# Internship report

## Typing the Nix language

Théophane Hufschmitt
supervised by Giuseppe Castagna (CNRS and Université Paris Diderot)
and Mathieu Boespflug (Tweag)

August 21 2017

### General context

Nix [7] is a package manager for Unix-like systems which tries to apply concepts coming from the world of programing languages (functional ones in particular) to package management. This approach solves in a very elegant way many problems encountered by conventional package managers (Apt, Pacman, Yum, …). The state of a machine managed by Nix is (apart from some irreducible mutable state) entirely described by the result of the evaluation of an expression in a (purely functional) specialized language also called Nix.

### Problem studied

The Nix language is not typed, although type tests are present at runtime. The aim of this internship is to design a type-system for this language, and implement a typechecker for it. The goal of this type-system is the same as the one of Typed Racket by Tobin-Hochstad et al. [13]: to add some guaranties to possibly already existing code, while being the least intrusive as possible. This means that we have to adapt to the specificities of the language and to the existing idioms instead of enforcing some new constructs that would be easier to type.

The lack of a type-system is an issue that is often pointed out, but no one has so far attempted to design it (mostly because retro-fitting a type-system to an existing language is a difficult task and the community is still rather small).

### Proposed contributions

This works makes several technical contributions. In particular:

- the compilation of `if–then–else` constructs into typecases to account for occurrence typing,
- an improved definition of bidirectional typing for set-theoretic type-systems. In particular, a technique to propagate type informations through the syntactic tree (especially through lambdas),
- an extension of the gradual type-system of Castagna and Lanvin [3] with records,
- a new way of typing records with dynamic labels that extends the formalism established by Frisch [9] in the case of static records.

But besides these technical aspects, the most important contribution of this work is, by far, that it brings together and integrates into a unique system five different typing techniques that hitherto lived in isolation one from the other, that is:

1. gradual typing [12, 3]
2. occurrence typing [13]

3. set-theoretic types [9]
4. bidirectional typing techniques [10, 8]
5. dynamic records

The important distinctive aspect that characterizes our study is that this integration is studied not for some *ad hoc* toy/idealized academic language, but for an existing programming language with an important community of programmers, with thousands of lines of existing code, and, last but surely not least, which was designed not having types in mind, far from that. The choice of the language dictated the adoption of the five characteristics above: *gradual typing* was the solution we chose to inject the flexibility needed to accept already existing code that would not fit standard static typing disciplines; *occurrence typing* was needed to account for common usage patterns in which programmers use distinct piece of codes according to dynamic type-checks of some expressions; *set-theoretic types* were chosen because they provide both intersection types (we need them to precisely type overloaded functions which, typically, appear when occurrence typing is performed on a parameter of a function) and union types (we need them to give maximum flexibility to occurrence typing, which can typed by calculating the least upper bound of the different alternatives); *bidirectional typing* was adopted to allow the programmer to specify overloaded types for functions via a simple explicit annotation, without resorting to the heavy annotation that characterise functions in most similar type-systems; *dynamic records* were forced on us by the insanely great flexibility that Nix designers have decided to give to their language.

Choosing an existing language also forced us to privilege practical aspects over theoretical ones – with the drawbacks and the advantages that this choice implies. The main drawback is that we had to give up having a system that is formally proved to be sound. For instance, we could have designed a type system in which record field selections are statically ensured always to succeed, but this would have meant to reject nearly all programs that use dynamic labels (which are many); likewise, gradual typing is used to shunt out the type system rather than to insert explicit casts that dynamically check the soundness of programs as Siek and Taha [12] do: in that sense we completely adhere to the philosophy of Typed Racked that wants to annotate and document existing code but not to modify (or, worse, reject) any of it. The implementation of this type system is the most important practical contribution of this work. To this and to the design of how practically integrate types in Nix we dedicated an important part of the time spent on this internship.

## Arguments supporting the validity of these contributions

The implementation is not yet advanced enough to be considered as a finished product, but it already is more than a simple proof of concept. It is capable of efficiently typing a lot of constructs with a reasonable amount of annotations. This was not obviously feasible, as the language is often really permissive, which raises a lot of problems when trying to type it – see a session excerpt in Figure 17 in the appendix.

The developments made on the theoretical side make the use of the type-system more expressive and easier to use thanks to the bidirectional typing, and proved its flexibility by showing that it can easily be adapted to a call-by-name semantics. The use of the gradual typing of Castagna and Lanvin [3] in a more complex type-system also serves as a proof of the accuracy of their approach.

## Summary and future work

The type-system we designed covers most of the requirements for the original practical problem, and was an opportunity for developing the framework of the set-theoretic type-systems.

A logical next step is to develop the implementation further to make it usable at large. This would prove the relevance of this approach for concrete applications.

An annoying lack that would deserve to be filled is the absence of polymorphism in the type system. This was essentially due to the fact that the gradual typing system of Castagna and Lanvin [3] was set in a monomorphic type-system. However, a recent extension by Castagna et al. [5] adds polymorphism to this system, which may be used here.

**Note on the language** A large part of this document is intended to be presented in the conference <Programming> 2018[1], which is why it is written in English.

## Outline

We present in Section 1 the background for our work: Nix (the package manager and the language) which is the object of our work, and the set-theoretic types framework in which the resulting type-system is developped. The Nix language is difficult to reason about because of its flexibility. Because of this, we will not work directly on it, but we designed a simplified version called Nix-light, which will serve as a basis for all the work. We define this Nix-light language and its semantics in Section 2 and we show that a large subset of Nix can be embedded into Nix-light via a compilation process that we describe there. This allows us to backport some properties from Nix-light to Nix. In particular, the semantics of Nix is entirely defined by the semantics of Nix-light. We then present the type-system of Nix-light in Section 3, first by studying a restriction to its core calculus and then adding other constructs. This system is based on set-theoretic types as presented by Frisch [9], with the extension to gradual types elaborated by Castagna and Lanvin [3] and adapted to a lazy semantics as explained by Castagna and Petrucciani [4]. Like for the semantics, the type-system may be transposed back to Nix. Section 4 details some aspects of the implementation and discusses some possible extensions.

# 1 Background

## 1.1 Nix

### 1.1.1 General presentation

**The package manager** Nix [7] – described as "the purely functional package manager" – is a package manager for Unix-like systems. It features a radically new approach to package management, taking most of its inspiration from functional languages: conventional package managers (such as APT, Yum, or Pacman) treat the file system as one giant shared mutable data structure. Installing or removing a package means updating this structure in-place. Such modifications are really hard to control – especially given the fact that parts of them are often delegated to shell scripts whose semantic is notoriously hard to understand[2].

In practice, this model presents several problems. For example, updates can not be made atomic, so the state is incoherent during updates, and an unexpected interruption (electric failure, user interrupt, …) can leave the system in a state where it is not even able to boot. Moreover, the state of the system is hardly reproducible: several machines with the same set of installed packages may be in totally different states depending on the exact sequence of actions that led to having that given set of packages. In particular, installing and then uninstalling a package may not get the system back to its original state.

Nix proposes a radically different approach: from the point of view of the user, the configuration of the system is fully determined as the result of the evaluation of an expression in a pure functional language (also called Nix). Coupled with an on-disk memoization system, this approach brings many improvements, like better reproducibility, transparent rollbacks, atomic upgrades, …

**The Nix language** A huge part of the relevance of this approach relies on the language used to describe the system.

This language is essentially a lazy lambda-calculus, with lists, records and a notion of type at runtime (with functions such as `isInt` which returns *`true`* if and only if its argument is an integer). Additionally, it is untyped (not really by choice, but more because of a lack of time at its beginning – the original author himself considers that "Nix won't be complete until it has static typing" [6]).

[1] http://2018.programming-conference.org/

[2] An ANR project has been recently funded to check bash scripts for this exact use-case – see https://www.irif.fr/~treinen/colis/

Adding a type-system to this language would be an improvement on several aspects:

- `nixpkgs`, the Nix package collection is today several hundreds of thousands of lines of code, with some really complex parts. The absence of typing makes every non-trivial modification particularly complicated and dangerous;
- Partly because of its very fast grow, the project suffers from a huge lack of documentation. In particular, a lot of complex internal abstractions are used but not documented at all. A type-system would reduce the impact of this problem;
- Nix explores some really interesting ideas by applying to system management some principles coming from programing. Adding a type-system to it opens new opportunities in this field. Although this type-system is strictly restricted to the language itself without any interaction with the "package management" part, it offers an opportunity for such an extension.

The language presents many characteristics that constrain a type-system for it:

- It is possible to know at runtime the type of a value. This functionality is used a lot in practice and requires a notion of union types to be useful, as we will see;
- The fields of the records may have dynamically defined labels (i.e., labels which are defined as the result of the evaluation of an arbitrary expression – provided it evaluates to a string);
- The language has been existing for ten years without a type-system. This naturally led to the introduction of several programing patterns are hard to type.

### 1.1.2 Syntax and semantics

The language we study is not the whole Nix language, but a simplified version of it (which omits some minor features without importance for the type system and some others that will be discussed in Section 4.3.4), to which we add *type annotations*. In this document, every reference to Nix is intended to refer to this variant of the language.

The full syntax is given in appendix in Figures 4 and 5 (the added type annotations are highlighted in red). Its semantic is informally described below (the formal semantic is given in Section 2).

This language is a lazily-evaluated lambda calculus, with some additions, namely:

- Constants, let-bindings (always recursive), some (hardcoded) infix operators and `if` constructs;
- Lists, constructed by the `[<expr> ··· <expr>]` syntax;
- Records, defined by the `{ <record−field>; ... <record−field>; }` syntax. The labels of record fields may be dynamically defined as the result of arbitrary expressions (the only limitation being that these expressions must evaluate to string values). All labels in a record must be distinct in order for the record to be well-defined. Records may be recursively defined (using the `rec` keyword), in which case, fields may depend one from another. For example, the expression `rec { x = 1; y = x; }` is equivalent to `{ x = 1; y = 1; }`;
- A syntax for accessing record fields, of the form `<expr>.<access−path>`. Like for the definition of record litterals, the field names may be arbitrary expressions. If the field is not present in the record then a runtime error is thrown, unless a default value is provided (using the `<expr>.<access−path> or <expr>`) syntax.

  For example, `{ x = { y = 1; }; }.x.y` evaluates as `1`, `{ x = { y = 1; }; }.x.z or 2` evaluates as `2` and `{ x = { y = 2; }; }.y` raises an error;

- Lambda-abstractions, that can be defined with patterns. Patterns only exists for records and are of the form `{ <pattern-field>, ⋯, <pattern-field> }` or `{ <pattern-field>, ⋯, <pattern-field>, .. }`, with the `<pattern-field>` construct of the form `<ident>` or `<ident> ? <expr>` (the latter specifying a default value in case the field is absent). The `..` at the end of the pattern indicates that this pattern is *open*, which means that it will accept a record with more fields than the ones that are written.

  Contrary to most languages where the capture variable may be different from the name of the field (for example in OCaml, a pattern matching e record would be of the form `{ x = fieldname1; y = fieldname2; }` and the pattern `{ x; y }` is nothing but syntactic sugar for `{ x = x; y = y }`), Nix requires the name of the capture variable to be the same as the name of the field;

- Type annotations (absent in the actual language but added for this work). These annotations have been added as they are required by the typechecker. There are two productions for types: the static types (noted `<t>`) and the gradual types (noted `<τ>`). The meaning of the types will be presented in Section 3.

  The constructions `<R>` and `<ρ>` represents type regular expressions which will be presented in Section 3.3.1.

  The construction `<constant>` (for a type) represents singleton types: for a constant `c`, the type `c` is the type whose sole value is `c`.

In addition to these syntactic constructions, a lot of the expressiveness of the language resides in some predefined functions. For example, some functions perform some advanced operations on records, like the `attrNames` function, which when applied to a record returns the list of the labels of this record (as strings). Another important class of functions is the set of functions that discriminate over a type, that is functions such as `isInt`, `isString`, `isBool`, and so on, which return *`true`* if their argument is an integer (resp. a string or a boolean), and *`false`* otherwise. When used with a conditional, these functions allow an expression to have a different behaviour depending on the type of an expression. The type system must be able to express this feature.

For example, we want to give the type *`Int`* to an expression such as

```
if isInt x then x else 1
```

The type-system thus has to be aware of the fact that, in the **`then`** branch, every occurrence of the `x` variable has the type $t \wedge Int$ and in the **`else`** branch, every occurrence of `x` has the type $t \setminus Int$, where $t$ is the type of `x` outside the **`if-then-else`** (in particular, this implies recognizing that the condition has a particular form, that needs a special treatment). This type-system feature originally found in Typed Racket [13] is known as *occurrence typing*.

## 1.2 Set-theoretic types

The characteristics of the language enforce several restrictions on the type-system. In particular :

- Because of the important base of (untyped) existing code, the system needs to be flexible enough to accept as many expressions as possible, possibly sacrificing some security guaranties.
- It has to handle anonymous records with dynamic labels, as well as several operations on those records (adding or removing a field, merging records, etc..)
- As shown above, the presence of expressions such as `if isInt x then x else 1` requires the type-system to perform some occurrence typing.
- Related to the occurrence typing, the type-system should also handle some form of union types, so that an expression such as `λx. if isInt x then x==1 else not x` can be typed (with type $Int \vee Bool \to Bool$).

Several existing type-systems already try to satisfy these requirements. The most well-known one is probably the type-system of Typed Racket (Tobin-Hochstad et al. [13]), which already implements at an industrial level most of what is needed in the context of the Scheme language.

The approach we choose here is based on the work of Frisch [9] and Castagna [2] on set-theoretic types, with extensions brought by Nguyễn [11] and Castagna and Lanvin [3].

The system is based on a set-theoretic interpretation of types as a sets of values, which provides a natural interpretation for union, intersection or difference (as the corresponding operations on the underlying sets). This interpretation also provides a natural definition of a subtyping relation which corresponds to the inclusion relation on the interpretations as sets. Moreover, this relation is decidable. The work of Castagna and Lanvin [3] adds gradual typing to this system, which solves even more efficiently the requirement for a flexible type system. Unfortunately, this graduality forbids us from using polymorphism, but a recent extension (Castagna et al. [5]) opens new perspectives on this subject.

This system offers thus more flexibility than most alternatives (allowing in particular arbitrary intersection types, which are a must-have as they allow a precise typing of the overloading of functions[3]), although the (hopefully temporary) sacrifice of polymorphism is a huge price to pay.

# 2 Nix-light

## 2.1 Motivation

Because the Nix language has grown away from any idea of typing, a lot of constructs that require a special treatment from the type-system are not syntactically distinct. In particular, the conditionals that discriminate on the type of a variable (*e.g.* `if isInt x then e else e'`) need some special treatment although they are just some particular instances of a more general construct. Some builtin functions also can not be given an useful type, but their application can, hence it is useful to give them separate typing rules. For example, there is a `mapAttr` function which takes as argument a record `r` and a function `f` and applies `f` to all the values of `r`. This function alone can not be given any type more interesting than the one of a function that takes as argument a record and a function from *Any* (the supertype of all types) to *Any* and returns a record. However, the application of this function to a record and a function can be typed in a way more precise way. For example, the whole expression `mapAttr (λx. x + 1) { x = 1; }` can be given the type `{ x = Int }`. Thanks to the use of overloaded functions, we can even give a useful typing to more complex applications. If `f` is of type `(Int → Bool) ∧ (String → String)` (i.e., `f` is an overloaded function which is both a function of type `Int → Bool` and a function of type `String → String`), we can give the type `{ x = Bool; y = String }` to the expression `mapAttr f { x = 1; y = "foo"; }`. But these two expressions corresponds to two incompatibles types for `mapAttr` (namely `(Int → Int) → { x = Int } → { x = Int }` and `(Int → Bool ∧ String → String) → { x = Int; y = String } → { x = Bool; y = String }`).

Because of this, it is quite hard to reason on Nix. Thus, we decided not to work on Nix itself, but on a simpler language, Nix-light, to which Nix (or at least a large enough subset of it) can be compiled.

## 2.2 Description

**Grammar** Nix-light is similar to Nix, but makes syntactically distinct all the elements which require a special treatment from the typechecker.

The differences are:

- The `if` construct is replaced by the more general "typecase" construct of the form `(x = e0 ∈ t) ? e1 : e2` (which evaluates to `e1[x:=e0]` if `e0` evaluates to a value of type `t` and to

[3] A form of overloading has been recently added to Typed Racket, but not as powerful as the one that intersection types gives us

`e2[x:=e0]` otherwise). The general case `if e0 then e1 else e2` will be compiled to `(x = (e0 : Bool) ∈ true) ? e1 : e2` (where `x` is a fresh variable), whereas particular cases such as `if isInt x then e1 else e2` will be compiled to specialized versions such as `(x = x ∈ Int) ? e1 : e2`. This lightens the type-system as both forms can be treated with the same rule. However, this reduces the expressiveness, as an expression such as `let f = isInt in if f x then x else 1` will not be recognized by the compiler (because it has no type information, so can not see that `f` is the same predicate over types as `isInt` and thus should be treated the same way)[4].

The binding in `(x = e0 ∈ t) ? e1 : e2` is necessary to allow occurence typing (because we must refine the type of a variable, so we can not do anything on `e0` directly).

- The definition of records is simplified: Nix has a specific syntax to define nested records (for example `{ x.y = 1; x.z = 2; }` is equivalent to `{ x = { y = 1; z = 2; }; }`) which is absent in Nix-light.
- Recursive record definitions are not allowed either, they have to be encoded using a (recursive) let-binding.
- List types are replaced by the `Nil` and `Cons(·, ·)` constructors. The encoding is explained in Section 3.3.1.

Nix-light's grammar is given in Figures 6 and 7. The construct $\langle\hat{t}\rangle$ (denoting the types that appears in a typecase) is a non-recursive version of $\langle t\rangle$ (so the typecase is in reality more something like a "kind-case" which just checks for the head constuctor). The type $Any \vee \nabla$ represents an optional field (the reason for this notation is explained in Section 3.3.2). Like in Nix, we impose an additional constraint on records which is that in a record `{ x1 = e1; ...; xn = en }`, the labels `x1`, …, `xn` must be pairwise distinct.

### Semantic

**Pattern-matching** Pattern-matching in Nix-light has a rather classical semantic for a lazy language, with the simplification that, as patterns are not recursive, the argument is either non-evaluated at all, or evaluated in head normal form.

If `r` is a variable pattern – hence in the form `x` or `x:τ` where `x` is an ident and `τ` a type – we define the variable represented by `r` (that we note $\mathscr{V}(r)$) as $\mathscr{V}(x) = \mathscr{V}(x : \tau) = x$. We extend this defintion to a record pattern field `l` (of the form `r` or `r?c` where `c` is a constant) by stating that $\mathscr{V}(r?c) = \mathscr{V}(r)$. In what follows, `l` denotes a record-pattern field of the form `r` or `r?c` (where `c` is a constant and `r` a variable pattern).

For a pattern `p` and a value `v` (resp. an expression `e`), we note $^{p}/_{v}$ (resp. $^{p}/_{e}$) the substitution generated by the matching of `v` (resp. `e`) against `p`. It is defined in Figure 8. The basic idea from which all the rules derive is that the matching of any expression `e` against a variable pattern will produce the substitution `x := e` and that the matching of a record value `{ x1 = e1; ...; xn = en }` against a record pattern `p` with fields `x1`, …, `xn` will produce the substitution `x1 := e1; ...; xn := en`.

**Operational semantic** The full semantic is given in Figure 9.

The reduction rules should be self-explanatory. The only rules worth mentioning are the two rules for the typecase, which involve a typing judgement (and thus make the semantics typing dependent). This typing judgement is however very simple at it simply checks the toplevel constructor of the given value. Its definition is given in annex in Figure 13.

[4]Typed racket [13] does some tracking of type predicates in the type-system itself by annotating arrows. This can also be achieved (and in a less intrusive way) using this type-system and has in fact been implemented. This will be discussed in Section 4.3.4

## 2.3 From Nix to Nix-light

**Compilation** A Nix program `p` is compiled to a Nix-light program $[\![p]\!]$ according to the rules of Figure 11 (except for the type annotations). The type annotations are exactly the same in Nix and Nix-light, and the compilation is just an identity mapping, except for the lists types which differ. The syntax of regular expression lists as well as their compilation to a combination of **`Cons`** and **`Nil`** types is covered by Frisch [9] (the basic idea of this compilation is reminded in Section 3.3.1).

Both languages are quite similar, hence most transformations are simply a transposition of a given structure to the same structure in the other language.

The if construct is compiled to a typecase, with two separate rules depending on the form of the form of the condition:

- The general case is to compile `if e0 then e1 else e2` to `(x = (e0 : Bool) ∈ true) ? e1 : e2` with `x` a fresh variable. The semantics of the target expression is defined on a larger domain than the one we expect from the source one, as it is defined even if `e0` evaluates to something other than a boolean. However, the type annotation `e0 : Bool` enforces that (if the expression is well-typed) `e0` evaluates to a boolean. Hence, the semantic coincides for well-typed terms.
- If `e0` is of the form `isT x` where `isT` is a discriminant of the type `t` (i.e., a builtin function that returns `true` if its argument is of type `t` and `false` otherwise), then the expression will be compiled to `(x = x ∈ t) ? e1 : e2`. It is easy to check that this has the same semantics as the expression that would have been obtained without this special case.

The compilation of records is slightly complex, for two reasons:

- The first one is that Nix has a special construct for recursive records, whereas Nix-light does not, which means this construct has to be encoded (using a let-binding as they are recursive).
- The second is the Nix shortcut for writing nested records ( `{ x.y = 1; x.z = 2; }` which is equivalent to `{ x = { y = 1; z = 2; }; }`). As this syntax has been removed in Nix-light, we have to transform it into the explicit form.

  The real problem arises when this syntax is mixed with dynamic labels, because it is then no longer possible to statically determine the shape of the record. For example, the record `{ ${e1}.x = 1; ${e2}.y = 2; }` may be either of the form `{ z = { x = 1; y = 2; }; }` or `{ z1 = { x = 1; }; z2 = { x = 2; }; }` depending on whether `e1` and `e2` evaluate to the same value `z` or not. It is then impossible to compile this exactly.

  The solution we use here is to always assume that they are different. If the type-checker confirms this, then everything is fine. Otherwise, the program will not correctly typecheck, so while this is too restrictive, we at least do not introduce an incorrect behaviour for well typed programs.

We define a function flatten to transform a non recursive record `{ ap1 = e1; ...; apn = en }` into a record without the nested records syntax (which we call a "flattened" record definition). Its definition is given in Figure 10.

The definition is quite contrived but is simply the generalisation of the idea that we want to transform `{ x.y = 1; x.z = 2; }` into `{ x = { y = 1; z = 2 }; }`

To translate a recursive record definition, we first flatten it using the flatten function defined above. A flattened recursive record definition `rec { x1 = e1; ...; xn = en; }` is then translated to a non-recursive one by the derec function defined as

$$\text{derec}\left(\text{ rec } \left\{\overline{x_i = e_i}^{i\in I}\right\}\right) = \text{let } \overline{x_i = e_i}^{i\in I} \text{ in } \left\{\overline{x_i = x_i}^{i\in I}\right\}$$

The source Nix program is preprocessed before this compilation to make the compilation of conditionals more precise: we want for example the expression `if (isInt x | isBool x) then x else false` (where | is the boolean disjunction) to be given the type `Int ∨ Bool`, but the compilation process will compile it as `(y = (isInt x | isBool x) ∈ true) ? x : false`, so we will not be able to perform occurrence typing on `x`. A simple pre-processing however is sufficient here, as this expression is semantically equivalent to `if isInt x then x else (if isBool x then x else false)`, and the latter will be compiled to the Nix-light expression `(x = x ∈ Int) ? x : ((x = x ∈ Bool) ? x : false)`, on which we will be able to perform occurrence typing for `x`. This pre-processing can be done for any boolean combination of expressions in the `if` clause thanks to the rewriting rules of Figure 12.

# 3 Typing

## 3.1 Types and subtyping

### 3.1.1 Presentation of the types

The types are inspired by the set-theoretic types used in the ℂDuce language and described by Frisch [9]. Their syntax is given alongside Nix-light's grammar in Figure 6.

We distinguish two type productions: the static types (noted by roman letters: $\mathsf{t}, \mathsf{s}, \ldots$) and the gradual types (in greek letters: $\tau, \sigma, \ldots$). These productions essentially correspond to the types presented by Castagna and Lanvin [3], with as addition the record and list types which are similar to the ones presented in [9].

A type `t` is interpreted as the set of all the values of type `t`. The union (∨), intersection (∧) and difference (\) connectives correspond respectively to the set-theoretic union (∪), intersection (∩) and difference (\). So an expression of type `t1 ∨ t2` is an expression that can be either of type `t1` or of type `t2`; an expression of type `t1 ∧ t2` is an expression that is both of type `t1` and of type `t2` and an expression of type `t1 \ t2` is an expression that is of type `t1` but not of type `t2`.

Following the work of Frisch [9], the types also include singleton types (i.e., for every constant `c`, there exists a type `c` such that `c` is the only value of type `c`). For example, the type `Bool` is not a builtin type, but is the type `true ∨ false` (where the types `true` and `false` are the singleton types associated to the constants `true` and `false`).

### 3.1.2 Subtyping

Like in [3], the subtyping relation $\leq$ on static types is extended into a relation $\tilde{\leq}$ on gradual types. However, the subtyping relation is not the same. Indeed, the relation used in [3] – which is the relation established in [9] – is based on an interpretation of types as sets of values, and used directly cause some safety problems on a lazy semantic. The reason for this is that the interpretation supposes fully evaluated values, while a lazy language manipulates possibly non-evaluated expressions, whose evaluation may never finish if forced. In particular, the `Empty` types behaves differently: there exists no value of this type (so its interpretation as a set is naturally the empty set), but there may be expressions of this type, which themselves may appear inside values in a lazy setting. For example, the type `{ x = Empty }` would not be inhabited in a strict semantic (so it would itself be a subtype of every other type, the same way as `Empty` itself), whereas in a lazy semantic such as the one of Nix-light, it is inhabited for example by `{ x = let y = y in y; }`.

It is possible to modify this interpretation to take this difference into account, by adding at some places a special constant (noted $\bot$) to the interpretation of the types. For example, assume for a moment that our types contain a product constructor $\cdot \times \cdot$. The interpretation $[\![\mathsf{A} \times \mathsf{B}]\!]$ of the type $\mathsf{A} \times \mathsf{B}$ is $[\![\mathsf{A}]\!] \times [\![\mathsf{B}]\!]$ in the interpretation of Frisch [9]. Here, it becomes $([\![\mathsf{A}]\!] \cup \bot) \times ([\![\mathsf{B}]\!] \cup \bot)$.

This new interpretation (which is described by Castagna and Petrucciani [4]) entails a new subtyping algorithm that fits the requirements of a lazy semantic. It is this relation (that we note $\leq$) that we use and

extend to the gradual subtyping relation $\tilde{\leq}$.

## 3.2 Functional core

### 3.2.1 General description

In this part, we consider a restriction of the language to a subset of itself consisting of a lambda-calculus with typecase (i.e., we exclude lists and records and all the related operations which will be dealt with in Section 3.3).

The type-system is divided into two parts: an *inference* system and a *check* one. The first one corresponds to classical bottom-up type inference while the second one is a top-down system which does not do any inference but only tries to check that an expression accepts a given type. This double system is an extension of the use of type annotations in inference algorithms which adds the possiblility of using them in a non-local way. This result may be obtained in many languages by simply preprocessing the code and trying to propagate the type annotations (that is for example what OCaml does in order not to enforce the programmer to directly annotate the return type of a pattern-matching on a GADT, but let him simply annotate for example the top-level expression). In Nix-light however, the presence of union and intersection types makes certain expressions impossible to annotate because they will be given several different types under different type environments.

For example, consider the following expression:

```
let f =
  λcond.λx.
    (_ = cond ∈ true) ? x + 1 : not x
in f
```

We wish to give it the type `(true → Int → Int) ∧ (false → Bool → Bool)`. This means that `x` must have the type `Int` if `cond` is `true`, and `Bool` otherwise. It is thus impossible to annotate `x` directly – except by the type `Int ∧ Bool` which equals `Empty`, or by the gradual type.

However, the *check* type-system is able – if we annotate `f` with the type `(true → Int → Int) ∧ (false → Bool → Bool)` – to check that the expression indeed admits both types `true → Int → Int` and `false → Bool → Bool` (and thus their intersection).

### 3.2.2 Typing of patterns

For the restriction of the language that we consider here, the only possible patterns are `<var>` and `<var>:τ`.

We define two operators $\langle p \rangle$ and ${}^{p}/_{\tau}$ which corresponds respectively to the type accepted by a pattern $p$ (i.e., the type whose interpretation is the set of all the values accepted by the pattern) and the typing environment generated by the matching of a type $\tau$ against the pattern $p$ by:

$$\langle x \rangle = ?$$
$$\langle x : \tau \rangle = \tau$$

and

$${}^{x}/_{\tau} = x : \tau$$
$${}^{x:\tau}/_{\sigma} = \sigma \wedge \tau$$

By default (in the absence of annotation), the type accepted by a pattern is the gradual type. This choice, combined with the rule *IAbs* in figure Figure 1, makes the type-system very permissive in the absence of annotations.

### 3.2.3 Typing rules

The judgement $\Gamma \vdash^{\Uparrow} e : \tau$ corresponds to the *inference* system and the judgement $\Gamma \vdash^{\Downarrow} e : \tau$ to the *check* system. The notation $\Gamma \vdash^{\delta} e : \tau$ means either $\Gamma \vdash^{\Uparrow} e : \tau$ or $\Gamma \vdash^{\Downarrow} e : \tau$ (but always the same in a given inference rule).

The rules for both systems are given in Figure 1.

**Constants and variables** We assume the existence of a function $\mathscr{B}$ which associates to each constant $c$ its type $\mathscr{B}(c)$.

**Lambda-abstractions** To *infer* the type of a lambda-abstraction $\lambda p.e$ under the hypothesis $\Gamma$, we first determine its domain, which is the accepted type of the pattern $p$. Then we compute its codomain, which is the type of the body $e$ under the extra-hypothesis provided by the pattern. These extra hypothesis is given by $p/\langle p \rangle$. Indeed, $\langle p \rangle$ is by definition the most general type that the pattern accepts, so this environment is the most general one under which we may have to type the body.

The *check* is somehow more complex. The idea is that to check that an expression $\lambda p.e$ admits the type $\tau$ under the hypothesis $\Gamma$ we must check that each arrow type $\sigma_1 \to \sigma_2$ "contained" in $\tau$ is admitted by the expression, which means that under the hypothesis $\Gamma; p/\sigma_1$, $e$ admits the type $\sigma_2$.

To properly define this notion of "containment", we introduce the $\mathscr{A}$ operator defined as follows for a type $\tau$ subtype of $\texttt{Empty} \to \texttt{Any}$:

If $\tau$ is of the form

$$\tau = \bigvee_{i \in I} \left( \bigwedge_{p \in P_i} (\sigma_p \to \tau_p) \wedge \bigwedge_{n \in N_i} \neg(\sigma_n \to \tau_n) \right) \tag{1}$$

then $\mathscr{A}(\tau)$ is defined as

$$\mathscr{A}(\tau) = \bigsqcup_{i \in I} \{\sigma_p \to \tau_p | p \in P_i\}$$

where $\sqcup$ is itself defined as

$$\{\sigma_i \to \tau_i | i \in I\} \sqcup \{\sigma_j \to \tau_j | j \in J\} = \{(\sigma_i \wedge \sigma_j) \to (\tau_i \vee \tau_j) | i \in I, j \in J\}$$

Frisch [9] shows that $\tau$ can always be expressed in the form of the equation 1 (in fact this possibility is fundamental for the subtyping algorithm as shown by Castagna [2]).

In the example of Section 3.2.1 the type $\tau$ is equal to $(\texttt{true} \to \texttt{Int} \to \texttt{Int}) \wedge (\texttt{false} \to \texttt{Bool} \to \texttt{Bool})$, so $\mathscr{A}(\tau)$ is the set $\{\ \texttt{true} \to \texttt{Int} \to \texttt{Int}; \texttt{false} \to \texttt{Bool} \to \texttt{Bool}\ \}$. This means that the *CAbs* rule checks that the function `f` has both types in this set.

**Application** The inference rule for the application is different from the one used in simply typed lambda-calculus: Because of the presence of union and intersection types, the types of functions may not be just arrow types, but in general, they are all the subtypes of the $\texttt{Empty} \to \texttt{Any}$ type. As consequence, the definitions of the domain and image of such function types is more complex.

We reuse the definitions of the $\tilde{\text{Dom}}$ and $\tilde{\circ}$ operators defined by Castagna and Lanvin [3]. The intuition for these operators is that $\tilde{\text{Dom}}(\tau)$ is the domain of the functions of type $\tau$ and $\tau \tilde{\circ} \sigma$ is the image by $\tau$ of all the elements of type $\sigma$. For example, $\tilde{\text{Dom}}(\tau_1 \to \sigma_1 \wedge \tau_2 \to \sigma_2)$ is $\tau_1 \vee \tau_2$, and $(\tau_1 \to \sigma_1 \wedge \tau_2 \to \sigma_2) \tilde{\circ} \tau_1$ is $\sigma_1$ (provided that $\sigma_1 \wedge \sigma_2 = \text{Empty}$). Once these operators are defined, the inference rule for the application is rather straightforward.

The *check* rule is simpler, but requires using the inference system, as the type of the argument is unknown. Hence, to check that $\Gamma \vdash^{\Downarrow} e_1\ e_2 : \tau$, we first *infer* the type $\sigma$ of $e_2$, and then we check that $e_1$ has type $\sigma \to \tau$. We also could imagine first inferring the type $\tau'$ of $e_1$, and then try to calculate the preimage $\sigma'$ of $\tau$ by $\tau'$ and check that $e_2$ has type $\sigma'$. However, this approach is more complicated (it is

unsure whether there is an easy way to calculate this preimage) and probably less useful in practice (as inferring the type of functions is often the difficult part while checking it is easier).

**Let-bindings** The let-bindings are the places where the system goes from inference to checking: for each variable that is being defined, if it is annotated, then we *check* that its definition has the given type (else we simply *infer* the type of its definition). As let-bindings are recursive and no unification is made, the non-annotated variables must be given a default type to type the definitions. The chosen type is `?` (although choosing `Any` instead is an equally reasonable choice, simply more restrictive as for instance an expression of type `?` can be the argument of any function – because `?` is a gradual subtype of every other type – while an expression of type `Any` can only be the argument of a function whose domain is `Any` or `?`). This rule has the advantage of being quite general. In particular it can type non-recursive let-bindings without requiring annotations and without loss of precision.

**Typecase** The typing of the typecase uses *occurrence typing*. This means that when typing the expression `(x = e ∈ t) ? e1 : e2`, the type of `x` will be refined in each branch: its occurrences will be assumed to be of type $t_e \wedge t$ in `e1` and of type $t_e \wedge \neg t$ in `e2` (where $t_e$ is the type of `e`). Moreover, if the system infers that `e` is always of type `t` (or $\neg t$), then the dead branch does not need to be typed. This is what the two implications in the inference rule express: If one of the two conditions will never be met (or both), then there is no need to type the corresponding branch. This particular characteristic may seem undesirable (as it implies in particular that an expression may be well-typed while some of its sub-terms are not), but the need for it is clear if we once again consider the example of Section 3.2.1. Indeed, we want the body of the function to be well-typed under the hypothesis `cond: True; x : Int`, while `not x` is not (but is never reached under those hypothesis). One last particularity is that both branches may have different types, the final type being the union of them. For example, we want to accept the expression `(x = e ∈ Int) ? x : not x` although the then-branch is of type `Int` and the else-branch of type `Bool`. Of course, it may be the case that the fact that the two branches have two different types is an error, in which case the type error will probably occur anyway, but at the place where the result of this expression is used rather than at the place of its definition.

The check works the same way, except that we can directly check the same type for both branches. Note that we have to infer the type of the tested expression since we can not know it in advance.

## 3.3 Data structures

We now extends our core calculus with the two builtins data structures of Nix (lists and records).

### 3.3.1 Lists

Unlike most languages, Nix-light does not have a "list" type. Lists are instead a collection of types. Indeed, lists are not homogeneous like they use to be but may contain elements of arbitrary different types.

In Nix, they correspond to the types `[ρ]`[5]. The regular expressions ρ describe the types of the elements of the list the same way as a textual regular expression describes a family of characters sequence. For example, `[ Int* ]` is the type of (possibly empty) lists whose elements are of type `Int`, `[ Bool Int* ]` the type of lists containing a boolean and then an arbitrary number of integers and `[ (Bool + | (Int String*)) Int? ]` is the type of lists that contain either at least one boolean, or an integer followed by any number of strings, and may end with an integer.

In Nix-light, we define list types as the set of types inductively generated by the following grammar:

```
<l> ::= Nil
  | Cons(<τ>, <l>)
  | let <x> = <l> in <x>
```

[5]To keep things simple, we only consider gradual list types, but all this section also applies to concrete types

$$\frac{}{\Gamma \vdash^{\Uparrow} x : \Gamma(x)}(\mathit{IVar}) \qquad \frac{\Gamma(x)\tilde{\leq}\tau}{\Gamma; \vdash^{\Downarrow} x : \tau}(\mathit{CVar}) \qquad \frac{}{\Gamma \vdash^{\Uparrow} c : \mathscr{B}(c)}(\mathit{IConst})$$

$$\frac{\mathscr{B}(c)\tilde{\leq}\tau}{\Gamma \vdash^{\Downarrow} c : \tau}(\mathit{CConst}) \qquad \frac{\Gamma; {}^{p}/_{\langle p \rangle} \vdash^{\Uparrow} e : \tau}{\Gamma \vdash^{\Uparrow} \lambda p.e : \langle p \rangle \to \tau}(\mathit{IAbs})$$

$$\frac{\tau \leq \text{Empty} \to \text{Any} \qquad \forall \sigma_1 \to \sigma_2 \in \mathscr{A}(\tau), \qquad \Gamma; {}^{p}/_{\sigma_1} \vdash^{\Downarrow} e : \sigma_2}{\Gamma \vdash^{\Downarrow} \lambda p.e : \tau}(\mathit{CAbs})$$

$$\frac{\Gamma \vdash^{\Uparrow} e_1 : \tau_1 \qquad \Gamma \vdash^{\Uparrow} e_2 : \tau_2 \qquad \tau_1 \tilde{\leq} \text{Empty} \to \text{Any} \qquad \tau_2 \tilde{\leq} \tilde{\text{Dom}}(\tau_1)}{\Gamma \vdash^{\Uparrow} e_1\ e_2 : \tau_1 \tilde{\circ} \tau_2}(\mathit{IApp})$$

$$\frac{\Gamma \vdash^{\Uparrow} e_2 : \sigma \qquad \Gamma \vdash^{\Downarrow} e_1 : \sigma \to \tau}{\Gamma \vdash^{\Downarrow} e_1\ e_2 : \tau}(\mathit{Capp})$$

$$\frac{\begin{array}{c}\forall i \in I, \Gamma; \overline{\phantom{x}}^{i' \in I}; \overline{x_{i'} : ?}; \overline{\phantom{x}}^{j \in J}; \overline{x_j : \tau_j} \vdash^{\Uparrow} e_i : \tau_i \\ \forall j \in J, \Gamma; \overline{\phantom{x}}^{i \in I}; \overline{x_i : ?}; \overline{\phantom{x}}^{j' \in J}; \overline{x_{j'} : \tau_{j'}} \vdash^{\Downarrow} e_j : \tau_j \qquad \Gamma; \overline{\phantom{x}}^{i \in I}; \overline{x_i : \tau_i}; \overline{\phantom{x}}^{j \in J}; \overline{x_j : \tau_j} \vdash^{\delta} e : \tau\end{array}}{\Gamma \vdash^{\delta} \text{let } \overline{\phantom{x}}^{i \in I}; \overline{x_i = e_i}; \overline{\phantom{x}}^{j \in J}; \overline{x_j : \tau_j = e_j}; \text{ in } e : \tau}(\mathit{Let})$$

$$\frac{\Gamma \vdash^{\Uparrow} e : \tau \qquad \tau \not\leq \neg t \Rightarrow \Gamma; x : \tau \wedge t \vdash^{\Uparrow} e_1 : \sigma_1 \qquad \tau \not\leq t \Rightarrow \Gamma; x : \tau \wedge \neg t \vdash^{\Uparrow} e_2 : \sigma_2}{\Gamma \vdash^{\Uparrow} (x = e \in t)?e_1 : e_2 : \sigma_1 \vee \sigma_2}(\mathit{ITcase})$$

$$\frac{\Gamma \vdash^{\Uparrow} e : \tau \qquad \tau \not\leq \neg t \Rightarrow \Gamma; x : \tau \wedge t \vdash^{\Downarrow} e_1 : \sigma \qquad \tau \not\leq t \Rightarrow \Gamma; x : \tau \wedge \neg t \vdash^{\Downarrow} e_2 : \sigma}{\Gamma \vdash^{\Downarrow} (x = e \in t)?e_1 : e_2 : \sigma}(\mathit{CTcase})$$

Figure 1: Inference and check rules for the lambda-calculus with typecase

$$\frac{\begin{array}{c}\Gamma \vdash^{\delta} e_1 : \tau_1 \\ \Gamma \vdash^{\delta} e_2 : \tau_2 \\ \tau_2 \tilde{\leq} \text{Cons}(\text{Any}, \text{Any})\end{array}}{\Gamma \vdash^{\delta} \text{Cons}(e_1, e_2) : \text{Cons}(\tau_1, \tau_2)}(\textit{Cons}) \qquad \frac{\Gamma \vdash^{\Uparrow} e : \tau \qquad \tau \tilde{\leq} \text{Cons}(\tau_1, \tau_2)}{\Gamma \vdash^{\delta} \text{Head}(e) : \tau_1}(\textit{Head})$$

$$\frac{\Gamma \vdash^{\Uparrow} e : \tau \qquad \tau \tilde{\leq} \text{Cons}(\tau_1, \tau_2)}{\Gamma \vdash^{\delta} \text{Tail}(e) : \tau_2}(\textit{Tail})$$

Figure 2: Typing rules for lists

```
| <l> ∧ <l> | <l> ∨ <l>
| ?
```

Frisch [9] shows that these types can encode the regular expression list types. The encoding is not reminded here as it is rather verbose (we just show a couple of examples below), but it is essentially a transposition in the domain of types of the compilation of regular expressions into finite automatons.

For example, the type `[ Int* ]` is compiled to `let X = Cons(Int, X) ∨ Nil in X`. The type `[ (Int | Bool*) ]` is compiled to `let X = Cons(Int, X) or Nil in X | Cons (Int, Nil)`.

**Typing** The typing of lists is presented in Figure 2.

### 3.3.2 Records

Typing anonymous records is often difficult. Nix (and as consequence Nix-light) pushes this to its limits by making them insanely flexible. It is thus impossible to type the records with enough accuracy to cover every usage case, so one important thing to know is how they are used in practice to understand in which direction to go.

In Nix, there are two main uses of records

- Static records: This is the classical use of records as they are used in statically typed languages: some predefined fields contain various informations about a structure. In this setting, the labels are all statically defined as expressions whose type is type is a singleton string[6] and the different fields are likely to contain values of different types.
- Dynamic maps: In this case, the record is used to store arbitrary key-value associations (where the key is a string). The labels here may be defined as arbitrary expressions, however most of the time the values will all be of the same type.

Castagna [2] presents a powerful formalism to handle them in absence of dynamic labels.

We use here an extension of this formalism to allow dynamic labels, which lets us nicely handle both use-case. (we also slightly modify it to meet the requirements of lazy evaluation and gradual typing). We assume the existence of a distinguished constant type $\nabla$ which represents an absent field in a record type.

$\{ x_1 = \tau_1; \cdots; x_n = \tau_n; \_ = \tau \}$ is the type of a record whose fields $x_1, \cdots, x_n$ are respectively of type $\tau_1, \cdots, \tau_n$ and whose other fields are of type $\tau$ (all these fields being possibly equal to or

[6]For some shameful reasons, it sometimes happens that their type is a finite union of singleton strings, but the rules are easy to extend to this use-case

containing the special $\nabla$ field meaning that they are not defined in the record). We write `{ x1 = τ1; ⋯; xn = τn }` as a shorthand for `{ x1 = τ1; ⋯; xn = τn; _ = ∇ }` and `{ x1 = τ1; ⋯; xn = τn; .. }` for `{ x1 = τ1; ⋯; xn = τn; _ = Any ∨ ∇ }`. The former corresponds to a *closed* record type (i.e., a record of this type will have *exactly* the listed fields with the given types) while the latter corresponds to an *open* record type (i.e., a record of this type will have *at least* the listed fields with the given types).

An optional field of type $\tau$ is a field of type $\tau \vee \nabla$ (i.e., a field which may be either of type $\tau$ or undefined). We write `x =? τ` as a shorthand for `x = τ ∨ ∇`.

In this formalism, $t(x)$ is the type associated to $x$ in the record type $t$.

Record types are (as shown by Castagna [2]) unions of atomic record types (types of the form `{ s1 = τ1; ...; sn = τn; _ = τ }`). All the operations that we define on atomic record types may be extended to those unions. We also extend them to gradual types by identifying `?` and `{ _ = ? }`.

The typing of records is done in three steps: we first define the typing of literal records, and we then define the typing of merge operation to type more complex records. Finally, we define the typing of field access.

The rules are given in Figure 3. In these rules, the $s_i$ denote singleton string types.

**Literal records** We have two rules for the literal records, corresponding to the two use-cases of records:

- The *RFinite* rule handles the case of static records. It applies when the labels have singleton types and corresponds to the classical typing rule for records in absence of dynamic labels.
- The *IRInfinite* and *CRInfinite* rules handle the case of dynamic maps. In this case, we do not try to track all the elements of the record, we just give it the type we would assign e.g. to a `Map` in OCaml, which is the type of a record whose elements are all of the same type and are all optional. We take $\bigvee_{i \in I} \sigma_i$ as the type of the elements as it is the least common subtype of the elements we introduced.

  In this case, we decided to give up safety for more flexibility: as a literal record in Nix must have all its labels different[7], a really safe version of the rules would forbid the definition of any non-trivial record with a dynamic label (a definition such as `{ e = 1; e' = 2; }` could not be accepted as in the general case it is not possible to prove that `e` and `e'` will not evaluate to the same value). We allow this in our system – although the implementation may emit a warning (depending on the configuration).

**Concatenation of records** We define the concatenation $r_1 + r_2$ of two record types:

If $\tau_1$ and $\tau_2$ are atomic record types then $\tau_1 + \tau_2$ is defined by:

$$(\tau_1 + \tau_2)(x) = \begin{cases} \tau_1(x) & \text{if } \tau_1(x) \wedge \nabla \leq \text{Empty} \\ (\tau_1(x) \backslash \nabla) \vee \tau_2(x) & \text{otherwise} \end{cases}$$ ∎

There are in fact three possible cases in this formula:

- If $\tau_1(x)$ does not contain $\nabla$, the field is defined in $\tau_1$, and we take its type $\tau_1(x)$ for $(\tau_1 + \tau_2)(x)$,
- if $\tau_1(x)$ is $\nabla$, then the field is undefined in $\tau_1$ and the type of $(\tau_1 + \tau_2)(x)$ is the type of $\tau_2(x)$,
- else, the field may be defined and $\tau_1(x) = \tau_x \vee \nabla$ for some type $\tau_x$. The type of the result may be $\tau_x$ or $\tau_2(x)$.

[7] This is unlike most other dynamic programming languages such as Perl or Python. In those languages, a field may be defined twice in a literal record, in which case the last declaration has precedence over the others

This definition enjoys a natural extension to arbitrary record types (i.e., subtypes of `{..}`), as shown by Castagna [2]: a record type can be expressed as an union of atomic record types, and we extends the $+$ operator by stating that

$$\left(\bigvee_{i\in I} \tau_i\right) + \left(\bigvee_{j\in J} \tau_j\right) = \bigvee_{i\in I, j\in J} (\tau_i + \tau_j)$$

This effectively allows us to type the union of expressions whose types are arbitrary record types.

**Field access** We now define the typing of expressions of the form `e1.e2` or `e1.e2 or e3` (i.e., the access of a field of a record).

For a record type $\tau = \{x_1 = \tau_1; \cdots; x_n = \tau_n; \_ = \tau_O\}$, we refer to $\tau_O$ by $\text{def}(\tau)$.

We have two sets of rules depending on whether a default value is provided or not.

For the case where a default value is provided, we distinguish the case where the name of the accessed field is statically known from the case where it is not. Making such a distinction does not really make sense if no default value is provided, as when the accessed field is unknown there is no way to ensure that the accessed field indeed exists. We could also here accept some unsoundness and allow this type of access like we do for literal records, but this pattern seems less used in practice so we prefer not to add unnecessary unsoundness.

When the name of the accessed field is unknown, the return type is the union of all the types contained in the record and of the type of the default value, minus the $\nabla$ type as if the field is undefined then the default value is returned instead.

**Record patterns** We also extend the language of patterns to include the record-related ones from the rules of Figure 6. We thus also extend the $\langle p \rangle$ and $^p/_\tau$ operators. For the definition of $\langle p \rangle$, we need to propagate the type informations down the pattern because of the rule stating that a non-annotated variable is given the type `?`. Indeed, having a general rule $\langle p : \tau \rangle = \langle p \rangle \wedge \tau$ would not fit our needs because for example, the accepted type for the pattern `{ x } : { x = Int }` would be `{ x = ? ∧ Int }` while we want it to be `{ x = Int }`. So we need a rule implying that the accepting type of `{ x } : { x = Int }` is the accepted type of `{ x : Int }`. This is what the rules of Figure 14 state. The extension of the $^p/_\tau$ operator is given in Figure 15.

### 3.4 About soundness

The definition of soundness in the presence of gradual types is usually the soundness of an intermediate language in which explicit casts have been added in places where the gradual type is used.

This however only makes sense as far as the gradually typed language is compiled to something else. However in our case, we did not want to do so (for backward compatibility reasons with the original language), so stating such a property would not make sense.

The only notion of safety that we could state is that the language is safe (in the classical sense of "every well-typed expression either reduces to a value of the same type either infinitely reduces") as far as no gradual type is involved. But even this property does not hold because of records (as said in Section 3.3.2, we deliberately allow some unsoundness for the records).

This is however not a real problem as our goal is not to provide an ideal theoretical type-system but rather a practical tool to make the development easier, so perfect soundness is not required.

## 4 Implementation

This type-system has been implemented as an OCaml program. The sources are available at `https://www.github.com/regnat/tix`.

$$\frac{\forall i \in I, \Gamma \vdash^{\Uparrow} e_i : s_i \qquad \forall i \in I, \Gamma \vdash^{\delta} e'_i : \tau_i \qquad \forall (i,i') \in I^2, i \neq i' \Rightarrow s_i \neq s_{i'}}{\Gamma \vdash^{\delta} \left\{\overline{e_i = e'_i;}^{i \in I}\right\} : \{s_i = \tau_i\}} (\textit{RFinite})$$

$$\frac{\forall i \in I, \Gamma \vdash^{\Uparrow} e_i : \tau_i \qquad \forall i \in I, \Gamma \vdash^{\Uparrow} e'_i : \sigma_i \qquad \forall i \in I, \tau_i \leq \text{String}}{\Gamma \vdash^{\Uparrow} \left\{\overline{e_i = e'_i;}^{i \in I}\right\} : \left\{\_ = \left(\bigvee_{i \in I} \sigma_i\right) \vee \nabla\right\}} (\textit{IRInfinite})$$

$$\frac{\forall i \in I, \Gamma \vdash^{\Uparrow} e_i : \tau_i \qquad \forall i \in I, \Gamma \vdash^{\Downarrow} e'_i : \sigma \qquad \forall i \in I, \tau_i \leq \text{String}}{\Gamma \vdash^{\Downarrow} \left\{\overline{e_i = e'_i;}^{i \in I}\right\} : \{\_ = \sigma \vee \nabla\}} (\textit{CRInfinite})$$

$$\frac{\Gamma \vdash^{\Uparrow} e_1 : \tau \qquad \Gamma \vdash^{\Uparrow} e_2 : \sigma \qquad \tau \leq \{..\} \qquad \sigma \leq \{..\}}{\Gamma \vdash^{\delta} e_1 + e_2 : \tau + \sigma} (\textit{RMerge})$$

$$\frac{\Gamma \vdash^{\Uparrow} e_1 : \tau \qquad \Gamma \vdash^{\Uparrow} e_2 : s \qquad \tau \tilde{\leq} \{..\} \qquad \tau(s) \wedge \nabla \leq \text{Empty}}{\Gamma \vdash^{\delta} e_1.e_2 : \tau(s)} (\textit{RAccessFinite})$$

$$\frac{\Gamma \vdash^{\Uparrow} e_1 : \tau \qquad \Gamma \vdash^{\Uparrow} e_2 : s \qquad \Gamma \vdash^{\Uparrow} e_3 : \sigma}{\Gamma \vdash^{\Uparrow} e_1.e_2 \text{ or } e_3 : (\tau(s) \backslash \nabla) \vee \sigma} (\textit{IRGuardedAccessFinite})$$

$$\frac{\Gamma \vdash^{\Uparrow} e_1 : \tau \qquad \Gamma \vdash^{\Uparrow} e_2 : s \qquad \Gamma \vdash^{\Downarrow} e_3 : \sigma \qquad \tau(s) \backslash \nabla \leq \sigma}{\Gamma \vdash^{\Downarrow} e_1.e_2 \text{ or } e_3 : \sigma} (\textit{CRGuardedAccessFinite})$$

$$\frac{\Gamma \vdash^{\Uparrow} e_1 : \tau \qquad \Gamma \vdash^{\Uparrow} e_2 : \tau' \qquad \tau' \leq \text{string} \qquad \Gamma \vdash^{\Uparrow} e_3 : \sigma}{\Gamma \vdash^{\Uparrow} e_1.e_2 \text{ or } e_3 : \left(\text{def}(\tau) \vee \bigvee_{s \in \tilde{\text{Dom}}(\tau)} \tau(s) \backslash \nabla\right) \vee \sigma} (\textit{IRGuardedAccessInfinite})$$

$$\frac{\Gamma \vdash^{\Uparrow} e_1 : \tau \qquad \Gamma \vdash^{\Uparrow} e_2 : \tau' \qquad \tau' \leq \text{string} \qquad \Gamma \vdash^{\Downarrow} e_3 : \sigma \qquad \left(\text{def}(\tau) \vee \bigvee_{s \in \tilde{\text{Dom}}(\tau)} \tau(s) \backslash \nabla\right) \leq \sigma}{\Gamma \vdash^{\Downarrow} e_1.e_2 \text{ or } e_3 : \sigma} (\textit{CRGuardedAccessInfinite})$$

Figure 3: Typing rules for records

The figure Figure 17 shows some runs of the typechecker. Note that the concrete syntax of Nix different from the syntax we gave here: The lambdas are defined with the syntax `<pattern>:<expr>` and the type annotations with the syntax `/*: τ */`[8].

## 4.1 ℂDuce and subtyping

Instead of working our own implementation of types and of the subtyping algorithm, we reused those of the implementation of the ℂDuce language[1].

ℂDuce is a language specialized in the manipulation of XML documents which features a type-system based on the set-theoretic types of Frisch [9] (on which this work is itself based). Although its type-system is obviously not the same as the one presented here, its types form a supertype of ours (with the exception of the gradual type which is absent in ℂDuce but may be encoded), so we could reuse its representation.

We also wanted to reuse its subtyping algorithm (since implementing it would have taken way too much time), but this was more difficult, as it implements a subtyping algorithm in the context of a strict semantic and static types. Fortunately, both the adaptation to a lazy semantic and the extension to gradual subtyping may be expressed simply by an encoding of the types (thus without modifying the algorithm itself).

### 4.1.1 Lazy subtyping

As Castagna and Petrucciani [4] explain, the lazy subtyping algorithm may be reduced to the strict one by an encoding of the types. The idea is to encode directly into the types the differences in the set-theoretic interpretation: instead of giving ourselves a distinguished element $\bot$ of the domain and adding it to some of the sets representing the types, we give ourselves a distinguished type $\bot$ (which has an empty intersection with any other basic type) and rewrite the types as shown in Figure 16

### 4.1.2 Gradual subtyping

The very definition of the gradual subtyping relation as given by Castagna and Lanvin [3] uses the static subtyping relation *via* an encoding of the types. This definition can be expressed as:

$$\tau_1 \tilde{\leq} \tau_2 \text{ if } \tau_1^{\Downarrow} \leq \tau_2^{\Uparrow}$$

Where $\tau^{\Downarrow}$ (resp. $\tau^{\Uparrow}$) is obtained by replacing every covariant occurrence of `?` in $\tau$ by *Any* (resp. *Empty*) and every contravariant occurrence of `?` by *Empty* (resp. *Any*).

## 4.2 Adaptation of the type-system

Here we presented a particular type-system. However several rules may be modified depending on the exact notion of safety we want to achieve.

In particular, we may decide to get rid of the implicit gradual typing (the fact that an unannotated variable in a pattern is given the type `?`), and decide that an unannotated variable will be typed with the type *Any*. This forces the programmer to explicitly annotate the places where he wants the gradual typing to occur (in fact this forces the programmer to annotate almost every variable since the type-system does no unification).

We could even decide to go further by locally disabling gradual typing. In this case, the gradual type could become a new distinguished type constant. We still need to provide a pair of explicit cast functions (a function `fromGrad : ? → Empty` and a function `toGrad : Any → ?`, both implemented as the identity) in order to be able to use gradual values from the outside world.

[8] This has been chosen because it is syntactically a comment in nix, so annotated code can be understood by the original Nix program, which is a requirement of our work

## 4.3 Extensions

The language we studied was only a subset of the actual Nix language.

We present here some parts of the language that have been omitted until now and an informal typing for them.

### 4.3.1 Special operators on lists and records

We quickly talked in Section 2 of the `mapAttrs` function which can not be given a generic enough type by itself but needs to be considered as a special operator with a custom typing rule. Several other functions go into this category. Even the `map` function on lists can not be typed in a useful way with heterogeneous lists.

Here is for example a possible pair of rules fore the `mapAttrs` function with atomic record types (the real rules need of course to be extended to arbitrary record types):

$$\frac{\Gamma \vdash^{\Uparrow} e : \left\{\overline{s_i = \tau_i;}^{i\in I}\right\} \qquad \Gamma \vdash^{\Uparrow} f : \tau_f \qquad \forall i \in I, \tau_f \tilde{\circ} \tau_i = \sigma_i}{\Gamma \vdash^{\Uparrow} \text{mapAttrs}(f, e) : \left\{\overline{s_i = \sigma_i;}^{i\in I}\right\}}$$

$$\frac{\Gamma \vdash^{\Uparrow} e : \left\{\overline{s_i = \tau_i;}^{i\in I}\right\} \qquad \forall i \in I, \Gamma \vdash^{\Downarrow} f : \tau_i \to \sigma_i}{\Gamma \vdash^{\Downarrow} \text{mapAttrs}(f, e) : \left\{\overline{s_i = \sigma_i;}^{i\in I}\right\}}$$

### 4.3.2 Tracking of the predicates on types

In the formalism we present here, we compile the **`if`** constructs differently depending on their form using a hardcoded list of predicates on types. This is not fully satisfactory as this only works at a syntactic-level, and will not even detect some simple modifications such as the aliasing of a predicate. So **`if`** `isInt x` **`then`** `x+1` **`else`** `1` will typecheck (under the hypothesis that `x` is defined of course), while **`let`** `f = isInt;` **`in if`** `f x` **`then`** `x+1` **`else`** `x` will not.

It is possible to get more flexibility by recognizing that the notion of a predicate on a type `t` is a function of type `(t → true) ∧ (¬t → false)`. We can thus modify the Nix-light language by removing the typecase and replacing it with an if construct, and replace the associated typing rules by the following ones:

$$\frac{\begin{array}{c}\Gamma \vdash^{\Uparrow} x : \tau_x \qquad \Gamma \vdash^{\Uparrow} f : (\tau \to \text{true}) \wedge (\neg\tau \to \text{false}) \\ \tau_x \not\leq \neg\tau \Rightarrow \Gamma; x : \tau \wedge \tau_x \vdash^{\Uparrow} e_1 : \sigma_1 \qquad \tau_x \not\leq \tau \Rightarrow \Gamma; x : \neg\tau \wedge \tau_x \vdash^{\Uparrow} e_2 : \sigma_2\end{array}}{\Gamma \vdash^{\Uparrow} \text{if } f\ x \text{ then } e_1 \text{ else } e_2 : \sigma_1 \vee \sigma_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\Uparrow} e_0 : \tau \qquad \tau \not\leq \text{true} \Rightarrow \Gamma \vdash e_1 : \sigma_1 \\ \tau \not\leq \text{false} \Rightarrow \Gamma \vdash e_2 : \sigma_2 \qquad e_0 \text{ not of the form } f\ x \text{ with } f \text{ a predicate on types}\end{array}}{\Gamma \vdash^{\Uparrow} \text{if } f\ x \text{ then } e_1 \text{ else } e_2 : \sigma_1 \vee \sigma_2}$$

The (theoretical) drawback of this approach is that instead of having a clean set of typing rules with one unified rule for the typecase and putting the ad-hoc part (the recognition of some special if constructs) into the preliminary compilation phase, we need to have a new ad-hoc rule for the if-then-else's (so the system is slightly more difficult to understand and modify).

This has however been implemented as it gives much more flexibility to the system.

### 4.3.3 The with construct

Nix accepts expressions of the form `with <expr>; <expr>`. The meaning of this is that, provided that the first expression evaluates to a record `{ x1 = e1; ⋯; xn = en }`, then the second one is evaluated with the content of the record in scope, which is with the new variables `x1`, …, `xn` available with value respectivly `e1`, …, `en`. Moreover, if a variable is already in the scope, then it can not be shadowed by a `with` construct.

Given its weird semantic and the difficulty to type it (greatly improved by the high versatility of the records in Nix), this construct is not presented at all here. It should be possible however to type it in some simple enough contexts.

### 4.3.4 Throw/assert and tryEval

Nix has a notion of exceptions (although their exact nature is not documented at all). In particular, ther exists a `throw` function (which raises an exception with a given message) and an `assert` construct (of the form `assert <expr>; <expr>`) which exists with an error if the first expression does not evaluates to *`true`* and evaluates the second one otherwise. The `throw` function can be directly compiled to a function of type *`String → Any`* and `assert e1; e2`; can be compiled to `if e1 then e2 else throw "assertion failed`.

The `tryEval` function implements a form of exception catching: if its argument raises an exception, it evaluates to `{ success = false; value = false; }`. Otherwise, if its arguments evaluates to a value `v`, it evaluates to `{ success = true; value = v; }`. As long as we do not try to track exceptions, a reasonable rule for this is:

$$\frac{\Gamma \vdash^{\delta} e : \tau}{\Gamma \vdash^{\delta} \text{tryEval}(e) : \{\text{success} = \text{Bool}; \text{value} = \text{false} \vee \tau\}}$$

(note that in presence of polymorphism, it would have been enough to type `tryEval` as a function of type `∀α. α → { success = Bool; value = α ∨ false}`).

## Conclusion

The type-system we designed satisfies our practical requirements. Is is powerful and flexible enough to adapt itself to the flexibility of the Nix language, and at the same time can give useful results with a reasonable amount of annotations.

On a more theoretical aspects, this has been a successful experiment in mixing together several type-checking techniques. We have been pleased to see that most of them can be used together without any major limitation (except of course the necessary abandon of polymorphism because of the gradual typing). In particular, the development of the bidirectional typing for set-theoretic types improves their expressiveness and reduces the amount of requested annotations, which is a huge gain. We hope that this work will also prove the usability of these techniques in a real-world setup.

As explained in the synthesis sheet, the next steps are on one hand to continue the work on the implementation in order to make it more usable in an enterprise context and on the other hand to develop the type-system further, in particular to add polymorphism to it.

## References


[1] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. “CDuce: An XML-Centric General-Purpose Language”. In: *Proceedings of the ACM International Conference on Functional Programming*. 2003.

[2] Giuseppe Castagna. *Covariance and Contravariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers)*. Unpublished manuscript. 2015.

[3] Giuseppe Castagna and Victor Lanvin. “Gradual Typing with Union and Intersection Types”. In: *ICFP ’17, 22nd ACM SIGPLAN International Conference on Functional Programming*. Sept. 2017.

[4] Giuseppe Castagna and Tommaso Petrucciani. *Call-by-need semantic subtyping*. Unpublished manuscript. 2017.

[5] Giuseppe Castagna et al. *Set-Theoretic Foundation for Polymorphic Gradual Typing*. Unpublished manuscript. 2017.

[6] Eelco Dolstra. *Issue #14 : Static type system*. `https://github.com/NixOS/nix/issues/14`. 2002.

[7] Eelco Dolstra. “The Purely Functional Software Deployment Model”. PhD thesis. Faculty of Science, Utrecht, The Netherlands., Jan. 2006.

[8] Joshua Dunfield and Neelakantan R. Krishnaswami. “Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism”. In: *Int’l Conf. Functional Programming*. `arXiv:1306.6032[cs.PL]`. Sept. 2013.

[9] Alain Frisch. “Théorie, conception et réalisation d’un langage adapté à XML”. PhD thesis. Université Paris Diderot, Dec. 2004.

[10] Haruo Hosoya et al. “Local type inference”. In: *In Conference Record of POPL ’98: the 25th ACM SIGPLAN-SIGACT Symposium on Princaples of Programming Languages*. 1998.

[11] Kim Nguyễn. “Langage de Combinateurs pour XML: Conception, Typage, Réalisation”. PhD thesis. Université Paris-Sud 11, 2008. URL: `files/thesis.pdf`.

[12] Jeremy G. Siek and Walid Taha. “Gradual typing for functional languages”. In: *Scheme and Functional Programming Workshop*. Sept. 2006.

[13] Tobin-Hochstad et al. “The Design and Implementation of Typed Scheme”. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’08. San Francisco, California, USA: ACM, 2008, pp. 395–406. ISBN: 978-1-59593-689-9. DOI: `10.1145/1328438.1328486`. URL: `http://doi.acm.org/10.1145/1328438.1328486`.

## Appendix

```
<expr> ::=
  <ident> | <constant>
  | λ <pattern>.<expr> | <expr> <expr>
  | let <var–pattern> = <expr>; ⋯; <var–pattern> = <expr>; in <expr>
  | [ <expr> ⋯ <expr> ]
  | { <record–field>; ⋯; <record–field>; }
  | rec { <record–field>; ⋯; <record–field>; }
  | if <expr> then <expr> else <expr>
  | <expr>.<acces–path>
  | <expr>.<acces–path> or <expr>
  | <expr> <infix–op> <expr>
  |  <expr> : <τ>

<constant> ::= <string> | <integer> | <boolean> | <paths>

<record–field> ::= inherit <ident> ⋯ <ident>
  | inherit (<expr>) <ident> <ident>
  | <access–path> = <expr> |  <access-path> : <τ> = <expr>

<pattern> ::= <record–pattern> | <record–pattern>@<ident>
  | <var–pattern>

<var–pattern> ::= <ident> |  <ident> : <τ>

<record–pattern> ::=
  | { <record–pattern–field>, ⋯, <record–pattern–field> }
  | { <record–pattern–field>, ⋯, <record–pattern–field>, ... }

<record–pattern–field> ::= <var–pattern> | <var–pattern> ? <expr>

<access–path> ::= <access–path–item>. ⋯ . <access–path–item>

<access–path–item> ::= <ident> | { <expr> }

<infix–op> ::= + | – | * | / | // | ++ | ⋯
```

Figure 4: Syntax of the Nix language

```
<basetype> ::= Bool | Int | String | Any | Empty

<t> ::= <constant> | <t> → <t>
  | <t> ∨ <t> | <t> ∧ <t> | ¬ <t>
  | [<R>]
  | { <ident> = <t>; ⋯; <ident> = <t>; _ = <t> }
  | <basetype>

<R> ::= <t> | <R>+ | <R>* | <R>?
  | <R> <R> | <R> ¦ <R>

<τ> ::= <t> | <τ> → <τ>
  | <τ> ∨ <τ> | <τ> ∧ <τ>
  | [<ρ>] | [<ρ> ?? ]
  | { <ident> = <τ>; ⋯; <ident> = <τ>; _ = <τ> }
  | ?

<ρ> ::= <τ> | <ρ>+ | <ρ>* | <ρ>?
  | <ρ> <ρ> | <ρ> ¦ <ρ>
```

Figure 5: Syntax of Nix types

```
<expr> ::= <ident> | <constant>
  | <expr>.<expr> | <expr>.<expr> or <expr>
  | λ <pattern>.<expr> | <expr> <expr>
  | let <var–pattern> = <expr>; ⋯; <var–pattern> = <expr>; in <expr>
  | Cons (<expr>, <expr>)
  | { <ident> = <expr>; ...; <ident> = <expr> }
  | (<ident> = <expr> ∈ <t̂>) ? <expr> : <expr>
  | <operator>
  | <expr>:<τ>

<constant> ::= <string> | <int> | <bool> | Nil

<operator> ::=
  | <expr> <infix–op> <expr>
  | Head(<expr>) | Tail(<expr>)

<infix–op> ::= + | − | * | / | // | ++ | ⋯

<pattern> ::= <record–pattern> | <record–pattern>@<ident>
  | <var–pattern>

<record–pattern> ::= <record–pattern>:τ
  | { <record–pattern–field>, ⋯, <record–pattern–field> }
  | { <record–pattern–field>, ⋯, <record–pattern–field>, … }

<record–pattern–field> ::= <var–pattern> | <var–pattern> ? <constant>

<var–pattern> ::= <ident> | <ident>:<τ>

<basetype> ::= Bool | Int | String | Any | Empty

<t> ::= <constant> | <basetype>
  | <t> ∨ <t> | <t> ∧ <t> | ¬ <t>
  | <t> → <t>
  | Cons(<t>, <t>) | let <ident> = <t>; ⋯; <ident> = <t> in <t>
  | { <ident> = <t>; ⋯; <ident> = <t>; _ = <t> }

<τ> ::= <constant> | <basetype> | <t> | ?
  | <τ> ∨ <τ> | <τ> ∧ <τ>
  | <τ> → <τ>
  | Cons(<τ>, <τ>) | let <ident> = <τ>; ⋯; <ident> = <τ> in <τ>
  | { <ident> = <τ>; ⋯; <ident> = <τ>; _ = <τ> }

<t̂> ::= <constant> | <basetype>
  | <t̂> ∨ <t̂> | <t̂> ∧ <t̂> | ¬ <t̂>
  | Empty → Any
  | Cons(Any, Any)
  | { <ident> = Any; ⋯; <ident> = Any; _ = Any∨∇ }
```

Figure 6: Nix-light grammar for expressions

```
<value> ::=
  | <constant>
  | Cons(<expr>, <expr>)
  | { <ident> = <expr>; ⋯; <ident> = <expr>; }
  | λ<pattern>.<expr>
```

Figure 7: Nix-light grammar for values

$$
\begin{aligned}
{}^{x}/_{e} &= x := e \\
{}^{p:\tau}/_{e} &= {}^{p}/_{e} \\
{}^{q@x}/_{v} &= x := v;\ {}^{q}/_{v} \\
{}^{\{..\}}/_{\{\cdots\}} &= \emptyset \\
{}^{\{\}}/_{\{\}} &= \emptyset \\
{}^{\{l,\ \overline{l_i}^{\,i\in I}\}}/_{\{x = e;\ \overline{x_j = e_j}^{\,j\in J}\}} &= {}^{l}/_{e};\ {}^{\{\overline{l_i}^{\,i\in I}\}}/_{\{\overline{x_j = e_j}^{\,j\in J}\}} \quad \text{if } x = \mathscr{V}(l) \\
{}^{\{l,\ \overline{l_i}^{\,i\in I},\ l_m,\ ..\}}/_{\{x = e;\ \overline{x_j = e_j}^{\,j\in J}\}} &= {}^{l}/_{e};\ {}^{\{\overline{l_i}^{\,i\in I},\ ..\}}/_{\{\overline{x_j = e_j}^{\,j\in J}\}} \quad \text{if } x = \mathscr{V}(l) \\
{}^{\{r?c,\ \overline{l_i}^{\,i\in I}\}}/_{\{\overline{x_j = e_j}^{\,j\in J}\}} &= {}^{r}/_{c};\ {}^{\{\overline{l_i}^{\,i\in I}\}}/_{\{\overline{x_j = e_j}^{\,j\in J}\}} \quad \text{if } \forall j \in J, x_j \neq \mathscr{V}(r) \\
{}^{\{r?c,\ \overline{l_i}^{\,i\in I},\ ..\}}/_{\{\overline{x_j = e_j}^{\,j\in J}\}} &= {}^{r}/_{c};\ {}^{\{\overline{l_i}^{\,i\in I},\ ..\}}/_{\{\overline{x_j = e_j}^{\,j\in J}\}} \quad \text{if } \forall j \in J, x_j \neq \mathscr{V}(r)
\end{aligned}
$$

Figure 8: Semantic of the pattern-matching in Nix-light

$$
\begin{aligned}
(\lambda x.e_1)\ e_2 &\rightsquigarrow e_1[x := e_2] \\
(\lambda p.e)\ v &\rightsquigarrow e[{}^{p}/_{v}] \\
(x = v \in t)\ ?\ e_1 : e_2 &\rightsquigarrow e_1[x := v] \quad \text{if } \vdash v : t \\
(x = v \in t)\ ?\ e_1 : e_2 &\rightsquigarrow e_2[x := v] \quad \text{if } \vdash v : \neg t \\
\{\ x = e;\ \cdots\ \}.x &\rightsquigarrow e \\
\{\ x = e;\ \cdots\ \}.x\ \mathbf{or}\ e' &\rightsquigarrow e \\
\{\ x_1 = e_1;\ \cdots;\ x_n = e_n\ \}.x\ \mathbf{or}\ e' &\rightsquigarrow e' \quad \text{if } x \notin \{x_1, \cdots, x_n\} \\
\mathtt{Head}(\mathbf{Cons}(e_1,\ e_2)) &\rightsquigarrow e_1 \\
\mathtt{Tail}(\mathbf{Cons}(e_1,\ e_2)) &\rightsquigarrow e_2 \\
e : \tau &\rightsquigarrow e
\end{aligned}
$$

Figure 9: Nix-light operational semantics

$$
\mathrm{flatten}\left(\left\{\overline{\mathsf{apf}_i = e_i;}^{\,i\in I}\ \overline{\mathsf{apf}_j.\mathsf{ap}_k = e_k}^{\,k\in K_j;\ j\in J}\right\}\right) = \left\{\overline{\mathsf{apf}_i = e_i;}^{\,i\in I}\ \overline{\mathsf{apf}_j = \mathrm{flatten}\left(\left\{\overline{\mathsf{ap}_k = e_k;}^{\,k\in K_j}\right\}\right)}^{\,j\in J}\right\}
$$

Where the apf's denote constructs of the form `<access-path-field>` and the ap's constructs of the form `<access-path>`. We furthermore assume that the $\mathsf{apf}_i$'s and $\mathsf{apf}_j$'s are pairwise distinct (we consider that two access paths fields of the form `${e}` are always distinct).

Figure 10: Definition of the flatten function on records

$$
\begin{array}{ll}
[\![ \texttt{x} ]\!] & = \texttt{x} \\
[\![ \texttt{c} ]\!] & = \texttt{c} \\
[\![ \texttt{λp.e} ]\!] & = \texttt{λ}[\![\texttt{p}]\!].[\![\texttt{e}]\!] \\
[\![ \texttt{e}_1\ \texttt{e}_2 ]\!] & = [\![\texttt{e}_1]\!]\ [\![\texttt{e}_2]\!] \\
[\![ \texttt{let } \texttt{v}_1 = \texttt{e}_1\texttt{; } \cdots\texttt{; } \texttt{v}_n = \texttt{e}_n\texttt{; in e} ]\!] & = \texttt{let } \texttt{v}_1 = [\![\texttt{e}_1]\!]\texttt{; } \cdots\texttt{; } \texttt{v}_n = [\![\texttt{e}_n]\!]\texttt{; in } [\![\texttt{e}]\!] \\
[\![ \texttt{[ e}_1\ \cdots\ \texttt{e}_n\ \texttt{]} ]\!] & = \texttt{[}[\![\texttt{e}_1]\!]\ \cdots\ [\![\texttt{e}_n]\!]\texttt{]} \\
[\![ \texttt{\{ ap}_1\ \texttt{= e}_1\texttt{; ...; ap}_n\ \texttt{= e}_n\ \texttt{\}} ]\!] & = \text{flatten}(\texttt{\{ } [\![\texttt{ap}_1]\!]\ \texttt{=} [\![\texttt{e}_1]\!]\texttt{; ...; } [\![\texttt{ap}_n]\!]\ \texttt{=} [\![\texttt{e}_n]\!]\ \texttt{\}}) \\
[\![ \texttt{rec \{ record \}} ]\!] & = \text{derec}([\![\texttt{\{ record \}}]\!]) \\
[\![ \texttt{if isT x then e}_1\ \texttt{else e}_2 ]\!] & = \texttt{(x = x } \in \texttt{ T) ? } [\![\texttt{e}_1]\!]\texttt{ : } [\![\texttt{e}_2]\!] \\
[\![ \texttt{if e}_0\ \texttt{then e}_1\ \texttt{else e}_2 ]\!] & = \texttt{(x = (}[\![\texttt{e}_0]\!]\texttt{ : } \mathit{Bool}\texttt{) } \in \mathit{true}\texttt{) ? } [\![\texttt{e}_1]\!]\texttt{ : } [\![\texttt{e}_2]\!] \\
 & \textit{if } \texttt{e}_0 \textit{ is not of the form } \texttt{isT y}\textit{; with } \texttt{x} \textit{ a fresh variable} \\
[\![ \texttt{e.apf}_1\texttt{. } \cdots\texttt{ . apf}_n ]\!] & = [\![\texttt{e}]\!]\texttt{. apf}_1\texttt{ . } \cdots\texttt{ . apf}_n \\
[\![ \texttt{e}_1\texttt{.ap or e}_2 ]\!] & = [\![\texttt{e}_1\texttt{.ap}]\!]\ \texttt{or}\ [\![\texttt{e}_2]\!] \\
[\![ \texttt{e}_1\ \texttt{op e}_2 ]\!] & = [\![\texttt{e}_1]\!]\ \texttt{op}\ [\![\texttt{e}_2]\!] \\
[\![ \texttt{e : τ} ]\!] & = [\![\texttt{e}]\!]\texttt{ : } [\![\texttt{τ}]\!] \\
\texttt{\$\{e\}} & = [\![\texttt{e}]\!] \\
\texttt{x} & = \texttt{"x"}
\end{array}
$$

Figure 11: Compilation from Nix to Nix-light

$$
\begin{array}{ll}
\texttt{if e}_1\texttt{|e}_2\ \texttt{then e}_{\text{if}}\ \texttt{else e}_{\text{then}} & \to \texttt{if e}_1\ \texttt{then e}_{\text{if}}\ \texttt{else (if e}_2\ \texttt{then e}_{\text{if}}\ \texttt{else e}_{\text{then}}\texttt{)} \\
\texttt{if e}_1\texttt{\&e}_2\ \texttt{then e}_{\text{if}}\ \texttt{else e}_{\text{then}} & \to \texttt{if e}_1\ \texttt{then (if e}_2\ \texttt{then e}_{\text{if}}\ \texttt{else e}_{\text{then}}\texttt{) else e}_{\text{if}} \\
\texttt{if not e then e}_{\text{if}}\ \texttt{else e}_{\text{then}} & \to \texttt{if e then e}_{\text{then}}\ \texttt{else e}_{\text{if}}
\end{array}
$$

Figure 12: Preprocessing of Nix conditionals

$$
\frac{}{\vdash \texttt{c} : \mathscr{B}(\texttt{c})} \qquad \frac{}{\vdash \lambda \texttt{x.e} : \text{Empty} \to \text{Any}}
$$

$$
\frac{}{\vdash \{\texttt{x}_1 = \texttt{e}_1; \cdots; \texttt{x}_n = \texttt{e}_n;\} : \{\texttt{x}_1 = \text{Any}; \cdots; \texttt{x}_n = \text{Any};\}} \qquad \frac{}{\vdash \text{Cons}(\texttt{e}_1, \texttt{e}_2) : \text{Cons}(\text{Any}, \text{Any})}
$$

Figure 13: Typing judgement of the typecase

$$
\begin{aligned}
\langle \texttt{r?c} \rangle &= \langle \texttt{r} \rangle \\
\left\langle \left\{ \overline{\texttt{l}_i,}^{\,i \in I} \right\} \right\rangle &= \left\{ \overline{\mathscr{V}(\texttt{l}_i) = \langle \texttt{l}_i \rangle}^{\,i \in I} \right\} \\
\left\langle \left\{ \overline{\texttt{l}_i}^{\,i \in I} \right\} : \left\{ \overline{\texttt{x}_i = \tau_i}^{\,i \in I} \right\} \right\rangle &= \left\langle \left\{ \overline{\texttt{l}_i : \tau_i}^{\,i \in I} \right\} \right\rangle
\end{aligned}
$$

Figure 14: Extension of the $\langle \cdot \rangle$ operator for record patterns

$$
\begin{aligned}
\tau/_{x?c} &= x : ? \vee \mathscr{B}(c) \\
\tau/_{x\,:\,\tau?c} &= x : \tau \vee \mathscr{B}(c) \\
\{\overline{x_i = \tau_i}^{i\in I}\}/\{\overline{l_i}^{i\in I}, \overline{r_j ?c_j}^{j\in J}\} &= \overline{\tau_i/_{l_i}}^{i\in I} && \text{if } \forall i \in I, x_i = \mathscr{V}(l_i) \\
\{\overline{x_i = \tau_i}^{i\in I}\}/\{\overline{l_i}^{i\in I}, \overline{r_j ?c_j}^{j\in J}, ..\} &= \overline{\tau_i/_{l_i}}^{i\in I} && \text{if } \forall i \in I, x_i = \mathscr{V}(l_i)
\end{aligned}
$$

Figure 15: Extension of the $p/_\tau$ operator for record patterns

$$
\begin{aligned}
[\![b]\!] &= b \\
[\![c]\!] &= c \\
[\![\text{Empty}]\!] &= \text{Empty} \\
[\![\text{Any}]\!] &= \text{Any} \\
[\![\text{Cons}(\tau, \sigma)]\!] &= \text{Cons}([\![\tau]\!] \vee \bot, [\![\sigma]\!] \vee \bot) \\
[\![\tau \to \sigma]\!] &= ([\![\tau]\!] \vee \bot) \to \sigma \\
[\![\tau \vee \sigma]\!] &= [\![\tau]\!] \vee [\![\sigma]\!] \\
[\![\tau \wedge \sigma]\!] &= [\![\tau]\!] \wedge [\![\sigma]\!] \\
[\![\neg t]\!] &= \neg([\![t]\!] \vee \bot) \\
[\![\{x_1 = \tau_1; \cdots; x_n = \tau_n; \_ = \tau\}]\!] &= \{x_1 = [\![\tau_1]\!] \vee \bot; \cdots; x_n = [\![\tau_n]\!] \vee \bot; \_ = [\![\tau]\!] \vee \bot\}
\end{aligned}
$$

Figure 16: Rewriting of lazy types into strict ones

```
» cat /tmp/test.nix
let
  f /*: (true → Int → Int) & (false → Bool → Bool) */ = cond: x:
  if cond then x+1 else __not x;
in f
» ./_build/install/default/bin/tix /tmp/test.nix
(`false → Bool → Bool) & (`true → Int → Int)

» cat examples/record_pattern.nix
let
  f = { x /*: Bool */, y}:
  if x then
    y
    else x;
in f
» ./_build/install/default/bin/tix examples/record_pattern.nix
{ x=Bool y='? } → Bool | '?

» cat examples/data_structures.nix
let
  r = { x = true; y = r; z = [ 1 2 r ]; };
in r
» ./_build/install/default/bin/tix examples/data_structures.nix
{ x=`true y='? z=[ 1 2 '? ] }

» cat /tmp/test.nix
let
  f = { x /*: Bool */, y /*: Int */}:
  if x then
    y
    else x;
  r = { y = 1; };
in f r
» ./_build/install/default/bin/tix /tmp/test.nix
error: This expression has type { y=1 } while a subtype of { x=Bool
   y=Int } was expected at file /tmp/test.nix, line 7, character 6
```

Figure 17: examples of run of the typechecker