\{\ x \coloneqq x; y \coloneqq y\ \}$.

- **inherit** $(e)\ x_1 \ldots x_n$ is like **inherit** as described above. Only here, all variable names listed must be members of the attribute set that e reduces to. This matches the behavior of `inherit` in Nix in the context of a non-recursive attribute set. See section 2.3 for more details.

- **inherit**$_r$ $(e)\ x_1 \ldots x_n$ is like the case described above, but instead of inserting non-recursive bindings, it inserts recursive bindings. This matches the behavior of `inherit` in Nix in the context of a recursive attribute set. Again, we refer to section 2.3 for more details.

3.2 The prelude

As shown in fig. 3.1, **true**, **false** and **null** are all values. However, these should be considered *language-internal.* That is, users should not be able to directly access them. In the Nix language, it is also not possible to directly access the language-internal representation of true. Users of the Nix language can make use of `true`, `false` and `null`, which are predefined primitive operations (builtins) that give the language-internal value.

In Nix, there are two classes of builtins. Those that are immediately available (*e.g.*: `true`, `false`, `null`), and those that are only available in `builtins` (*e.g.*: `currentTime`, `getEnv`) [15]. All builtins in the former class are also available in the latter, but not vice-versa. So one can use `true` and `builtins.true` interchangeably, but writing `getEnv` will not work while writing `builtins.getEnv` will. Although the builtins that we define are all in the first class, we still explicitly illustrate these two classes in our prelude definition.

For Mininix, we only define the builtins 'true', 'false' and 'null'. To make these available in programs, we define the prelude. Its definition can be seen in fig. 3.3. For some program e, the prelude can be used by substitution with the prelude: $e[\text{closeds}(\text{prelude})]$. Or equivalently, by a let binding: $\langle$**let** prelude **in** $e\rangle$. Note that this means that these constants can be shadowed by user-defined bindings. For example, the program $\langle$**let** `true` $\coloneqq_r 42$ **in** `true` $= 42\rangle$ returns **true**. This is also possible in the Nix language.

3.3 Semantics

There are a good number of components to the semantics of Mininix. In fig. 3.5, we can see the reduction rules for Mininix. Note that this figure consists of two parts. In fig. 3.7, we can see the evaluation contexts [16] used in the reduction

$$\begin{aligned} \text{prelude} \quad \equiv \quad & \text{true} \coloneqq \textbf{true};\ \text{false} \coloneqq \textbf{false};\ \text{null} \coloneqq \textbf{null}; \\ & \text{builtins} \coloneqq \{\, \text{true} \coloneqq \textbf{true};\ \text{false} \coloneqq \textbf{false};\ \text{null} \coloneqq \textbf{null} \,\} \end{aligned}$$

Figure 3.3: Prelude.

relation. Figure 3.8 shows the $\overset{\circledcirc}{\rightarrow}$ relation, which defines how binary operations should be evaluated. Finally, fig. 3.9 shows the $\rightsquigarrow$ relation, which is used to match attribute sets to the patterns of functions that pattern-match on their attribute set argument ($\{\, m \,\} : e$).

In fig. 3.10, we can see the substitution function $e[\,\overrightarrow{b}\,]$. We will first discuss strict evaluation in section 3.3.1. In section 3.3.2, we cover the reduction rules. Finally, we discuss the substitution function in section 3.3.4.

3.3.1 Strict evaluation

Normally, Mininix programs reduce no further than values. fig. 3.1 describes what a value is. Non-recursive attribute sets are among the values. This means that non-recursive attribute sets cannot be reduced. However, this is not always desirable.

In the Nix language, equality checks are deep. In practical terms, this means that attribute sets are compared recursively. However, attribute sets in Mininix are values. *I.e.*, they are normal forms. This is problematic if we want to translate Nix programs to Mininix. To overcome this problem, we also need the values of the attributes to be reduced, so we can properly compare them.

To facilitate this and possibly other use cases, we have introduced the **force** construct into the language. Writing **force** e will return v_s, the strong value resulting from the strict evaluation of e. The definition of strong values can be seen in fig. 3.4. Strong values are not defined as a separate syntactic category; instead, these values are a strict subset of values in which attribute values must also be strong values.

Now, if we would have some expression $\langle$ `x = ` e_1`;  =  x = ` e_2`;` $\rangle$ in the Nix language, then we should write this as $\langle\textbf{force}\ \{\, \mathsf{x} \coloneqq e_1 \,\} = \textbf{force}\ \{\, \mathsf{x} \coloneqq e_2 \,\}\rangle$ in Mininix to ensure that the deep comparison gives a meaningful result.

3.3.2 The reduction rules

We define two reduction relations. The base reduction relation $\longrightarrow_{\text{base}}$ contains valid reductions under (strict) contexts. The reduction relation $\longrightarrow$ only contains one rule: CTX. This rule expresses that for any context valid E, $E[e]$ can be rewritten to $E[e']$ if e can be reduced to e' per the base reduction relation.

Binding	$b_s \coloneqq x \coloneqq_s v_s$
Strong value	$u_s, v_s \coloneqq$
Literal	$\mid$ **true** $\mid$ **false** $\mid$ **null** $\mid n \mid s$
Function	$\mid x : e \mid \{ m \} : e$
Attribute set	$\mid \{ b_s^* \}$

Figure 3.4: Strong values.

On notation:

- Reductions on terms are described by the step relation $\rightarrow$. The reflexive transitive closure of $\rightarrow$ is denoted as $\xrightarrow{*}$. The n-step reduction with the relation $\rightarrow$ is denoted as $\xrightarrow{n}$. The same conventions also apply for other reduction relations.
- Where in the grammar we write a list of semicolon-separated bindings (with no duplicate keys) as b^* or b_r^*, we write this as $\overrightarrow{b}$ or $\overrightarrow{b_r}$ in the semantics. We use $x \coloneqq e \in \overrightarrow{b}$ (or $\overrightarrow{b_r}$) and $x \coloneqq_r e \in \overrightarrow{b_r}$ to write that respectively $x \coloneqq e$ and $x \coloneqq_r e$ are in the list of bindings. We also write $\mathrm{dom}(\overrightarrow{b})$ (or $\overrightarrow{b_r}$) for the set of keys that the list of bindings $\overrightarrow{b}$ (or $\overrightarrow{b_r}$) comprises. We denote an empty list as ε.
- Where in the grammar we write a non-empty period-separated list of identifiers as x^+ or y^+, we write this as $\overrightarrow{x}$ or $\overrightarrow{y}$ in the semantics. In the semantics, these lists may also not be empty.
- For matchers m, we use $x \mathrel{?} e \in m$ to write that the matcher m optionally matches the key x with e as the default (fallback) value. We also use $x \in m$ to write that the matcher m matches the key x mandatorily, *i.e.*, x must be in the attribute set which is matched against. As for lists of bindings, we use ε to write that an empty matcher that is strict (*i.e.*, does not end with $\langle \ldots \rangle$).
- We use $\overrightarrow{b} \sim m \rightsquigarrow \overrightarrow{b_r'}$ to write that the matcher m matches bindings $\overrightarrow{b}$ of the attribute set $\{ \overrightarrow{b} \}$, giving the list of bindings $\overrightarrow{b_r'}$ that should be used in order to access the fields matched again directly, and to use the default values for fields which were not present in the provided attribute set $\{ \overrightarrow{b} \}$.

Reduction under a context. As mentioned before, the rule CTX allows for reduction using $\rightarrow_{\text{base}}$ under a weak context E. We have two types of evaluation contexts, as shown in fig. 3.7. Weak evaluation contexts E are for reductions that are not under **force**. Strict evaluation contexts E_s include all weak evaluation contexts E, but also allow for reduction under attribute set values. Note that a weak evaluation context can still very well end up using a strict evaluation context internally. For example, take the context $E \equiv \langle \textbf{force}\ \{ x \coloneqq [\cdot]; \varepsilon \} = e_2 \rangle$.

$$\textsc{ctx}\ \frac{e \longrightarrow_{\text{base}} e'}{E[e] \longrightarrow E[e']} \qquad \textsc{closed}\ \text{closed}(e) \longrightarrow_{\text{base}} e \qquad \textsc{placeholder}\ \text{placeholder}_x(e) \longrightarrow_{\text{base}} e$$

$$\textsc{force}\ \textbf{force}\ v_s \longrightarrow_{\text{base}} v_s \qquad \textsc{select}\ \frac{x \coloneqq e \in \overrightarrow{b}}{\{\ \overrightarrow{b}\ \}.x \longrightarrow_{\text{base}} e} \qquad \textsc{mselect}\ \{\ \overrightarrow{b}\ \}.x.\overrightarrow{y} \longrightarrow_{\text{base}} (\{\ \overrightarrow{b}\ \}.x).\overrightarrow{y}$$

$$\textsc{select-or}\ \{\ \overrightarrow{b}\ \}.x\ \textbf{or}\ e \longrightarrow_{\text{base}} \textbf{if}\ \{\ \overrightarrow{b}\ \}\ ?\ x\ \textbf{then}\ \{\ \overrightarrow{b}\ \}.x\ \textbf{else}\ e$$

$$\textsc{mselect-or}\ \{\ \overrightarrow{b}\ \}.x.\overrightarrow{y}\ \textbf{or}\ e \longrightarrow_{\text{base}} \textbf{if}\ \{\ \overrightarrow{b}\ \}\ ?\ x\ \textbf{then}\ (\{\ \overrightarrow{b}\ \}.x).\overrightarrow{y}\ \textbf{or}\ e\ \textbf{else}\ e$$

$$\textsc{rec}\ \textbf{rec}\ \{\ \overrightarrow{b_r}\ \} \longrightarrow_{\text{base}} \{\ \text{recsubst}(\overrightarrow{b_r})\ \} \qquad \textsc{let}\ \textbf{let}\ \overrightarrow{b_r}\ \textbf{in}\ e \longrightarrow_{\text{base}} e[\text{closeds}(\text{recsubst}(\overrightarrow{b_r}))]$$

$$\textsc{with}\ \textbf{with}\ \{\ \overrightarrow{b}\ \};\ e \longrightarrow_{\text{base}} e[\text{placeholders}(\overrightarrow{b})] \qquad \textsc{with-no-attrset}\ \frac{\neg\text{attrset}\ v_1}{\textbf{with}\ v_1;\ e_2 \longrightarrow_{\text{base}} e_2}$$

$$\textsc{apply-simple}\ (x : e_1)\ e_2 \longrightarrow_{\text{base}} e_1[x \coloneqq \text{closed}(e_2)] \qquad \textsc{apply-attrset}\ \frac{\overrightarrow{b} \sim m \rightsquigarrow \overrightarrow{b'_r}}{(\{\ m\ \} : e)\ \{\ \overrightarrow{b}\ \} \longrightarrow_{\text{base}} \textbf{let}\ \overrightarrow{b'_r}\ \textbf{in}\ e}$$

$$\textsc{apply-functor}\ \frac{\texttt{__functor} \coloneqq e_2 \in \overrightarrow{b}}{\{\ \overrightarrow{b}\ \}\ e_1 \longrightarrow_{\text{base}} e_2\ \{\ \overrightarrow{b}\ \}\ e_1} \qquad \textsc{if-true}\ \textbf{if true then}\ e_2\ \textbf{else}\ e_3 \longrightarrow_{\text{base}} e_3$$

$$\textsc{if-false}\ \textbf{if false then}\ e_2\ \textbf{else}\ e_3 \longrightarrow_{\text{base}} e_2 \qquad \textsc{assert}\ \textbf{assert true};\ e_2 \longrightarrow_{\text{base}} e_2 \qquad \textsc{op}\ \frac{u_1\ [\![\circledcirc]\!]\ u_2 \multimap v}{u_1 \circledcirc u_2 \longrightarrow_{\text{base}} v}$$

$$\textsc{op-has-attr-true}\ \frac{x \in \text{dom}(\overrightarrow{b})}{\{\ \overrightarrow{b}\ \}\ ?\ x \longrightarrow_{\text{base}} \textbf{true}} \qquad \textsc{op-has-attr-false}\ \frac{x \notin \text{dom}(\overrightarrow{b})}{\{\ \overrightarrow{b}\ \}\ ?\ x \longrightarrow_{\text{base}} \textbf{false}} \qquad \textsc{op-has-attr-no-attrset}\ \frac{\neg\text{attrset}\ v}{v\ ?\ x \longrightarrow_{\text{base}} \textbf{false}}$$

Figure 3.5: Reduction rules.

Auxiliary predicates:

$$\frac{}{\text{attrset } \{\ \overrightarrow{b}\ \}}\ \textsc{attrset}$$

Auxiliary functions:

$$\begin{aligned}
\text{recs}(\overrightarrow{b_r}) &= \{\, x \coloneqq_r e \mid x \coloneqq_r e \in \overrightarrow{b_r} \,\} \\
\text{nonrecs}(\overrightarrow{b_r}) &= \{\, x \coloneqq e \mid x \coloneqq e \in \overrightarrow{b_r} \,\} \\
\text{indirect}(\overrightarrow{b_r}) &= \{\, x \coloneqq \textbf{rec}\ \{\, \overrightarrow{b_r} \,\}.x \mid x \in \text{dom}(\overrightarrow{b_r}) \,\} \\
\text{closeds}(\overrightarrow{b}) &= \{\, x \coloneqq \text{closed}(e) \mid x \coloneqq e \in \overrightarrow{b} \,\} \\
\text{placeholders}(\overrightarrow{b}) &= \{\, x \coloneqq \text{placeholder}_x(e) \mid x \coloneqq e \in \overrightarrow{b} \,\} \\
\text{recsubst}(\overrightarrow{b_r}) &= \text{recs}(\overrightarrow{b_r})[\text{closeds}(\text{indirect}(\overrightarrow{b_r}))];\ \text{nonrecs}(\overrightarrow{b_r})
\end{aligned}$$

Figure 3.6: Reduction rules: auxiliaries.

$E \Coloneqq [\cdot]$	hole
$\mid E.\overrightarrow{x}$	selection: lhs
$\mid E.\overrightarrow{x}$ **or** e_2	selection with default: lhs
$\mid$ **with** $E;\ e_2$	inclusion: lhs
$\mid E\ e_2$	application: lhs
$\mid v_1\ E$	application with lhs value: rhs
$\mid (\{\, m \,\} : e_1)\ E$	pattern-matching fn: argument
$\mid$ **if** E **then** e_2 **else** e_3	if: condition
$\mid$ **assert** $E;\ e_2$	assert: condition
$\mid E \circledcirc e_2$	binop: lhs
$\mid v_1 \circledcirc E$	binop with lhs value: rhs
$\mid E\ ?\ x$	has attribute: lhs
$\mid$ **force** E_s	force: argument
$E_s \Coloneqq E$	weak
$\mid \{\, x \coloneqq E_s;\ \overrightarrow{b} \,\}$	attribute value

Figure 3.7: Evaluation contexts.

The different contexts that can be created primarily facilitate normal order reduction, *i.e.*, a reduction order in which the leftmost outermost redex is reduced first [22, Sec. 2.3.1]. For example, take the two evaluation context rules for application: $E\ e_2$ and $v_1\ E$. The left-hand side of the application must first be a value (normal form) before the right-hand side may be rewritten (via CTX).

Such a constraint, however, does not exist for attribute sets under strict evaluation. Here, we can select any attribute x for reduction at random. Nevertheless, we do not risk being able to create infinite derivation trees where this would not be possible with deterministic semantics: all values can only be reduced to normal form and no further; if some attribute value does generate an infinite derivation tree, then this would also have happened under a deterministic semantics.

We would also like to note that no evaluation context ever introduces any bindings for the expression in the hole, as the constructs used simply do not introduce bindings in the places where other contexts can be inserted. (For example, the `with` construct only provides new bindings for its second argument, but we can only insert contexts in the first argument.)

Placeholders and terms marked as closed. When we want to reduce some term $E[\mathsf{closed}(e)]$ or $E[\mathsf{placeholder}_x(e)]$, we know that there are no constructs used in E that could introduce any new bindings for e (see fig. 3.7). That means that there will be no substitutions performed on $\mathsf{closed}(e)$ and $\mathsf{placeholder}_x(e)$. For that reason, we can simply strip these 'wrappers': both $\mathsf{closed}(e)$ and $\mathsf{placeholder}_x(e)$ can simply be reduced to e. This is exactly what the CLOSED and PLACEHOLDER rules do.

Force. For the results of strict evaluation to be usable, we must be able to escape **force**. For this, we have the FORCE rule. When the argument of **force** has been reduced to a strong value (as described in fig. 3.4), no further reductions under (strict) contexts will be possible anymore, so **force** returns its argument verbatim.

Selections. As can be seen in fig. 3.5, there are six different rules. The reason for this is that there are different types of selections, which require different rules:

- Selections which have no default value, *i.e.*, which may fail: $e.\overrightarrow{x}$. The relevant rules are MSELECT and SELECT.

  With the appropriate context, the CTX rule handles reductions on the left-hand side of the selections. We can only select values for attribute sets, so the expression on the left-hand side of the selection must first be fully reduced to an attribute set before we can proceed. Furthermore, we need the attribute that we will select to not have any referenced to other attributes in the same attribute set, so we require the attribute set to be a value and therefore non-recursive.

The MSELECT rule handles selection paths which contain more than one element. It turns a selection $\{\ \overrightarrow{b}\ \}.x.\overrightarrow{y}$ into a selection $(\{\ \overrightarrow{b}\ \}.x).\overrightarrow{y}$. The first part of this selection, itself a selection, can then be reduced using, *e.g.*, the SELECT rule.

The SELECT rule handles selection paths consisting of only a single part. In this rule, we require that the attribute which is being selected is part of the list of bindings $\overrightarrow{b}$ of the attribute set $\{\ \overrightarrow{b}\ \}$.

- Selections which have a default value, *i.e.*, which will always succeed: $e_1.\overrightarrow{x}$ **or** e_2. The relevant rules are SELECT-OR and MSELECT-OR.

  To reduce the amount of rules, we have made the SELECT-OR and MSELECT-OR rules reduce the term to if statements. These handle failing selections by using the default value as a fallback value. The default value is passed along when parts of the selection have succeeded (but not the full selection has been completed yet).

Recursive attribute sets and let. The REC and LET rules are very simple. Both make use of the *recsubst* function, which translates a set of bindings potentially containing recursive bindings into a set of non-recursive bindings. Non-recursive bindings are left untouched, while all bindings within the set substituted for in the recursive bindings by means of indirection (see the *indirect* function).

In the REC rule, we make use of the auxiliary construct $\mathsf{closed}(e)$. For more information on why this is necessary, we refer to the section 3.3.4.

Inclusion. Inclusion, as performed by the WITH rule, is conceptually very simple. All bindings in the provided attribute set $\{\ \overrightarrow{b}\ \}$ are made available in the subterm e. However, we do make use of the auxiliary construct $\mathsf{placeholder}_x(e)$ here via $\mathsf{placeholders}(\overrightarrow{b})$. We explain why this is necessary in section 3.3.4.

In the case that the left-hand side of the **with** statement ends up reducing to a value that is not an attribute set, we apply the WITH-NO-ATTRSET rule instead.

Application. We have three rules for application.

APPLY-SIMPLE is for simple lambda abstractions. Other than marking the argument as closed before substitutions, it performs trivial β-reduction.

APPLY-ATTRSET Besides simple lambda abstractions, Mininix has functions which can pattern-match against their argument. There is one condition: the argument must be an attribute set. Based on the matching relation $\overrightarrow{b} \sim m \rightsquigarrow \overrightarrow{b'_r}$, the bindings in the result of the match $\overrightarrow{b'_r}$ are brought into scope for the function body. See section 3.3.3 for a more detailed explanation of the matching relation.

APPLY-FUNCTOR is a rule covering a newer feature. See section 2.4 for a discussion of functors and how they work. This rule is quite simple: whenever there is a juxtaposition with an attribute set on the left-hand side and any expression on the right-hand side, we check if the '`__functor`' attribute is in the attribute set on the left-hand side. If yes, the juxtaposition reduces to a double function application:

- first, the value of the '`__functor`' attribute (which should be a function or an attribute set containing the '`__functor`' attribute) applied with the attribute set on the left-hand side of the juxtaposition;
- the result of the previous application applied with the expression on the right-hand side of the juxtaposition.

Conditionals and assertions. Assertions (ASSERT) and conditionals (IF-TRUE, IF-FALSE) are very simple. For assertions, reduction becomes stuck when the condition does not reduce to true. For if statements, reduction also becomes stuck when the condition does not reduce to a Boolean value.

Binary operations. For binary operations, the left-hand side and right-hand side are reduced under the CTX rule. When both sides are reduced to values, the OP rule can be applied. The actual application of the operation is done by the reduction relation for binary operations. This relation can be seen in fig. 3.8. It is trivial that this relation is deterministic: $u_1 \llbracket \circledcirc \rrbracket u_2 \multimap v_1$ and $u_1 \llbracket \circledcirc \rrbracket u_2 \multimap v_2$ implies that $v_1 = v_2$.

Most rules here should be relatively straightforward, but there are a few where it is good to explicitly indicate what we mean.

BINOP-ADD-STR. We use the binary operator $+\!\!+$ to denote string concatenation.

BINOP-LT-STR. With $s_1 <_{lex} s_2$, we mean that s_1 is lexicographically smaller than s_2. Specificially, we mean the following. If s_1 is shorter than s_2, then s_1 is also smaller than s_2. If s_1 is longer than s_2, then s_1 is not smaller than s_2. If s_1 and s_2 are equally long, we perform a character-by-character (lexicographic) comparison. This is done on the basis of the respective ASCII codes.

BINOP-EQ. The definition of *expreq* should be applied from top-to-bottom; the last case $\text{expreq}(e_1, e_2) = \text{false}$ should only be used when none of the other cases can be used.

BINOP-UPD-ATTRSET. The union of the two sets of bindings, written $\overrightarrow{b_2} \cup \overrightarrow{b_1}$, is left-biased. So if we have $k := d \in \overrightarrow{b_1}$ and $k := e \in \overrightarrow{b_2}$, then we will have $k := e \in \overrightarrow{b_2} \cup \overrightarrow{b_1}$.

$$\textsc{binop-add-int}\quad n_1 \llbracket + \rrbracket n_2 \multimap n_1 + n_2$$

$$\textsc{binop-add-str}\quad s_1 \llbracket + \rrbracket s_2 \multimap s_1 +\!\!+ s_2$$

$$\textsc{binop-min-int}\quad n_1 \llbracket - \rrbracket n_2 \multimap n_1 + n_2$$

$$\textsc{binop-div-int}\quad \frac{n_2 \neq 0}{n_1 \llbracket / \rrbracket n_2 \multimap n_1 + n_2}$$

$$\textsc{binop-lt-int}\quad n_1 \llbracket < \rrbracket n_2 \multimap n_1 < n_2$$

$$\textsc{binop-lt-str}\quad s_1 \llbracket < \rrbracket s_2 \multimap s_1 <_{lex} s_2$$

$$\textsc{binop-eq}\quad v_1 \llbracket = \rrbracket v_2 \multimap \mathrm{expreq}(v_1, v_2)$$

$$\textsc{binop-upd-attrset}\quad \{\, \overrightarrow{b_1} \,\} \llbracket // \rrbracket \{\, \overrightarrow{b_2} \,\} \multimap \overrightarrow{b_2} \cup \overrightarrow{b_1}$$

Auxiliary functions:

$$\begin{aligned}
\mathrm{expreq}(\mathbf{null}, \mathbf{null}) &= \mathrm{true} \\
\mathrm{expreq}(\mathbf{true}, \mathbf{true}) &= \mathrm{true} \\
\mathrm{expreq}(\mathbf{false}, \mathbf{false}) &= \mathrm{true} \\
\mathrm{expreq}(n_1, n_2) &= \mathrm{true} \quad \text{if } n_1 = n_2 \\
\mathrm{expreq}(s_1, s_2) &= \mathrm{true} \quad \text{if } s_1 = s_2 \\
\mathrm{expreq}(\{\, \overrightarrow{b_1} \,\}, \{\, \overrightarrow{b_2} \,\}) &= \mathrm{true} \quad \text{if } \mathrm{dom}(\overrightarrow{b_1}) = \mathrm{dom}(\overrightarrow{b_2}) \text{ and } \forall k \in \mathrm{dom}(\overrightarrow{b_1}). \\
&\qquad\qquad \mathrm{expreq}(\overrightarrow{b_1}(k), \overrightarrow{b_2}(k)) = \mathrm{true} \\
\mathrm{expreq}(e_1, e_2) &= \mathrm{false}
\end{aligned}$$

Figure 3.8: Binary operation reduction.

MATCH-EMPTY

$$\varepsilon \sim \varepsilon \rightsquigarrow \varepsilon$$

MATCH-ANY

$$\overrightarrow{b} \sim \ldots \rightsquigarrow \varepsilon$$

MATCH-MANDATORY

$$\frac{x \notin \mathrm{dom}(\overrightarrow{b}) \qquad x \notin \mathrm{dom}(m) \qquad \overrightarrow{b} \sim m \rightsquigarrow \overrightarrow{b'_r}}{x := e;\ \overrightarrow{b}\ \sim\ x,\ m\ \rightsquigarrow\ x := e;\ \overrightarrow{b'_r}}$$

MATCH-OPT-AVAIL

$$\frac{x \notin \mathrm{dom}(\overrightarrow{b}) \qquad x \notin \mathrm{dom}(m) \qquad \overrightarrow{b} \sim m \rightsquigarrow \overrightarrow{b'_r}}{x := d;\ \overrightarrow{b}\ \sim\ x\ ?\ e, m\ \rightsquigarrow\ x := d;\ \overrightarrow{b'_r}}$$

MATCH-OPT-DEFAULT

$$\frac{x \notin \mathrm{dom}(\overrightarrow{b}) \qquad x \notin \mathrm{dom}(m) \qquad \overrightarrow{b} \sim m \rightsquigarrow \overrightarrow{b'_r}}{\overrightarrow{b}\ \sim\ x\ ?\ e, m\ \rightsquigarrow\ x :=_r e;\ \overrightarrow{b'_r}}$$

Figure 3.9: Matching rules.

Attribute set membership. The attribute membership check operator ⟨?⟩ is not among the normal operators because its right-hand side may not be reduced: it must be an identifier. As such, we also have separate top-level rules for this operator: OP-HAS-ATTR-TRUE, OP-HAS-ATTR-FALSE and OP-HAS-ATTR-NO-ATTRSET. The first two of these rules are trivial. The last rule, OP-HAS-ATTR-NO-ATTRSET, is also there, as it is not an error to check for the existence of an attribute in a value that is not an attribute set; checking if the integer zero has the attribute "foo" should simply give **false**.

3.3.3 Pattern matching

We already discussed pattern-matching functions in section 2.2. Now, we will discuss the relation that turns these patterns and a passed attribute set into a list of bindings.

In fig. 3.9, the matching relation $\overrightarrow{b} \sim m \rightsquigarrow \overrightarrow{b'_r}$ is shown. We say that the bindings $\overrightarrow{b}$ of the passed attribute set are matched by the pattern m, yielding the list of potentially recursive bindings $\overrightarrow{b'_r}$. Substituting for these bindings will make the matched attributes (or their default values) available.

Variables matched against with a default value are handled by the MATCH-OPT-AVAIL and MATCH-OPT-DEFAULT rules. The former can be used when the variable matched against is in the bindings of the passed attribute set; the latter is used when this variable is missing, and instead adds a binding with the default value to the list of resulting bindings.

The MATCH-MANDATORY rule handles attributes mandatorily matched against.

If that attribute is indeed member of the passed attribute set, then its binding is copied to the list of resulting bindings.

The MATCH-EMPTY rule handles an empty strict matcher and empty passed attribute set, naturally yielding an empty list of bindings.

We also support patterns which are not strict (*i.e.*, that end with $\langle \ldots \rangle$). The MATCH-ANY rule matches an empty matcher with an arbitrary list of bindings $\overrightarrow{b}$ from the passed attribute set, yielding an empty list of bindings.

3.3.4 Parallel substitution

The parallel substitution function is shown in fig. 3.10. Most cases are trivial, but we still list them for the sake of completeness.

We can see that the substitution function prevents substitutions of variables that are guaranteed to be bound in the bodies of let statements and functions. However, it is not decidable what variables will be made available by the inclusion (**with**) construct, so we simply substitute in the body of this construct as well. To overcome the issues this causes, we make use of placeholders (in the WITH rule), which we will explain now.

Placeholders. Mininix includes so-called placeholders, which the Nix language does not have. A placeholder, denoted $\mathsf{placeholder}_x(e)$, can be seen as an identifier x which has already been substituted for, but which can be substituted for again. When not substituted for, a placeholder reduces to its associated expression e.

Placeholders, as a run-time construct, are necessary when evaluating Mininix programs. In particular, they are required for the correct evaluation of the inclusion (`with`) construct. In section 2.3, we already discussed the way the semantics of this construct had changed.

Let us take the example from section 2.3 once again to illustrate the use of placeholders:

$$\textbf{with}\ \{\ \mathsf{x} \coloneqq 1\ \};\ \textbf{with}\ \{\ \mathsf{x} \coloneqq 2\ \};\ \mathsf{x}$$

Now, let us rewrite this program until we arrive at a normal form. But instead of using the WITH rule, we define a new rule, WITH-BAD, which does not make use placeholders.

$$\begin{array}{l}\text{WITH-BAD}\\ \textbf{with}\ \{\ \overrightarrow{b}\ \};\ e \longrightarrow_{\text{base}} e[\,\overrightarrow{b}\,]\end{array}$$

When using this rule, we get the following reduction (leaving out contexts for

$$
\begin{aligned}
\mathbf{null}[\vec{b}\,] &= \mathbf{null}\\
\mathbf{true}[\vec{b}\,] &= \mathbf{true}\\
\mathbf{false}[\vec{b}\,] &= \mathbf{false}\\
s[\vec{b}\,] &= s\\
n[\vec{b}\,] &= n\\
x[\vec{b}\,] &= \begin{cases} e & \text{if } x \coloneqq e \in \vec{b} \text{ for some } e\\ x & \text{otherwise}\end{cases}\\
\{\,\vec{b'}\,\}[\vec{b}\,] &= \left\{ \left\{ x \coloneqq e[\vec{b}\,] \;\middle|\; x \coloneqq e \in \vec{b'} \right\} \right\}\\
\mathbf{rec}\ \{\,\vec{b_r}\,\}[\vec{b}\,] &= \left\{ \begin{array}{l} \left\{ x \coloneqq e[\vec{b}\,] \;\middle|\; x \coloneqq e \in \vec{b_r} \right\} \cup \\ \left\{ x \coloneqq_r e[\vec{b} \setminus \mathrm{dom}(\vec{b_r})] \;\middle|\; x \coloneqq_r e \in \vec{b_r} \right\} \end{array} \right\}\\
(\mathbf{let}\ \vec{b_r}\ \mathbf{in}\ e)[\vec{b}\,] &= \mathbf{let}\ \left\{ x \coloneqq d[\vec{b}\,] \;\middle|\; x \coloneqq d \in \vec{b_r} \right\} \cup\\
&\qquad\ \left\{ x \coloneqq_r d[\vec{b} \setminus \mathrm{dom}(\vec{b_r})] \;\middle|\; x \coloneqq_r d \in \vec{b_r} \right\}\\
&\quad\ \mathbf{in}\ e\\
(e.\vec{x})[\vec{b}\,] &= e[\vec{b}\,].\vec{x}\\
(e_1.\vec{x}\ \mathbf{or}\ e_2)[\vec{b}\,] &= e_1[\vec{b}\,].\vec{x}\ \mathbf{or}\ e_2[\vec{b}\,]\\
(x : e)[\vec{b}\,] &= x : e[\vec{b} \setminus \{x\}]\\
(\{m\} : e)[\vec{b}\,] &= \left\{ \begin{array}{l} \{\, x \mid x \in m \,\} \cup \\ \left\{ x \mathbin{?} e[\vec{b} \setminus \mathrm{dom}(m)] \;\middle|\; x \mathbin{?} e \in m \right\} \end{array} \right\} :\\
&\quad\ e[\vec{b} \setminus \mathrm{dom}(m)]\\
(e_1\ e_2)[\vec{b}\,] &= e_1[\vec{b}\,]\ \ e_2[\vec{b}\,]\\
(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)[\vec{b}\,] &= \mathbf{if}\ e_1[\vec{b}\,]\ \mathbf{then}\ e_2[\vec{b}\,]\ \mathbf{else}\ e_3[\vec{b}\,]\\
(\mathbf{with}\ e_1;\ e_2)[\vec{b}\,] &= \mathbf{with}\ e_1[\vec{b}\,];\ e_2[\vec{b}\,]\\
(\mathbf{assert}\ e_1;\ e_2)[\vec{b}\,] &= \mathbf{assert}\ e_1[\vec{b}\,];\ e_2[\vec{b}\,]\\
(e_1 \circledcirc e_2)[\vec{b}\,] &= e_1[\vec{b}\,]\ \circledcirc\ e_2[\vec{b}\,]\\
(e \mathbin{?} x)[\vec{b}\,] &= e[\vec{b}\,]\ \mathbin{?}\ x\\
(\mathbf{force}\ e)[\vec{b}\,] &= \mathbf{force}\ e[\vec{b}\,]\\
\mathrm{closed}(e)[\vec{b}\,] &= \mathrm{closed}(e)\\
(\mathsf{placeholder}_x(d))[\vec{b}\,] &= \begin{cases} e & \text{if } x \coloneqq e \in \vec{b} \text{ for some } e\\ \mathsf{placeholder}_x(d) & \text{otherwise}\end{cases}
\end{aligned}
$$

Figure 3.10: Parallel substitution.

brevity):

$$\begin{aligned}
&\textbf{with}\ \{\ \mathsf{x} \coloneqq 1\ \};\ \textbf{with}\ \{\ \mathsf{x} \coloneqq 2\ \};\ \mathsf{x} \\
&\quad \to \textbf{with}\ \{\ \mathsf{x} \coloneqq 2\ \};\ \mathsf{x}[\mathsf{x} \coloneqq 1] && \text{by WITH-BAD} \\
&\quad \to \mathsf{x}[\mathsf{x} \coloneqq 1][\mathsf{x} \coloneqq 2] && \text{by WITH-BAD} \\
&\quad = 1[\mathsf{x} \coloneqq 2] && \text{by definition of } e[\vec{b}] \\
&\quad = 2 && \text{by definition of } e[\vec{b}]
\end{aligned}$$

As explained in section 2.3, this is not the result that we want to see. Now instead, let us use the actual Mininix rules.

$$\begin{aligned}
&\textbf{with}\ \{\ \mathsf{x} \coloneqq 1\ \};\ \textbf{with}\ \{\ \mathsf{x} \coloneqq 2\ \};\ \mathsf{x} \\
&\quad \to \textbf{with}\ \{\ \mathsf{x} \coloneqq 2\ \};\ \mathsf{x}[\mathsf{x} \coloneqq \mathsf{placeholder}_{\mathsf{x}}(1)] && \text{by WITH} \\
&\quad \to \mathsf{x}[\mathsf{x} \coloneqq \mathsf{placeholder}_{\mathsf{x}}(1)][\mathsf{x} \coloneqq \mathsf{placeholder}_{\mathsf{x}}(2)] && \text{by WITH} \\
&\quad = \mathsf{placeholder}_{\mathsf{x}}(1)[\mathsf{x} \coloneqq \mathsf{placeholder}_{\mathsf{x}}(2)] && \text{by definition of } e[\vec{b}] \\
&\quad = \mathsf{placeholder}_{\mathsf{x}}(2) && \text{by definition of } e[\vec{b}] \\
&\quad \to 2 && \text{by PLACEHOLDER}
\end{aligned}$$

One might wonder why we cannot simply adjust the substitution function for inclusions, simply not substituting for the variables which are in the attribute set on the left-hand side, just like let-in statements. But this approach is not feasible: unlike let-in statements, the left-hand side of inclusions is an expression. We can not always statically determine *if* this expression will reduce to an attribute set, let alone what attributes it will have.

Terms marked as closed. Take the following Mininix program:

$$(\mathsf{x} : \mathsf{y} : \mathsf{x})\ (\textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ \mathsf{y})$$

Now, let us rewrite this program until we arrive at a normal form. But instead of using the APPLY-SIMPLE rule, we define a new rule, APPLY-SIMPLE-BAD, which does not make use of ‘terms marked as closed’. We also use the WITH-BAD rule from section 3.3.4.

APPLY-SIMPLE-BAD

$$(x : e_1)\ e_2 \to_{\text{base}} e_1[x \coloneqq e_2]$$

Then, we would have the following reduction:

$$
\begin{aligned}
&(\mathsf{x} : \mathsf{y} : \mathsf{x})\ (\textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ \mathsf{y})\ 1 \\
&\quad \longrightarrow (\mathsf{y} : \mathsf{x}[\mathsf{x} \coloneqq \textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ \mathsf{y}])\ 1 && \text{by APPLY-SIMPLE-BAD} \\
&\quad = (\mathsf{y} : \textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ \mathsf{y})\ 1 && \text{by definition of } e[\,\overrightarrow{b}\,] \\
&\quad \longrightarrow \textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ \mathsf{y}[\mathsf{y} \coloneqq 1] && \text{by APPLY-SIMPLE-BAD} \\
&\quad = \textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ 1 && \text{by definition of } e[\,\overrightarrow{b}\,] \\
&\quad \longrightarrow 1[\mathsf{y} \coloneqq 2] && \text{by WITH-BAD} \\
&\quad = 1 && \text{by definition of } e[\,\overrightarrow{b}\,]
\end{aligned}
$$

But this is not the desired result! The argument of a lambda should not be able to capture any arguments bound by the lambda (in this case: y). Indeed, the official Nix interpreter gives the result '2' for this program.

We avoid capture by *marking terms as closed*. Simply put, when substituting the argument for a function or substituting let bindings, we mark all substituted terms as closed. Whenever another substitution is applied that reaches this term, the substitution function will not recurse into this marked term. Instead, the term will remain unchanged, and capture will not occur.

Let us look at the example from just then once again, now with Mininix and placeholders (leaving out the contexts for brevity):

$$
\begin{aligned}
&(\mathsf{x} : \mathsf{y} : \mathsf{x})\ (\textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ \mathsf{y})\ 1 \\
&\quad \longrightarrow (\mathsf{y} : \mathsf{x}[\mathsf{closeds}(\mathsf{x} \coloneqq \textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ \mathsf{y})])\ 1 && \text{by APPLY-SIMPLE} \\
&\quad \longrightarrow (\mathsf{y} : \mathsf{x}[\mathsf{x} \coloneqq \mathsf{closed}(\textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ \mathsf{y})])\ 1 && \text{by definition of } \textit{closeds} \\
&\quad = (\mathsf{y} : \mathsf{closed}(\textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ \mathsf{y}))\ 1 && \text{by definition of } e[\,\overrightarrow{b}\,] \\
&\quad \longrightarrow \mathsf{closed}(\textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ \mathsf{y})[\mathsf{y} \coloneqq 1] && \text{by APPLY-SIMPLE} \\
&\quad = \mathsf{closed}(\textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ \mathsf{y}) && \text{by definition of } e[\,\overrightarrow{b}\,] \\
&\quad \longrightarrow \textbf{with}\ \{\ \mathsf{y} \coloneqq 2\ \};\ \mathsf{y} && \text{by CLOSED} \\
&\quad \longrightarrow \mathsf{y}[\mathsf{y} \coloneqq \mathsf{placeholder}_{\mathsf{y}}(2)] && \text{by WITH} \\
&\quad = \mathsf{placeholder}_{\mathsf{y}}(2) && \text{by definition of } e[\,\overrightarrow{b}\,] \\
&\quad \longrightarrow 2 && \text{by PLACEHOLDER}
\end{aligned}
$$

As we can see, this reduction does yield the right result.

Further considerations. Of course, both placeholders and terms marked as closed are very ad-hoc ways of implementing capture avoidance, and more well-known ways of doing this such as De Bruijn indices [9] might be more appropriate. We discuss the tradeoffs in chapter 5.

Chapter 4

Building a verified interpreter for Mininix

In this chapter, we cover the process of building and verifying an interpreter for Mininix. We do so using the proof assistant Coq [23].

In section 4.3, we discuss how the interpreter was written. As preparation for verification of the interpreter, we discuss mechanization of the semantics described from section 3.3 in section 4.2. We discuss the actual verification of the interpreter in section 4.4, particularly touching on soundness and completeness in section 4.4.4 and section 4.4.5, respectively.

4.1 Mechanizing shared components

As the interpreter we give is very simple, it shares quite some functionality with the semantic rules. For example, the substitution function is shared between the two. The syntax for expressions and values and the way strong values are defined are also shared. In this section, we do not discuss how we translate the substitution function to Coq, because this is relatively trivial.

4.1.1 Representing expressions

We define expressions as an inductive type `expr` with different constructors for every syntactic element. The definition of this type can be seen in listing 7. Most constructs are very simple to represent, so we will not explicitly cover them. A few things stand out.

Attribute sets, let bindings, pattern-matching functions. There are many ways to represent attribute sets; which one is best depends on the application. In Mininix, the name of an attribute is always a simple string—there are no dynamic attributes that first have to be reduced from an expression to a string. So naturally, a partial map is a good fit here. More specifically, we use the new generic finite

```
Inductive expr : Type :=
  | X_V (v : value)                                                          (* v *)
  | X_Id (x : string)                                                        (* x *)
  | X_Attrset (bs : gmap string b_rhs)                                 (* { b_r* } *)
  | X_LetBinding (bs : gmap string b_rhs) (e : expr)            (* let b_r* in e *)
  | X_Select (e : expr) (xs : nonempty string)                          (* e.xs *)
  | X_SelectOr (e : expr) (xs : nonempty string) (or : expr)  (* e.xs or e *)
  | X_Apply (e1 e2 : expr)                                               (* e1 e2 *)
  | X_Cond (e1 e2 e3 : expr)                              (* if e1 then e2 else e3 *)
  | X_Incl (e1 e2 : expr)                                           (* with e1; e2 *)
  | X_Assert (e1 e2 : expr)                                       (* assert e1; e2 *)
  | X_Op (op : op) (e1 e2 : expr)                                   (* e1 <op> e2 *)
  | X_HasAttr (e1 : expr) (x : string)                                  (* e ? x *)
  | X_Force (e : expr)                                                 (* force e *)
  | X_Closed (e : expr)                                              (* closed(e) *)
  | X_Placeholder (x : string) (e : expr)                     (* placeholder_x(e) *)
with b_rhs :=
  | B_Rec (e : expr)      (* :=  e *)
  | B_Nonrec (e : expr)   (* :=_r e *)
with matcher :=
  | M_Matcher (ms : gmap string m_rhs) (strict : bool)
with m_rhs :=
  | M_Mandatory
  | M_Optional (e : expr)  (* ? e *)
with value :=
  | V_Bool (p : bool) : value                    (* true | false *)
  | V_Null : value                                       (* null *)
  | V_Int (n : Z) : value                                   (* n *)
  | V_Str (s : string) : value                              (* s *)
  | V_Fn (x : string) (e : expr) : value                 (* x: e *)
  | V_AttrsetFn (m : matcher) (e : expr) : value    (* { m }: e *)
  | V_Attrset (bs : gmap string expr) : value.       (* { b* } *)
```

Listing 7: Expressions, formalized in Coq. Snippet from `expr.v`.

map facility from Coq-std++ [24], called 'gmap'. Based on the work by Appel and Leroy [5], this structure has a few very desirable properties, as summarized by Krebbers [18]; the following are relevant for us:

- it satisfies the extensionality property $m_1 = m_2 \leftrightarrow \forall k.\ m_1(k) = m_2(k)$,
- it works well in nested recursive definitions: nested induction is possible.

Attribute selection paths. Remember that the right-hand side of selections, the selection path, may not be empty. To facilitate this requirement, we define our own `nonempty` type, which defines a generic list that contains at least one element. It is represented as an algebraic datatype with a single constructor, which is exactly like the 'cons' constructor for lists: the first argument is the head and the second argument is the tail.

Binary operations. We also define the `op` type, which is a variant of all different operators that can be represented. Strings, integers and Booleans are represented with types provided by the standard library of Coq. The string type provided by Coq can represent ASCII strings.

4.1.2 Strong values

Strong values, as shown in fig. 3.4, are shared between the semantics and the interpreter. They are defined as a separate inductive type and are very close to the `value` type; the only difference is that the attribute set constructor now takes a `gmap string strong_value` instead of a `gmap string expr`. To facilitate the conversion from strong values to values, we have an injective function `value_from_strong_value`. We also define `expr_from_strong_value` as ⟨`X_V ∘ value_from_strong_value`⟩.

Because we later perform induction for a recursive function on strong values, we need to define a custom induction principle. This must allow us to prove that $P(v_s)$ holds for an entire strong value that is an attribute set $\{\ \overrightarrow{b_s}\ \}$ if $P(v_s)$ holds for every attribute of the set. Our induction principle (`strong_value_ind'`) is based on a test for finite maps in Coq-std++ [24], as we have to define this induction principle in such a way that it works with the `gmap` data structure.

4.1.3 Coercions

We define six coercions which makes working with our expression representation more convenient. Namely: from `string` to `expr` via `X_Id`, from `value` to `expr` via `X_V`, from `Z` (integers) to `value` via `V_Int`, from `bool` to `value` via `V_Bool` and from `strong_value` to `value` via `value_from_strong_value`. For example, this allows us to use `true` in a context where an `expr` is expected. This is then the same as writing `X_V (V_Bool true)`.

4.2 Mechanizing the semantics

For the relations $\rightarrow_{\text{base}}$ and $\rightarrow$, we define two inductive types: `base_step` and `step`, respectively. These have the type `expr → expr → Prop`. For every reduction rule, we define a constructor in the type for the respective relation.

Note that in the `E_ApplyAttrset` rule, we make use of the matching relation `bs ~ m ~> bs'`, just like in the corresponding reduction rule APPLY-ATTRSET.

4.2.1 Evaluation contexts

We define evaluation contexts in a functional way, very similar to the way that Jacobs [17] suggests doing this. See listing 9 to see the structure of our code for evaluation contexts.

```
Inductive base_step : expr → expr → Prop :=
  | E_Force (sv : strong_value) :
      X_Force sv -->base sv
  (* ... *)
  | E_ApplyAttrset m e bs bs' :
      bs ~ m ~> bs' →
      X_Apply (V_AttrsetFn m e) (V_Attrset bs) -->base
      X_LetBinding bs' e
  (* ... *)
  | E_Op op v1 v2 u :
      v1 ⟦op⟧ v2 -⊚-> u →
      X_Op op v1 v2 -->base u
  (* ... *)
  | E_Assert e2 :
      X_Assert true e2 -->base e2
where "e -->base e'" := (base_step e e').

(* ... *)

Variant step : expr → expr → Prop :=
  E_Ctx e1 e2 E uf_int :
    is_ctx false uf_int E →
    e1 -->base e2 →
    E e1 --> E e2
where "e --> e'" := (step e e').
```

Listing 8: The $\longrightarrow_{\text{base}}$ and $\longrightarrow$ relations in Coq.

```
Variant is_ctx_item : bool → bool → (expr → expr) → Prop :=
  | IsCtxItem_Select uf_ext xs :
      is_ctx_item uf_ext false (λ e1, X_Select e1 xs)
  (* ... *)
  | IsCtxItem_ApplyAttrsetR uf_ext m e1 :
      is_ctx_item uf_ext false (λ e2, X_Apply (V_AttrsetFn m e1) e2)
  (* ... *)
  | IsCtxItem_Force uf_ext :
      is_ctx_item uf_ext true (λ e, X_Force e)
  | IsCtxItem_ForceAttrset bs x :
      is_ctx_item true true (λ e, X_V (V_Attrset (<[x := e]>bs))).

(* ... *)

Inductive is_ctx : bool → bool → (expr → expr) → Prop :=
  | IsCtx_Id uf : is_ctx uf uf id
  | IsCtx_Compose E1 E2 uf_int uf_aux uf_ext :
      is_ctx_item uf_ext uf_aux E1 →
      is_ctx uf_aux uf_int E2 →
      is_ctx uf_ext uf_int (E1 ∘ E2).
```

Listing 9: Predicates for evaluation contexts.

An evaluation context can be seen as a function $Expr \longrightarrow Expr$. However, not all functions in $Expr \longrightarrow Expr$ may be evaluation contexts. We define a predicate `is_ctx` to validate that a function is in the set of valid contexts.

We facilitate nested contexts by defining the predicate `is_ctx_item`, which defines a single 'level' of the context. The `is_ctx` predicate then only needs two rules: composition and the identity context.

Instead of having two separate evaluation context types, we have one that encodes both weak and strong contexts. We use Booleans to encode the relationship between weak and strong contexts. The first Boolean parameter of `is_ctx_item` specifies whether the item itself is strong, *i.e.*, whether it is part of the definition of E of E_s as in fig. 3.7. The second Boolean parameter of `is_ctx_item` specifies whether the item internally uses a weak or strong context. In both these cases, `true` stands for strong (E_s) and `false` stands for weak (E) contexts.

With this information for context items available, we can correctly compose context items. This is what `is_ctx` does.

4.3 Writing an interpreter

The most simple evaluation function signature thinkable is that of a partial map from expressions to values:

$$\text{eval} : Expr \rightharpoonup Value$$

However, as we are working with Coq, such a signature is not feasible. Firstly, functions in Coq must be total. Practically, this means (1) that our function may not be partial and (2) that we must prove that the function terminates.

(1) is easily satisfied by making the output type a variant, *i.e.*, using Coq's `option` type with the constructors `Some` e and `None`. (2) however, remains a challenge.

Because Mininix (as an extension of the λ-calculus) is Turing-complete, it is not decidable whether any given program will terminate. This means that we cannot prove that the interpreter will terminate for any arbitrary Nix program. Instead, we must resort to other measures to have a total interpreter. We can use the same approach as Amin and Rompf [4] by resorting to 'Partiality Fuel': we can parameterize our evaluation function with a 'fuel value' $n \in \mathbb{N}$, which is decreased every time the evaluation function recursively calls itself. When the fuel runs out (*i.e.*, $n = 0$), the interpreter then reports that evaluation has failed. The structural reduction of n in recursive calls allows Coq to reason about the totality of our evaluation function.

So, in Coq, we would have an evaluation function with a signature like this:

```
eval : nat → expr → option value
```

Although this signature does suffice to write an interpreter for Mininix, we still add one more parameter for convenience. We call this Boolean parameter uf: 'under

force'. When uf is true, then we know that we should recursively evaluate the values of attribute set members. This way, we reduce to strong values (see fig. 3.4). When uf is false, we instead simply reduce to 'normal' values (see fig. 3.1). Finally, in Coq, we have the following signature:

```
eval : nat → bool → expr → option value
```

For the sake of simplicity, our interpreter only returns an `option value` instead of, *e.g.*, `option (option value)` like the interpreter from Amin and Rompf [4]. Note that this means that, unlike the interpreter defined by Amin and Rompf [4], we cannot discern between the interpreter failing because of a faulty program, or because of fuel running out (non-termination).

Our interpreter is also not maximally lazy, as defined by Dolstra [10]—we have no cache which allows us to share terms and to rewrite in more than one place at the same time. We also perform no checks for simple infinite recursion using 'blackholing' like Dolstra [10]. Finally, our interpreter is substitution-based and does not use make use of environments.

Implementation of the interpreter. Because the entire interpreter is too long to fully write out here, we instead take a look at part of the interpreter in which we highlight a few cases. The snippet of the interpreter can be found in listing 10.

We define the evaluation function in two parts so that we have more control over the simplification of applications of `eval`. We define a trivial lemma (`eval_S`) that states that $\forall n.$ `eval` (`S` n) = `eval1` (`eval` n). We can then use this lemma to manually unfold applications of `eval`.

The interpreter makes use of the monadic operations defined for the `option` monad by std++ [24]. The syntax ⟨`x ← v; M`⟩ is syntactic sugar for the monadic bind operation similar to the `do` notation in Haskell, *i.e.*, ⟨`v ≫= λx, M`⟩.

The `eval1` function pattern-matches on the expression that needs to be evaluated. Let us now discuss a few specific cases that are listed.

- `d = X_Id x` ($d = x$). We are instructed to evaluate an identifier. But an identifier cannot be reduced to a value; this identifier was not substituted for. We report failure by returning `None`.
- `d = X_Force e` ($d =$ **force** e). We need to return the strict evaluation of e. We return the recursive call of the interpreter for the evaluation of e and provide uf = true, making the interpreter strictly evaluate e.
- `d = X_V (V_Attrset bs)` ($d = \{ \overrightarrow{b} \}$). What we need to do here depends on whether we are operating *under force* or not, *i.e.*. We do this by checking the argument provided for the `uf` parameter. If this is false, we can return the provided attribute set verbatim. If it is true however, we need to do a bit more work.

```
Definition eval1 (go : bool → expr → option value)
  (uf : bool) (d : expr) : option value :=
    match d with
    | X_Id _ => None
    | X_Force e => go true e
    (* ... *)
    | X_V (V_Attrset bs) =>
        if uf
        then vs' ← map_mapM (go true) bs;
             Some (V_Attrset (X_V <$> vs'))
        else Some (V_Attrset bs)
    | X_V v => Some v
    | X_Attrset bs => go uf (V_Attrset (rec_subst bs))
    (* ... *)
    | X_Apply e1 e2 =>
      v1' ← go false e1;
      match v1' with
      | V_Fn x e =>
          let e' := subst {[x := X_Closed e2]} e
          in go uf e'
      | V_AttrsetFn m e =>
          v2' ← go false e2;
          bs ← maybe V_Attrset v2';
          bs' ← matches m bs;
          go uf (X_LetBinding bs' e)
      (* ... *)
      end
    (* ... *)
    end.

Fixpoint eval (n : nat) (uf : bool) (e : expr) : option value :=
  match n with
  | O => None
  | S n => eval1 (eval n) uf e
  end.

Global Opaque eval.
```

Listing 10: Structure of the interpreter.

We recursively strictly evaluate every binding in `bs`. We do this by mapping the value v of every member to `go true` v. By using `map_mapM (go true) bs`, we get an `option (list value)` instead of the `list (option value)` we would get when using `go true <$> bs`. Practically, this means that either 'all values successfully strictly evaluated to a strong value' (we have `Some`) or 'the strict evaluation of at least one attribute value failed' (we have `None`).

`map_mapM` is an auxiliary function which we define separately. It functions very similarly to the `mapM` function for lists (as in Haskell and std++), but works for finite maps instead. Its simplified signature looks like this:

```
map_mapM : (f : A → M B) → gmap K A → M (gmap K B)
```

where `M` is a monad, `K` is the key type and `A` and `B` are the value types.

- `d = X_Assert e1 e2` (d = **assert** e_1; e_2). We need to check if `e1` evaluates to the Boolean value 'true'. So we first recursively call the interpreter to evaluate `e1` normally. Note that we do not pass along `uf`: as we need `e1` to evaluate to a Boolean, strictly evaluating it has no use.

  After evaluating `e1` (with the resulting value now in `v1'`), we check if it indeed evaluated to a Boolean value. If not, we stop return that an error occurred. Otherwise, we return the result of the recursive evaluation of `e2`. This time, we do pass along `uf`, as we do no further processing of the value returned by this recursive interpreter call.

The prelude. As the attentive reader may have noticed: the `val` function does not make the prelude available (see section 3.2). Because the interpreter calls itself recursively, this is also not desirable. However, we do have the top-level evaluation function `tl_eval`, which makes the prelude available by substitution. This is the function that should actually be called when evaluating a Mininix program.

4.4 Verifying the interpreter

We use the notation $e \overset{n}{\mapsto} v$ to write that the expression e successfully evaluates to the value v with n amount of partiality fuel. Moreover, we use $e \overset{n}{\mapsto}_s v$ to mean the same, but now under strict evaluation. The former would be written in Coq as ⟨`eval` n `false` e `= Some` v⟩. The latter as ⟨`eval` n `true` e `= Some` v⟩. Finally, we also have $e \overset{n}{\mapsto}_{uf} v$ where uf is a Boolean. When uf is true, then this means the same as $e \overset{n}{\mapsto}_s v$. Otherwise, it means the same as $e \overset{n}{\mapsto} v$.

Before we can verify the interpreter, we need to choose how we even define the interpreter 'functioning correctly'. The definition we use is as follows:

- The interpreter is complete with regard to the semantics when for every derivation tree proving the reduction from an expression to a value $e \overset{*}{\to} v'$

there exists an amount of fuel n such that the interpreter normally (non-strictly) evaluates e to v':

$$\forall e\, v'.\ e \xrightarrow{*} v' \implies \exists n.\ e \overset{n}{\mapsto} v'$$

- The interpreter is sound with regard to the semantics when there exists a derivation tree proving the reduction from an expression to a value $e \xrightarrow{*} v'$ if there exists an amount of fuel n such that the interpreter normally (non-strictly) evaluates e to v':

$$\forall n\, e\, v'.\ e \overset{n}{\mapsto} v' \implies e \xrightarrow{*} v'$$

4.4.1 Binary operations

The interpreter makes use of an auxiliary function `binop_eval` for binary operations. To prove the correctness of the interpreter, we have the following lemma.

Lemma 1 (`binop_eval_sound`, `binop_eval_complete`). `binop_eval op u1 u2 = Some v` *if and only if* `u1 ⟦op⟧ u2 -◎-> v`.

This correspondence also makes it trivial that the binary operation relation is deterministic.

4.4.2 Pattern matching

When evaluating the application of a pattern-matching function, the interpreter uses the auxiliary function `matches` to generate a list of bindings for the variables matched against. In the semantics, we use the matching relation $\overrightarrow{b} \sim m \rightsquigarrow \overrightarrow{b'_r}$ for this task. To be able to prove the soundness and completeness of the interpreter, we must first prove that the `matches` function is correct with respect to the matching relation.

Lemma 2 (`matches_correct`). `matches m bs = Some brs'` *if and only if* `bs ~> m ~> brs'`.

This correspondence also makes it trivial that the matching relation is deterministic, as `matches` is a function.

4.4.3 `map_mapM` and `map_Forall2`

The `gmap` plays a prominent role within our formalized semantics. It is used to represent collections of bindings for (recursive) attribute sets and the let construct, and is used to represent patterns for pattern-matching functions.

However, the way that the `gmap` is used differs between the semantics and the interpreter. The semantics prescribe no fixed order when reducing attribute set values under force, but the interpreter does use a fixed order with the use of `map_mapM`.

We have a higher-order relation that forms a bridge between these two models: `map_Forall2`. This relation is very similar to the `Forall2` relation for simple lists. Based on a relation `R` between the value types `A` and `B` of two maps `m1` and `m2` respectively, `map_Forall2 R m1 m2` holds iff `m1` and `m2` have the same domain and the values for the respective keys relate to each other as per the provided relation `R`. It is based on `map_relation` from std++ and has the following simplified definition:

```
map_relation (M := gmap K) R (λ x, False) (λ x, False)
```

where `K` is a key type and `R` is a relation between `A` and `B`, where `A` and `B` can be any type. The two occurrences of `(λ x, False)` instruct `map_relation` what to do in case a key is missing from one of the maps but present in the other; in both cases we return `False` because we require the domains of the two maps to be the same.

Recall that the signature of `map_mapM` is

```
(A → gmap K B) → gmap K A → M (gmap K B)
```

where `M` is a monad, `K` is the key type and `A` and `B` are value types.

Correspondence under the `option` monad. For simple lists, std++ already provides the lemma `mapM_Some`:

`mapM f l = Some k` *if and only if* `Forall2 (λ x y, f x = Some y) l k`

We derive a similar lemma for finite maps.

Lemma 3 (`map_mapM_Some_L`). `map_mapM f m1 = Some m2` *if and only if* `map_Forall2 (λ x y, f x = Some y) m1 m2`.

4.4.4 Soundness

For our soundness proof, we have a stronger theorem that we can use to prove the weaker soundness property. For this stronger theorem, we define the auxiliary function *cforce*:

$$\text{cforce}_{uf}(e) = \begin{cases} e & \text{if } uf = \text{true} \\ \textbf{force } e & \text{if } uf = \text{false} \end{cases}$$

We will now also make use of mathematical notation instead of the notation as in Coq, because this makes the lemmas and proofs easier to read. We use $\forall^2_m x \in m_1, y \in m_2.\ P(x, y)$ to denote `map_Forall2` P m_1 m_2.

Lemma 4 (`force_map_fmap_union`). $\forall^2_m e \in \overrightarrow{b}, v_s \in \overrightarrow{b'_s}.\ \textbf{force } e \xrightarrow{*} v_s$ *implies that* $\textbf{force } \{\ \overrightarrow{b} \cup \overrightarrow{c_s}\ \} \xrightarrow{*} \textbf{force } \{\ \overrightarrow{b'_s} \cup \overrightarrow{c_s}\ \}$.

Lemma 5 (`force_map_fmap`). $\forall^2_m e \in \overrightarrow{b}, v_s \in \overrightarrow{b'_s}.\ \textbf{force } e \xrightarrow{*} v_s$ *implies that* $\textbf{force } \{\ \overrightarrow{b}\ \} \xrightarrow{*} \textbf{force } \{\ \overrightarrow{b'_s}\ \}$.

Proof. This holds trivially by lemma 4 with $\vec{c_s} = \emptyset$. ◻

Theorem 1 (`eval_sound_strong`). *For every expression e, value v' and amount of fuel n, we have that $e \overset{n}{\mapsto}_{uf} v'$ implies that $cforce_{uf}(e) \xrightarrow{*} v$.*

Proof. We perform induction on the 'Partiality Fuel' n. The base case $n = 0$ trivially holds by the definition of eval. In the inductive case, we have the following IH:

$$\forall e\, v'.\ e \overset{n}{\mapsto}_{uf} v' \implies \text{cforce}_{uf}(e) \xrightarrow{*} v'$$

We have that $e \overset{n+1}{\mapsto}_{uf} v'$ and need to prove that $\text{cforce}_{uf}(e) \xrightarrow{*} v'$.

We then do a case distinction on e. In all the cases where e is not a value, we follow the interpreter and use the IH to finish the proof.

In the case where e is a value v, we do a case distinction on v and unfold eval in our assumption by one step. If v is not an attribute set, we have that $v = v'$ by definition of eval. Furthermore, v can be converted to from a strong value. We know that strong values can never be reduced, also not under force, so we have that cforce $uf\ v \xrightarrow{*} v$ by the reflexivity of $\xrightarrow{*}$.

If v is an attribute set, then it depends on the argument uf whether v will be reduced to a strong value. If uf is false, then we are done, just like before. Otherwise, we have that `map_mapM` (`eval` n `true`) $\vec{b}$ = `Some` $\vec{b'}$ and need to prove that $\textbf{force}\ \{\ \vec{b}\ \} \xrightarrow{*} \{\ \vec{b'}\ \}$. Due to the correspondence between `map_mapM` and `map_Forall2`, we have that $\forall^2_{m} e \in \vec{b},\ u \in \vec{b'}.\ e \overset{n}{\mapsto}_s u$. By the induction hypothesis, we can now state that $\forall^2_m e \in \vec{b},\ u \in \vec{b'}.\ \textbf{force}\ e \xrightarrow{*} u$. Because $\textbf{force}\ e \xrightarrow{*} u$ reduces to a value u, u must be a strong value and $\vec{b'}$ must therefore also be a strong value.

Then by lemma lemma 5, we have that $\textbf{force}\ \{\ \vec{b}\ \} \xrightarrow{*} \textbf{force}\ \{\ \vec{b'}\ \}$. By transitivity, we now only still have to prove that $\textbf{force}\ \{\ \vec{b'}\ \} \xrightarrow{*} \{\ \vec{b'}\ \}$. Which follows trivially by FORCE under an empty context, because $\{\ \vec{b'}\ \}$ is a strong value. ◻

4.4.5 Completeness

Lemma 6 (`eval_le`). *If we have $e \overset{n}{\mapsto}_{uf} v$, then for any $m \geq n$, we will also have $e \overset{m}{\mapsto}_{uf} v$.*

Lemma 7 (`eval_step_ctx`). *For some context E that must be strict if uf is true, $e \to_{\text{base}} e'$ and $E[e'] \overset{n}{\mapsto}_{uf} v''$ implies that there exists an amount of fuel m such that $E[e] \overset{m}{\mapsto}_{uf} v''$.*

Lemma 8 (`eval_step`). *$e_1 \to e_2$ and $e_2 \overset{n}{\mapsto} v_3$ implies that there exists an amount of fuel m such that $e_1 \overset{m}{\mapsto} v_3$.*

Proof. The only rule that could have been applied to get $e_1 \to e_2$ is CTX. So there must exist some weak context E and expressions e_1' and e_2' such that $e_1 = E[e_1']$ and $e_2 = E[e2']$. Furthermore, we must also have that $e_1' \to_{\text{base}} e_2'$. Then we are done by trivially applying lemma 7. ◻

Theorem 2 (`eval_complete`). *For every expression e and value v', we have that $e \xrightarrow{*} v$ implies that there exists an amount of fuel n such that $e \overset{n}{\mapsto} v'$.*

Proof by induction on the length of the derivation $e \xrightarrow{*} v'$. Base case: $e = v'$. Then we have that $v' \overset{1}{\mapsto} v'$ because the interpreter returns values verbatim when uf is false. *Inductive case: $e \to e' \xrightarrow{n} v'$.* We have the following IH: $\forall e\, v'.\ e \xrightarrow{n} v' \implies \exists m.\ e \overset{m}{\mapsto} v'$. We need to prove that there exists some amount of fuel k such that $e \overset{k}{\mapsto} e'$. By the IH applied on $e' \xrightarrow{n} v'$ there must be some amount of fuel m such that $e' \overset{m}{\mapsto} v'$. Then by lemma 8 applied with $e \to e'$ and $e' \overset{m}{\mapsto} v'$ we have that there must exist some amount of fuel k such that $e \overset{k}{\mapsto} v'$, so we are done. □

Chapter 5

Related work

In section 5.1, we discuss what problems exist with previous Nix semantics and how Mininix differs from these. Following that, we discuss what features Mininix is missing in comparison to the official Nix interpreter [14] in section section 5.2. In section section 5.3, we discuss alternatives for the ad-hoc capture avoidance/-substitution mechanism that we use in Mininix, particularly looking into De Bruijn indices [9]. Finally, we discuss some non-academic work on Nix in section 5.4.

5.1 Previous Nix semantics

The semantics for Mininix, as described in section 3.3, are largely based on previous work by Dolstra. Two papers and one dissertation are especially relevant: Dolstra's dissertation [11] is the first to describe reduction rules for the Nix language. Dolstra and Löh [13] introduces NixOS and quickly covers the Nix language in lesser detail but with slightly different syntax and reduction rules than Dolstra's disseration. Finally, Dolstra [10] discusses maximal laziness, a property of the Nix interpreter. This paper also glances over the semantics of the Nix language, but does not expand on the semantics of the language beyond his dissertation.

In this section, we will first discuss the limitations of the semantics as described in these works (section 5.1.1). Then, we will talk about the differences between the semantics we describe and these earlier works, and how our semantics solves the issues of these earlier semantics (section 5.1.2).

5.1.1 Limitations of the previous semantics

Other than the fact that the Nix expression language has evolved over the years since these previous works have been published, there are a few particular shortcomings what we would like to point out.

Mixed big-step and small-step reductions. The previous works [11, 13, 10] make use of mixed big-step and small-step reductions in rules, where the conse-

quent uses a single-step reduction, but assumptions use big-step reductions. Take the following rules, respectively taken verbatim from [11] and [13]:

$$\text{SELECT: } \frac{e \overset{*}{\mapsto} \{as\} \wedge \langle n = e' \rangle \in as}{e.n \mapsto e'} \qquad \frac{e \longrightarrow^{*} \{\overrightarrow{b}\} \qquad x = e' \in \overrightarrow{b}}{e.x \longrightarrow e'} \text{ (select)}$$

Although this is not wrong in any respect, it does make the semantics a bit harder to work with, other than being unconventional.

The closedness of terms. Dolstra [11, Lemma 1, p. 85] states that "[e]very term to which a semantic rule is applied during evaluation is closed." The base case states that it trivially holds since "the parser checks that these terms are closed and abors with an error otherwise."

Although this sounds appealing, this cannot be correct. Take the Nix program ⟨`with (let x = x; in x); y`⟩. It is not possible for the parser to statically analyze whether `y` will be bound at `y` or not; this requires evaluation of ⟨`let x = x; in x`⟩, which loops. Similarly, this program would evaluate to 2: ⟨`let foo = { y = 2; }; in with foo; y`⟩. But one would not expect a parser to perform any kind of evaluation here to validate that `y` is bound at `y`.

One could also very well argue that a term such as ⟨`let foo = { y = 2; }; in with foo; y`⟩ should be considered open, because ⟨`let y = 1; in let foo = { y = 2; }; in with foo; y`⟩ evaluates to 1. However, then the base case still does not hold: any program with variables not bound by let bindings or functions, but possibly bound by inclusion constructs, would always be rejected. This would make the inclusion construct practically unusable.

Reduction of and substitution in inclusions. Dolstra [11, Sec. 4.3.3] discusses the substitution function for the Nix language, but mentions no specific case for the inclusion construct. Instead, Dolstra remarks that the substitution function is recursively applied to all subexpressions for all cases not specifically mentioned. Following this logic, we could assume that we have

$$\text{subst}(subs, \text{with } e_1; e_2) = \text{with subst}(subs, e_1); \text{subst}(subs, e_2)$$

Having this substitution function, the program ⟨`with { x = 1; }; with { x = 2; }; x`⟩ returns 1. We could say that 'later' `with` statements do not shadow 'earlier' `with` statements.

However, the official Nix interpreter [14] now interprets this program differently. Instead, the output of this program has become 2. Hence, we can state that 'later' `with` statements have priority over 'earlier' `with` statements. Our semantics follow the official Nix interpreter. We overcome this by using so-called placeholders, which we explain in section 3.3.4.

Furthermore, `with` silently proceeds after evaluating its argument if it is not an attribute set, so the program ⟨`with null; 1`⟩ *will* work and return 1. In the previous semantics, there was no rule for this behavior. We have added the WITH-NO-ATTRSET rule.

Infinite derivation trees. In Dolstra [11], let bindings look very different to those in Dolstra and Löh [13]. In Dolstra [11], let bindings look like ⟨`let { x = 3; body = x; }`⟩ (where the result is `body` with `x` made available, therefore giving 3). In Dolstra and Löh [13], let bindings used the same syntax as those in modern Nix. Now, bindings looked more standard: ⟨`let { x = 3; }; in x`⟩ (gives 3).

Although the Rec/rec rules hardly differ between Dolstra [11] and Dolstra and Löh [13], there is a notable difference in the Let/let rule between the two. The latter still uses a recursive attribute set as a proxy, but now dynamically selects an attribute x to place the body of the let-in statement in. Note that this can allow the creation of infinite derivation trees for expressions such as ⟨`let y = with { z = 1; }; z; in y`⟩ (when selecting `z` for this dynamic attribute x).

Ambiguous equality. Dolstra [11, Sec. 4.3.4, p. 78] does not state much about equality other than that it is defined 'syntactically'. Furthermore, in relation to the maximal sharing feature of the ATerm library, it also states that "[I]f two terms are syntactically equal, then they occupy the same location in memory. This means that a shallow pointer equality test is sufficient to perform a deep syntactic equality test" [11, Sec. 4.4, p. 81]. It remains unclear whether this syntactic equality test is modulo α-conversion for simple lambda abstractions, *i.e.*, whether the terms ⟨`x: x`⟩ and ⟨`y: y`⟩ should be considered equal or not. Modern Nix seems to be more clear on this matter, but also has its warts, as we saw in section 2.5.

5.1.2 Differences with our semantics

First, let us cover two main aspects in which our semantics differ from the previous works:

Terms marked as closed, placeholders Where Dolstra [11] makes use of terms marked as closed as an optimization for substitution, we use terms marked as closed in combination with placeholders to control substitutions. We explain why this is necessary in section 3.3.4. We discuss ways that these may be avoided in section 5.3.

Distinct recursive and non-recursive binders Compared to Dolstra [11] and Dolstra and Löh [13], this is more of an aesthetic rather than a functional choice. Dolstra identifies only one type of binding, but partitions recursive attribute sets (`rec` $\{\overrightarrow{b}\}$) into two sets of binders: `rec` $\{\overrightarrow{b_1}/\overrightarrow{b_2}\}$. The first set of binders $\overrightarrow{b_1}$ contains recursive binders, and the second set of binders $\overrightarrow{b_2}$ contains non-recursive binders introduced by `inherit`.

We retain this distinction, but do not partition binders in recursive attribute sets. Instead, binders can be plain and non-recursive ($x := e$) or recursive ($x :=_r e$). Although not strictly necessary anymore to introduce non-recursive bindings in recursive attribute sets, we do retain the **inherit** construct in Mininix for clarity.

Reduction rules. The semantics for Mininix are largely based on the semantics as described by Dolstra [11] and Dolstra and Löh [13]. Instead of mixed big- and small-step semantics, the Mininix semantics are strict small-step operational semantics with evaluation contexts [16].

Like Dolstra [11], we have a call-by-name semantics that only reduces at the top level of a term. In the context of functions, the semantics also reduce to weak head normal form [22, Sec. 11.3.1]. Although the semantics for Mininix are not deterministic, they are strongly confluent [6, Definition 2.7.3]. For the proof on strong confluence, we refer to appendix A.

Now, let us look at a few rules where there are interesting differences between our semantics and the previous works:

LET. In contrast to both Dolstra [11] and Dolstra and Löh [13], our LET rule is simpler. It is practically a contraction of the LET, REC and SELECT rules from Dolstra and Löh [13].

WITH. As previously discussed in section 2.3, the inclusion construct (`with` statement) has behavior in modern Nix that does not match with the semantics and substitution function described by Dolstra [11].

Our WITH does not differ much from Dolstra's WITH rule [11], except that all substitution terms provided to the parallel substitution function are wrapped in 'placeholders'. In short, this allows later `with` statements to override substitutions performed by earlier `with` statements. We discuss these placeholders in greater detail in section 3.3.4.

We also describe the WITH-NO-ATTRSET rule. The modern Nix interpreter also accepts `with` statements where the left-hand side does not evaluate to an attribute set; in this case, the right-hand side is simply evaluated and returned without any substitution being performed.

APPLY-ATTRSET. Compared to Dolstra [11], our semantics allows default values inside the pattern to refer to each other. This is done so that we align with the semantics of modern Nix. Compared to Dolstra and Löh [13], our matching relation also has support for default values.

OP-HAS-ATTR-NO-ATTRSET. We introduce the OP-HAS-ATTR-NO-ATTRSET rule for the sake of compatibility. Based on observed behavior from the modern Nix interpreter, we have concluded that attribute set membership checks also function if the left-hand side is *not* an attribute set—the result will simply be **false**.

Matching. This relation is based on the matching relation given by Dolstra and Löh [13, Fig. 7]. The relation has been extended to support default values similarly to the β-REDUCE' rule from Dolstra [11, Sec. 4.3.4], but with support for mutual recursion under the default values, as possible in modern Nix.

```
rec {
  hello = "hello";
  x = {
    hello = "hallo!";
    ${null} = 1;  # no binding will be created
  };
  ${"has" + hello} = x ? ${hello};
}.hashello  # true
```

Listing 11: Dynamic attributes in the Nix language.

5.2 Missing features

The primary features of the Nix language that we do not support are dynamic attribute names and derivations. Although we do support attribute paths in selections, we do not support these on the right-hand side of the attribute membership check operator $\langle e\ ?\ x \rangle$. Other than that, we also lack support for more or less trivial features: lists, floating-point numbers, file system paths, and imports.

Fully compatible equality. We discussed the issues with equality in Nix in section 2.5 and section 5.1.1. Our implementation of equality, encoded in the *expreq* function (see fig. 3.8), does recurse into attribute sets, but does not share the pointer equality quirk that Nix has.

Derivations. These are arguably the most essential feature of Nix; they form the core of the build system. We would have been able to include analogues of Dolstra's DERIVATION and DERIVATION! rules [11, Sec. 4.3.4, p. 80]. However, these rules do little more than delay the execution of the primitive operation *instantiate*. Implementing *instantiate* would have been unrealistic due to its sheer complexity. To illustrate: the most certainly dated description of *instantiate* by Dolstra [11, Fig. 5.6] shows that it interacts with the file system and the Nix store.

Of course, it would have been trivial to have *instantiate* as a constant, being part of the set of values in Mininix (section 3.1). We do not believe that doing this would have created any additional value.

Dynamic attributes. Modern Nix allows dynamically naming attributes. For an example, see listing 11.

Although from observed behavior it seems that dynamic attributes (with interpolated names) are not available in other bindings, we do already now see that reduction of the names of attribute set members may get complex.

Interpolation of attribute names is not allowed in formal parameters of functions, and is also forbidden in let bindings. (Although in let bindings, the most simple case `${"test"}` seems to be allowed).

5.3 Bindings and substitution

Bindings in Mininix can be quite complex. There are four sources of bindings: functions, let-in statements, recursive attribute sets and inclusions. Of these four, inclusion also has lower 'priority' than the others. In this thesis, we have used an ad-hoc mechanism using terms marked as closed and placeholders to deal with this complexity.

A lot of research has already been done on the topic of bindings. The well-known De Bruijn indices [9] are specifically interesting for us: they may allow us to simplify our substitution mechanism, and elide mechanism such as terms marked as closed. But De Bruijn's work can not be applied on Mininix directly: De Bruijn [9] makes use of the simple λ-calculus, where only the lambda abstraction introduces bindings. In Mininix, there are four constructs that introduce bindings.

Therefore, before we can make use of De Bruijn indices, we must transform the program into a simpler form (or a more restricted language). Specifically, we must simplify programs so that only simple functions still introduce bindings.

In this section, we quickly discuss how we could transform Mininix programs so that we may be able to apply De Bruijn indices. We also discuss how explicit substitution [1] relates to our work.

Simplifying let expressions. Peyton Jones [22, Sec. 6.2] provides intuition for how recursive let expressions can be simplified to the ordinary λ-calculus. However, Peyton Jones's transformations depend on two things that Mininix does not have:

- pattern matching inside recursive let bindings and
- a distinct kind of non-recursive let construct, in which pattern matching is also supported.

One may argue that Mininix does have pattern-matching functions, and that these could be used for the purpose of pattern-matching like in Petyon Jones' let bindings. However, as we will see later, we express pattern-matching functions in terms of let bindings; having circular dependencies here would not work.

Nevertheless, we can work around this limitation by using non-recursive attribute sets and selection. Although we do not have a distinct non-recursive let construct, we can trivially reduce a let expression with non-recursive bindings:

$$\textbf{let } x_1 \coloneqq e_1; \dots ; x_n \coloneqq e_n \textbf{ in } d \;\equiv\; (x_1 : (\dots (x_n : d)\ e_n) \dots)\ e_1$$

We can also split let expressions that have both recursive and non-recursive bindings:

$$\begin{array}{lcl} \textbf{let } x_1 \coloneqq e_1; \dots ; x_n \coloneqq e_n; & & \textbf{let } x_1 \coloneqq e_1; \dots ; x_n \coloneqq e_n \\ \quad x_{n+1} \coloneqq_r e_{n+1}; \dots ; y_{n+m} \coloneqq_r e_{n+m} & \equiv & \quad \textbf{in let } x_{n+1} \coloneqq_r e_{n+1}; \dots ; y_{n+m} \coloneqq_r e_{n+m} \\ \textbf{in } d & & \qquad \textbf{in } d \end{array}$$

Now we only have let expressions with solely recursive bindings left to simplify. These require some more lifting, as we do not have pattern matching in let bindings.

$$
\begin{array}{l}\textbf{let}\ x_1 \coloneqq_r e_1; \ldots; x_n \coloneqq_r e_n \\ \ \textbf{in}\ d\end{array}
\quad\equiv\quad
\begin{array}{r}\textbf{let}\ y \coloneqq \mathbf{Y}\ (y : \textbf{let}\ x_1 \coloneqq y.x_1; \ldots; x_n \coloneqq y.x_n \\ \textbf{in}\ \{\, x_1 \coloneqq e_1; \ldots; x_n \coloneqq e_n \,\}); \\ \textbf{in let}\ x_1 \coloneqq y.x_1; \ldots; x_n \coloneqq y.x_n\ \textbf{in}\ d\end{array}
$$

Note that we here also make use of y, which is not matched on the left-hand side. This y may not be in $x_1, \ldots, x_n$ and may also not occur in $e_1, \ldots, e_n$ nor in d. We also make use of the fixed point combinator $\mathbf{Y} \equiv f : (x : f\ (x\ x))\ (x : f\ (x\ x))$ [7, Corollary 6.1.3].

Now that we have defined how let expression should be simplified, we can go on to different Mininix constructs that also require simplification.

Recursive attribute sets. These are now quite simple.

$$
\begin{array}{l}\textbf{rec}\ \{\, x_1 \coloneqq e_1; \ldots; x_n \coloneqq e_n; \\ \quad x_{n+1} \coloneqq_r e_{n+1}; \ldots; y_{n+m} \coloneqq_r e_{n+m} \,\}\end{array}
\quad\equiv\quad
\begin{array}{l}\textbf{let}\ x_1 \coloneqq e_1; \ldots; x_n \coloneqq e_n; \\ \quad x_{n+1} \coloneqq_r e_{n+1}; \ldots; y_{n+m} \coloneqq_r e_{n+m} \\ \textbf{in}\ \{\, x_1 \coloneqq x_1; \ldots; x_n \coloneqq x_n + m \,\}\end{array}
$$

Pattern-matching functions. There are two kinds of pattern-matching functions: those that are strict and those that are not. A pattern-matching function that is strict requires the domain of the passed attribute set to be a subset of the set attributes it matches against. For the non-strict case, we can define the following simplification:

$$
\begin{array}{l}\{\, x_1, \ldots, x_n, \\ \ x_{n+1}\ ?\ e_{n+1}, \\ \ \ldots, \\ \ x_{n+m}\ ?\ e_{n+m}, \\ \ \underline{\ldots}\,\} : d\end{array}
\quad\equiv\quad
\begin{array}{l}y : \textbf{let}\ x_1 \coloneqq y.x_1; \ldots; x_n \coloneqq y.x_n; \\ \quad x_{n+1} \coloneqq_r y.x_{n+1}\ \textbf{or}\ e_{n+1} \\ \quad \ldots \\ \quad x_{n+m} \coloneqq_r y.x_{n+1}\ \textbf{or}\ e_{n+m} \\ \ \textbf{in}\ e\end{array}
$$

Here, the underlined dots indicate that the pattern is not strict, just like in section 2.2.

The strict case is more difficult, because we need to perform the subset check that we described before. Because we have no mechanism of deleting attributes in Mininix, this becomes extra tricky. One may think that an equality check between the set of successfully matched attributes and the provided set would suffice, but Mininix only really defines equality on strong values. So this will also not function for terms with redexes.

Instead, we have to concede and assume that Mininix also defines the minus operator for attribute sets, removing attributes from the left-hand side which

are present in the attribute set on the right-hand side. This way, we can construct a static set of attributes $x_1, \ldots, x_{n+m}$ with only dummy values, *e.g.*, $\{ x_1 \coloneqq \textbf{null}; \ldots, x_m \coloneqq \textbf{null} \}$. We can then derive the following simplification:

$$
\begin{array}{lcl}
 & & y : \textbf{assert}\ y - \{ x_1 \coloneqq \textbf{null}; \ldots, x_m \coloneqq \textbf{null} \} = \{\ \}; \\
\{ x_1, \ldots, x_n, & & \quad \textbf{let}\ x_1 \coloneqq y.x_1; \ldots; x_n \coloneqq y.x_n; \\
\quad x_{n+1}\ ?\ e_{n+1}, & \equiv & \qquad x_{n+1} \coloneqq_r y.x_{n+1}\ \textbf{or}\ e_{n+1} \\
\quad \ldots, & & \qquad \ldots \\
\quad x_{n+m}\ ?\ e_{n+m} \} : d & & \qquad x_{n+m} \coloneqq_r y.x_{n+1}\ \textbf{or}\ e_{n+m} \\
 & & \quad \textbf{in}\ e
\end{array}
$$

Inclusions. We cannot simplify inclusions. They have a different priority than bindings generated by functions. Because we cannot always statically analyze what bindings an inclusion expression will generate, we in general can not make use of De Bruijn indices in combination with this feature. But there is a workaround: separating substitution logic for inclusions and the logic for all 'normal' mechanisms.

Split substitution logic. As we discussed before, we can statically convert most constructs that insert bindings to simple functions. Still, inclusions remain and cannot be ported to use De Bruijn indices due to their dynamic nature.

Luckily however, we have that inclusions are weaker than the formal parameters that simple functions bind. This means that we can statically analyze which terms are guaranteed to be bound by simple functions, and which will not be. So we can have two universes, as it were:

- the universe of simple functions, where we make use of De Bruijn indices, and
- the universe of inclusions, where bindings are dynamically generated and where we use names.

Explicit substitutions. Abadi et al. [1] define the λσ-calculus: an extension of the λ-calculus where substitutions are not a meta-operation, but instead are a true part of the language. They also make use of De Bruijn notation [9], and therefore it should not be difficult to adapt our language to also include explicit substitutions.

However, other than for the purposes of optimization in interpreter implementations and ease of reasoning, there does not seem to be an immediate benefit to using these. Nevertheless, a mechanism like that of explicit substitutions may be interesting for inclusions.

Currently, we make use of placeholders to allow 'later' inclusions to shadow 'earlier' inclusions. By instead using a variant of explicit substitutions where we use names in place of De Bruijn indices, we may be able to elide placeholders.

In this variant, we may not process substitutions until the variable with a pending substitution is a top-level redex. Furthermore, we might strip pending

substitutions from matching variables when performing substitution for an inclusion.

Closing thoughts on bindings and substitutions. De Bruijn indices [9] are an interesting way to deal with bindings in Mininix. By using these, we may be able to get rid of terms marked as closed. However, inclusions still remain difficult, although the ideas from explicit substitution [1] may help. We describe future work in the next chapter.

5.4 Non-academic work on Nix

In the introduction, we already mentioned the existence of Tvix [3] and HNix [27]. These are full reimplementations of Nix that also attempt to fully implement the Nix language. However, besides these, there are other efforts around the Nix language as well. Suggestions have been made about how Nix could adopt a type system [25], but do not seem to have gained traction in the last years. Interestingly enough, there are other initiatives such as the Nickel language [21] that seem to position itself as a successor to Nix with static typing. Integration with Nix also seems planned, but unfinished as of yet.

Chapter 6

Conclusions

In this thesis, we have defined a smaller version of the Nix expression language, Mininix, based on earlier work by Dolstra et al. [11, 13, 10]. We have mechanized these semantics with the Coq proof assistant. Furthermore, we have created an interpreter, also mechanized in Coq, that is formally verified to have the behavior as prescribed by our formal definition of the semantics of Mininix.

6.1 Future work

This work lays the groundwork for a well-specified Nix language, for which all behaviors are completely and formally described. However, there is still work to be done.

6.1.1 Running the official test suite

The official Nix interpreter [14] has a test suite that consists of many expressions and their expected outputs. Doing this successfully does require a few key parts: (1) an elaborator that facilitates the translation from Nix to Mininix, and (2) a more compatible Mininix that supports more features from the Nix language.

6.1.2 Missing features

As mentioned in section 5.2, there are still features of the Nix language that are missing from Mininix. To reach parity with the Nix language in terms of core features, more work would be required to see if features such as dynamic attributes could be included.

Although it might be tempting to say that derivation should be investigated, we may have to concede that this also muddies the separation between the Nix package manager itself and the Nix expression language.

Builtins. Although we have already laid some groundwork, our prelude (section 3.2, fig. 3.3) is very small. The actual Nix language has many builtins, which for

example allow mapping over the values of an attribute set (`mapAttrs`) and strictly evaluating some expression before returning another (`deepSeq`). Investigating how the latter relates to our **force** and maximal laziness and term sharing [10] may also be relevant.

Equality quirks. Despite the peculiarity of the equality of functions in the Nix language (as in section 2.5), it may be interesting to see if it is possible to reasonably match the behavior of the Nix interpreter when comparing functions.

Null-terminated byte strings. Our current string type is a sequence of ASCII characters. The official Nix interpreter, being programmed in C++, makes use of the `string` template from the C++ standard library. This makes it so that Nix considers any null-terminated string of bytes as a valid string. If we would indeed want to aim for full compatibility, we should also make strings in the language behave in exactly the same way.

6.1.3 More conventional substitutions

In section 5.3, we have discussed how we may be able to replace our ad-hoc binding system with terms marked as closed and placeholders with something more conventional based on De Bruijn indices [9] and possibly explicit substitutions [1].

However, everything that we have discussed so far remains conjecture. Studying if it is indeed possible to properly apply these two systems may be an interesting first step; seeing how different systems of handling bindings relate to each other in the context of a stripped down version of Mininix would certainly be interesting.

6.1.4 A core calculus

Mininix is not exactly *minimal*. It still contains some features which one may not consider 'core features'. Providing a core calculus for Nix is also not what this thesis intended to do; nevertheless, it could be very interesting to define a core calculus for Nix, which the Nix language can then be expressed in terms of, and researching if such a calculus would any value in the first place.

In section 5.3, we already discussed how a few constructs in Mininix could statically be rewritten to the λ-calculus, albeit with quite some effort. This potential core calculus could then omit features such as let bindings and instead focus on those constructs that are not (trivially) λ-definable. It may be a good idea to make this language more expressive with regard to attribute sets, so that built-in functions can be expressed in terms of the language itself.

6.1.5 A better interpreter

The Mininix interpreter we define is as simple as possible. However, this is at the expense of two things: efficiency and distinguishing non-termination from failure.

Extending this interpreter so that it can report different types of errors would be a good first step; making it more efficient would certainly not be bad, as this also poses interesting technical challenges.

Dolstra [11] describes the Nix interpreter as being *maximally lazy*, *i.e.*, implementing aggressive term sharing and caching. What effects this has on the semantics of the language, if any, might be worth looking into. Furthermore, it may be interesting to see if we can also formally prove such a maximally lazy interpreter to be sound and complete with respect to the semantics that we define.

Acknowledgements

I'd like to thank Robbert, my supervisor, for the weekly meetings and laughs we had, and for the thorough feedback that you provided. Doing this thesis, I felt challenged and learned more than I would have imagined. Your motivation and commitment is inspiring; I'm very glad to have had you as my supervisor!

Of course, I'd also like to thank Herman, my second assessor, for taking the time to evaluate this thesis.

Special thanks go to Jochem and Philipp, who took the time to read through parts of my thesis and to correct my mistakes.

Last but not least, I'd like to thank my parents, friends and familiy for their support during my studies, and especially for their support during this final period.

Bibliography

[1] Martin Abadi et al. "Explicit substitutions". In: *Journal of Functional Programming* 1.4 (Oct. 1991), pp. 375–416. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796800000186. URL: https://www.cambridge.org/core/product/identifier/S0956796800000186/type/journal_article (visited on 2024-05-29).

[2] Vincent Ambo. *Tvix Status - September '22*. TVL's blog. Sept. 12, 2022. URL: https://tvl.fyi/blog/tvix-status-september-22 (visited on 2024-06-24).

[3] Vincent Ambo et al. *Tvix - A new implementation of Nix*. Version (tag) r/7651; commit 04ac1c995. Mar. 5, 2024. URL: https://tvix.dev/ (visited on 2024-03-07).

[4] Nada Amin and Tiark Rompf. "Type soundness proofs with definitional interpreters". In: *ACM SIGPLAN Notices* 52.1 (Jan. 1, 2017), pp. 666–679. ISSN: 0362-1340. DOI: 10.1145/3093333.3009866. URL: https://dl.acm.org/doi/10.1145/3093333.3009866 (visited on 2024-06-13).

[5] Andrew W. Appel and Xavier Leroy. "Efficient Extensional Binary Tries". In: *Journal of Automated Reasoning* 67 (Jan. 12, 2023), Article number 8. DOI: 10.1007/s10817-022-09655-x. URL: https://inria.hal.science/hal-03372247 (visited on 2024-06-11).

[6] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge; New York: Cambridge University Press, 1998. 301 pp. ISBN: 978-0-521-45520-6.

[7] Henk Barendregt. *The lambda calculus: its syntax and semantics*. Rev. ed. Studies in logic and the foundations of mathematics vol. 103. Amsterdam: North-Holland, 1984. 621 pp. ISBN: 0-444-86748-3.

[8] Rutger Broekhoff. *Coq Formalization for Mininix*. Zenodo, June 27, 2024. DOI: 10.5281/zenodo.12208115. URL: https://zenodo.org/records/12208115 (visited on 2024-06-27).

[9] Nicolaas Govert de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (Jan. 1, 1972), pp. 381–392. ISSN: 1385-7258. DOI: 10.1016/1385-7258(72)

90034-0. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340 (visited on 2024-05-28).

[10] Eelco Dolstra. "Maximal Laziness: An Efficient Interpretation Technique for Purely Functional DSLs". In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the 8th Workshop on Language Descriptions, Tools and Applications (LDTA 2008) 238.5 (Oct. 10, 2009), pp. 81–99. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2009.09.042. URL: https://www.sciencedirect.com/science/article/pii/S157106610900396X (visited on 2024-02-29).

[11] Eelco Dolstra. "The purely functional software deployment model". PhD thesis. Utrecht, The Netherlands: Utrecht University, Jan. 18, 2006. ISBN: 9789039341308. URL: https://dspace.library.uu.nl/handle/1874/7540 (visited on 2024-02-28).

[12] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. "Nix: A Safe and {Policy-Free} System for Software Deployment". In: 18th Large Installation System Administration Conference (LISA 04). 2004. URL: https://www.usenix.org/conference/lisa-04/nix-safe-and-policy-free-system-software-deployment (visited on 2024-02-07).

[13] Eelco Dolstra and Andres Löh. "NixOS: a purely functional Linux distribution". In: *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. ICFP '08. New York, NY, USA: Association for Computing Machinery, Sept. 20, 2008, pp. 367–378. ISBN: 978-1-59593-919-7. DOI: 10.1145/1411204.1411255. URL: https://dl.acm.org/doi/10.1145/1411204.1411255 (visited on 2024-02-07).

[14] Eelco Dolstra and The Nix contributors. *Nix*. Version (tag) 2.22.1; commit 293d593. May 9, 2024. URL: https://github.com/NixOS/nix (visited on 2024-06-24).

[15] Eelco Dolstra and The Nix contributors. *Nix Reference Manual*. URL: https://nix.dev/manual/nix/2.22/nix-2.22.html (visited on 2024-06-24).

[16] Matthias Felleisen and Robert Hieb. "The revised report on the syntactic theories of sequential control and state". In: *Theoretical Computer Science* 103.2 (Sept. 14, 1992), pp. 235–271. ISSN: 0304-3975. DOI: 10.1016/0304-3975(92)90014-7. URL: https://www.sciencedirect.com/science/article/pii/0304397592900147 (visited on 2024-06-10).

[17] Jules Jacobs. *Functional Evaluation Contexts*. Sept. 22, 2021. URL: https://julesjacobs.com/notes/functionalctxs/functionalctxs.pdf (visited on 2024-04-25).

[18] Robbert Krebbers. *Efficient, Extensional, and Generic Finite Maps in Coq-std++*. July 29, 2023. URL: https://coq-workshop.gitlab.io/2023/abstracts/coq2023_finmap-stdpp.pdf (visited on 2024-03-14).

[19] Ramana Kumar et al. “CakeML: a verified implementation of ML”. In: *SIGPLAN Not.* 49.1 (Jan. 8, 2014), pp. 179–191. ISSN: 0362-1340. DOI: 10.1145/2578855.2535841. URL: https://doi.org/10.1145/2578855.2535841 (visited on 2024-06-28).

[20] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Commun. ACM* 52.7 (July 1, 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814 (visited on 2024-06-28).

[21] *Nickel.* Version 1.7.0. Modus Create LLC, June 11, 2024. URL: https://nickel-lang.org/ (visited on 2024-06-28).

[22] Simon L. Peyton Jones. *The implementation of functional programming languages.* Prentice-Hall international series in computer science. Englewood Cliffs, NJ: Prentice/Hill International, 1987. xviii, 445. ISBN: 978-0-13-453333-9 978-0-13-453325-4. URL: http://www.zentralblatt-math.org/zmath/en/search/?an=0712.68017 (visited on 2024-02-28).

[23] The Coq Development Team. *The Coq Proof Assistant.* Version 8.19. Zenodo, June 10, 2024. DOI: 10.5281/zenodo.11551307. URL: https://zenodo.org/records/11551307 (visited on 2024-06-12).

[24] The std++ developers and contributors. *Coq-std++: An extended "Standard Library" for Coq.* Version 1.10.0; commit eb2afa52. Apr. 12, 2024. URL: https://gitlab.mpi-sws.org/iris/stdpp/.

[25] Hufschmitt Théophane. “A type-system for Nix”. NixCon 2017 (Unterföhring, Germany). Oct. 28, 2017. URL: http://nixcon2017.org/schedule.nixcon2017.org/system/event_attachments/attachments/000/000/003/original/main.pdf (visited on 2024-06-28).

[26] Jörg Tjalheim and The NixOS Wiki contributors. *Overview of the NixOS Linux distribution.* NixOS Wiki. June 1, 2024. URL: https://nixos.wiki/index.php?title=Overview_of_the_NixOS_Linux_distribution&oldid=13114 (visited on 2024-06-24).

[27] John Wiegley. *HNix.* June 23, 2024. URL: https://github.com/haskell-nix/hnix (visited on 2024-06-24).

Appendix A

Confluence of Mininix

Mininix is strongly confluent. In this appendix, we give a quick sketch of how we prove this property of the semantics.

We cannot prove determinism for Mininix because reductions on attribute sets under force are non-deterministic (see section 3.3.2). Instead, we derive a property that is stronger than strong confluence. We prove that this property implies strong confluence, and also formalize that strong confluence implies confluence. For reference, we show the properties of determinism, strong confluence and confluence in fig. A.1.

Before we start proving confluence, we first prove determinism where possible.

Lemma 9 (`matches_deterministic`)**.** *The matching relation* $\vec{b} \sim m \rightsquigarrow \vec{b}'_r$ *is deterministic.*

Lemma 10 (`binop_deterministic`)**.** *The binary operation relation* $u_1 \llbracket \circledcirc \rrbracket u_2 \mathrel{-\!\circledcirc\!\!\to} v$ *is deterministic.*

Lemma 11 (`base_step_deterministic`)**.** *The base step relation* $e \longrightarrow_{\text{base}} e'$ *is deterministic.*

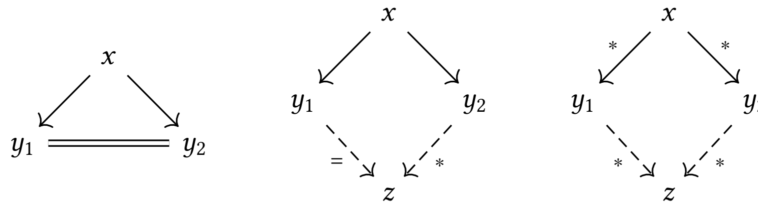

Figure A.1: In left-to-right order: determinism, strong confluence [6, Fig 2.5], and confluence [6, Fig 2.1].

Proof. Because there is no overlap between the rules, this only depends on the determinism of the matching and binary operation reduction relations. By lemmas 9 and 10, we know that these are also deterministic. So we are done. □

Now, let us define a property about $\to$ that is stronger than strong confluence.

Lemma 12 (`step_strongly_confluent_aux`)**.** *For some expressions c and d, we have:*

$$d_1 \leftarrow c \to d_2 \quad \Longrightarrow \quad d_1 = d_2 \;\vee\; \exists e.\, d_1 \to e \leftarrow d_2$$

Proof. This property intuitively makes sense. The only case in which d_1 and d_2 will not be equal is when we choose to reduce different attributes under an attribute set when under force, where the set contains at least two attributes.

Imagine that we have some attribute set with attributes `x` and `y`, where `x` is reduced in d_1 and `y` is reduced in d_2. Then we can apply the same reduction that was done to get d_1 from c on d_2 to get some new term e. We can do the same to d_1 by applying the same reduction that was performed to get d_2 from c on it. This gives us the same term e. This works for all attribute sets with more than one element, as only two distinct elements will be reduced; one by $c \to d_1$ and one by $c \to d_2$. □

Theorem 3 (`step_strongly_confluent`)**.** *The step relation $e \to e'$ is strongly confluent.*

Proof. This is now trivial by lemma 12. □

We also formalize that strong confluence implies confluence. We do this using a sequence of implications. First, we formalize that strong confluence implies semi-confluence [6, Def. 2.1.4, Lemma 2.7.4]. We follow this by formalizing that semi-confluence implies the Church-Rosser property [6, Theorem 2.1.5]. Finally, we have that the Church-Rosser property [6, Def. 2.1.3] is equivalent to confluence [6, Theorem 2.1.5]; we do not have to formalize this as Coq-std++ already includes a proof for this [24, `confluent_alt`].