SimpleNix: Type Inference for Nix

Bachelor Thesis - Sebastian Klähn
sebastian.klaehn@merkur.uni-freiburg.de

Examiner: Prof. Dr. Peter Thiemann
Advisor: Prof. Dr. Peter Thiemann

Abstract

Nix is a cross-platform package manager for Linux and MacOS focusing on reproducibility and security. Unlike other package managers, Nix uses a functional programming language to build packages, configure systems, and perform all kinds of meta programming. Even though Nix and NixOS have found great adoption in recent years, the Nix language still lacks common developer support in language servers, documentation, and type inference, making it a burden to write even simple scripts. My contribution is a parser for Nix written in Rust and a language server that provides type inference for the language.

Department of Computer Science University Freiburg

Contents

1. Motivation

The Nix package manager sits on top of a staggering package repository of over 80.000 user-contributed packages and can be used as a drop-in replacement for most modern package managers like cargo, yarn, PNPM, and many more. It not only provides the dependencies needed to run a local program but also virtually every other development tool, such as language servers, IDEs, built tools, linters, and formatters, making it the all-in-one solution for every dependency. The great distinction to other package managers and its greatest strength is the underlying Nix language upon which the whole ecosystem is built. Having a programming language at its core makes the package manager incredibly powerful, but it comes at the cost of a steep learning curve. New developers not only have to learn a new programming language, but they also have to remember hundreds of configuration fields and the quirks of many built-in and common helper functions from the standard library. A language server with good type inference should make it easier to write syntactically and logically correct programs while reducing the time needed to do so.

For the 20 years that Nix has existed, the problem of missing type inference and a language server has not been solved. Only recently have some projects popped up that want to improve the developer experience of Nix. The latest addition (2022) is Oxcalica's language server Nil that adds simple type inference with a Hindley-Milner style approach adopted for sets, but also some useful code actions like `jump to source` and `dead code analysis`. In this paper, I introduce SimpleNix, a type-inference algorithm based upon the great works of Lionel Parreaux [1] that infers type information using Hindley-Milner as a base and adds subtyping on top. The resulting algorithm supports polymorphism, subtyping, global type inference, and principal types, forming a much more expressive type system than the one used by Oxalica.

Nixd is another new project (2023) that combines a language evaluator and type inference. Unlike Nil, it is dynamic and not static in that it lazily loads set fields. This shows a way to tackle the problem of huge evaluation trees inherent to the Nix package repository that all type inference algorithms have to face.

Another important aspect of developer experience is documentation, and Nix has very bad documentation. There is no agreed-upon style guide for documentation, which led to vastly different documentation styles and many undocumented functions. Developers must refer to the actual source code of functions to learn about what they are doing. That said, there have also been great efforts to improve documentation recently. Mid 2023, RFC 145 was approved, which adds documentation comments that allow one to distinguish between outwards and inwards facing documentation. This creates the opportunity to greatly improve tools like Noogle, that can be used to search for function documentation. With documentation comments, type annotations are the next logical step, and an RFC already exists for this. The unofficial syntax definition at https://typednix.dev/ provides simple HM-style type definitions similar to the one used by SimpleSub [1].

NixOs [2] and Home-manager are both projects that enable users to manage their environments with configuration files. Home Manager can configure the locally installed programs and their configuration, while NixOs goes even further and allows users to create their entire operating system, including partitions, secure boot, window managers, journaling system, and other core components from a single configuration file. The root configuration file provides a function taking dependencies as input and outputs the user configuration. This configuration is a record (key-value mapping) that the executor then uses to determine what to build. To improve developer experience, both projects have introduced *local types*, with the convention that the `__type` field contains the type name as a string. While this does not provide any inference, it at least helps users add the proper fields in their configuration. The provided types can be parsed and converted to internal SimpleNix types, but this is left as an exercise for the reader :-)

2. The Nix Language

The Nix language is a declarative, pure, functional, lazy, and dynamically typed language that is very domain-specific for its use case. Declarativeness stems from the execution model of Nix, where dependencies between files are tracked through paths. These dependencies are used to create a built plan, which is then executed to create the final package. All imports are cached as derivations in the global Nix store, so other packages with the same dependencies can reuse them. To achieve perfect reproducibility, the language tries to remain pure, not allowing side effects in functions. This way, the same derivation executed on two different machines creates the same output, upholding Nix's reproducibility guarantee. It is possible to lift this restriction only for specific use cases to give more freedom in package builds. The reproducibility constraint is then passed to the reviewer, who finally accepts code changes. Because all 80.000 packages are part of one big package tree, laziness is another fundamental property of the Nix language. Without it, even building a single package would take ages, as all other packages would have to be evaluated as well. The problem of huge evaluation trees is also a problem that language servers and type inference algorithms face, as they also can't evaluate the whole package tree. What follows is an overview of the language syntax based on the official documentation at https://nixos.org/manual/nix/stable/language/ but extended with more examples and peculiarities.

2.1. Primitives

The Nix language supports all primitive types well-known from other languages, such as Bool, String, Int, and Float. It also adds the domain-specific type `Path` as paths are used to reference files in configurations and for tracking dependencies. The Nix package managers can create appropriate errors for non-existing paths instead of generic lookup errors and create the appropriate store paths, which would not be possible with ordinary strings. From a typing perspective, adding another primitive is trivial.

2.1.1. Operators

Nix supports all common arithmetic, logic, and comparison operators, which are further discussed in Section 4.2.4.

2.2. Language Constructs

Nix supports many common functional programming language constructs, like let-bindings, first-class functions, and conditionals. It also adds its own domain-specific language constructs, like `with` Section 2.2.4 and `inherit` Section 2.2.5 statements, as well as utilities for functions, making it much more expressive than the simple language MLSub discussed in [1] and [3].

2.2.1. AttrSets

AttrSets (records) are the most important language constructs in the Nix language, as they create all configuration files. AttrSets sets are enclosed in braces and can contain arbitrarily many statements of the form `name = expr;` where the computed value of `expr` is bound to `name`. The *rec* keyword can mark a set as *recursive*, allowing fields to reference other fields defined in the set.

```
{
  # A set with two fields
  set1 = { name = "john"; surname = "smith"; };

  # A recursive set that references itself in the definition
  set2 = rec { age = 35; age2 = age; };

  # A recursive set with primitive recursion
  set3 = rec { x = { y = x;}};

  # A recursive set with illegal mutual recursion.
  # This restriction is only enforced at run-time because of Nix's lazy nature.
  illegal_set = rec { x = y; y = x; };
}
```

Listing 1: AttrSets in Nix

2.2.2. Functions

Function definitions consist of a pattern followed by a double colon (:) and a final function body. A pattern can be a single identifier like `context` or a destructuring set-pattern { x, y }, that expects the function to be called with an AttrSet consisting of the two fields `x` and `y`. Calling a function with an AttrSet that has more than the specified fields is forbidden unless the any-pattern (ellipsis) is given. Deeply nested set-patterns like `{x, {y, {z}}}` are also not allowed. With currying style, it is easily possible to create functions that take more than one argument.

```
{
  # The trivial id function
  fun1 = x: x;

  # A simple function taking two arguments and returning the sum
  fun2 = x: y: x + y;

  # A function that expects one AttrSet with two fields
  fun3 = {x, y}: x + y;

  # A function that expects one AttrSet with _at least_ the two specified fields.
  fun4 = {x, y, ...}: x + y;

  # Nested set patterns are not allowed
  fun5 = {x, {y}}: y;
}
```

Listing 2: Functions in Nix

A function taking more than one argument can be partially applied, leaving the other fields unspecified and available for later use.

```
let out = map (concat "foo") [ "bar" "bla" "abc" ] in
  assert out == [ "foobar" "foobla" "fooabc" ]; {}
```

When a destructuring set pattern is used as a function argument, the whole set can be bound to an identifier using the `@` symbol.

```
let
  # A function that expects a set to have two fields and
  # returns the whole set
  fun = {name, surname} @ person: person;
  out = fun {name = "James"; surname = "Bond";};
```

```
in
  assert out.name && out.surname; {}
```

It is also possible to add default values to set patterns in case the given set does not contain the needed values:

```
let fun = {x ? 120, y}: x + y; in
  assert fun {y = 12} == 132; {}
```

2.2.3. Let Bindings

Let bindings can be used to introduce new named variables accessible in the body of the let binding. A let-binding starts with the `let` keyword and is followed by a finite number of assignments. An assignment is of the form `var = expr;` where `expr` is an arbitrary expression that reduces to a value which is then bound to the name `var`. All defined values are available in other assignments as well, allowing self-reference structs like these: `let x = {y = x;} in {inherit x;}` where any number of `.y` accesses is allowed and produces the same output `{y = {...};}`.

```
# A Simple let binding
let x = 1; in x;

# A Let binding with two bound variables and a none-primitive body.
let x = 1; y = x + 1; in y + 2;

# A let binding with mutual referencing bindings
let x = "Max"; y = x + " Mustermann"; in { concat = "Hello" + x;};

# A self-referencing let binding.
let x = {y = x;} in { inherit x; };

# x.y = {y = {...};}
# x.y.y.y = {y = {...};}
```

Listing 3: let-binding

2.2.4. With

With-statements can precede any expression and introduce all fields of the given AttrSet in the following body. This is a utility construct to reduce repetition in cases where many fields from an AttrSet are needed. When specifying the packages for NixOS or Home-manager, it is not uncommon to prefix the list with `with pkgs;` as the package list is very long most of the time.

```
{
  # Making all fields from pkgs (80.000 elements) available
  packages1 = [ pkgs.code pkgs.fz pkgs.git ];
  packages2 = with pkgs; [code fz git];

  # Introducing a name directly from a set
  with1 = with { name = "John"; }; name;
}
```

Listing 4: with-statement

2.2.5. Inherit

Inherits statements are syntactic sugar to reintroduce known names into let-bindings or AttrSets. Using an inherit statement is essentially the same as re-declaring a variable `x = x;` but it comes in handy for sets and records. During configuration, it is oftentimes important to reexport a lot of expressions, maybe from a deeply nested record. Inherit bindings allow for the specification of a record from which names should be imported, making it easy to reintroduce a lot of bindings quickly. The

same applies to let-bindings, where it is oftentimes needed to import auxiliary functions from outer scopes.

```
# A simple inherit statement
let
  set1 = { y = 1; };
  set2 = { inherit set1; };
in {};
# Is equivalent to
let
  set1 = { y = 1; };
  set2 = { set1 = set1; };
in {};

# Inherit in a let-binding
let
  x = { name = "john"; surname = "smith";};
in
  let
    inherit (x) name surname;
    full_name = name + surname;
  in
    full_name;
# Is equivalent to
let x = { name = "john"; surname = "smith";}; in
  let name = x.name; surname = x.surname;
  full_name = name + surname;
  in
    full_name;

# Inherit using a base-path
let x = { y = { z = 1; };}; in {
  inherit (x.y) z;
};
# This is equivalent to
let x = { y = { z = 1; };}; in {
  z = x.y.z;
}
```

Listing 5: Usage of the inherit statement

2.2.6. String Interpolation

String interpolation is used to insert the evaluated content of any expression into strings and paths. It can also be used to dynamically access or mutate AttrSet fields.

```
{
  # String interpolation for paths
  path = ./home/${user}/.config/${program}/config.toml;

  # String interpolation for strings
  string = "Toms surname is ${surname}";

  # String interpolation for records
  attrset = { ${field} = value;};
  name = { name = "John"; surname = "Smith"; }.${"name"};
}
```

Listing 6: String interpolation

3. Parser

Part of my contribution is a parser for the previously defined nix language written in Rust, which is available as part of the mono repo at https://github.com/Septias/garnix.git. The parser is written with the combinator style crate `nom` and uses Pratt Parsing to parse expressions with different operator precedence.

4. Inference

The Nix language was created as a configuration language to build packages and configure environments easily. Static type inference was not intended for the language, leaving it with many suboptimal rules for type inference. For such languages that do not have type inference built in, flow analysis is an inference method that works well and has already been applied successfully for languages like JavaScript with its superset Typescript. Gradual typing algorithms that only partially infer types analyze the flow of programs and create constraints based on the usage of functions and identifiers. For example, a function execution `f 1`, which applies `1` to the function `f`, leads to a constrain of f `f: int -> a` degrading the polymorphic function type `a -> a` to subtype that only allows arguments of type `int` to be supplied to the function. Lionel Parreaux, in `The Simple Essence of Algebraic Subtyping` [1], created such a type system for the simple programming language MLSub that creates subtyping constraints during program analysis and infers type information with it.

What follows is a brief introduction to the inference algorithm SimpleSub behind MLSub [1] and its main properties. The inference algorithm for the Nix language (SimpleNix) uses this algorithm and extends it where necessary to account for more complex types, different syntax, and other liberations.

4.1. An Introduction to Algebraic Subtyping

Simple-Sub [1], as introduced by Lionel Parreaux, is a practically implemented algorithm based upon the work of Dolan, Stephen, Mycroft, and Alan [3], but simplified and with explicit code examples to make it accessible for people with the less theoretical background. Instead of arguing about types in terms of sets, creating a distributive lattice, and simplifying terms with bi-unification, SimpleSub has a practical implementation in scalar at its root. The implementation is similar to Hindley-Milner style inference and upholds its properties in that it does not need type annotations and can always infer the principal type of an expression. This means it can always find the most general type of which all other possible types are instances, without needing hints from the programmer. The main improvement to HM is that it adds subtyping polymorphism to the language while still upholding these strong properties.

Both Dolan [3] and Parreaux [1] discuss the simple MLSub language in their paper, whose type and syntax definition are given in Figure 1. The syntax specification extends the known λ-calculus rules for functions and applications with records, field accesses, and let bindings. Although it is not imminent from the typing rules, the `rec` keyword that marks a let binding as recursive is optional. The type definition contains the usual λ-calculus types and extends them with SimpleSub-specific types needed for subtyping. $\top$ and $\bot$ are the unit and empty type, the union and intersection type $\tau \sqcup \tau$ and $\tau \sqcap \tau$ as well as the recursive type $\mu\alpha.\tau$ are further discussed in Section 4.1.2 and Section 4.1.3.

$$t := x \mid \lambda x.t \mid t\ t \mid \{l_0 = t; ...; l_n = t\} \mid t.l \mid \text{let rec } x = t \text{ in } t$$

$$\tau ::= \text{primitive} \mid \tau \to \tau \mid \{l_0 : \tau; ...; l_n : \tau\} \mid \alpha \mid \top \mid \bot \mid \tau \sqcup \tau \mid \tau \sqcap \tau \mid \mu\alpha.\tau$$

Figure 1: Syntax and type definition of SimpleSub

At the basis of SimpleSub and SimpleNix is the traversal of an abstract syntax tree (AST) by a typing function `type_term`.

```rust
fn type_term<'a>(ctx: &mut Context<'a>, term: &'a Ast, lvl: usize) -> Result<Type,
SpannedError> {
    match term {
        Ast::Identifier(super::ast::Identifier { name, span, .. }) => ctx
            .lookup_type(name).ok_or(InferError::UnknownIdentifier.span(span)),

        Ast::UnaryOp { rhs, .. } => /* handle unary ops */,

        Ast::BinaryOp { op, lhs, rhs, span } => {
            let ty1 = type_term(ctx, lhs, lvl)?;
            let ty2 = type_term(ctx, rhs, lvl)?;

            match op {
                BinOp::Application => {/* .. */}
                BinOp::AttributeSelection => { /* .. */}
                BinOp::ListConcat => {/* .. */}
                BinOp::Update => {/* .. */},
                BinOp::Mul | BinOp::Div | BinOp::Sub => {/* .. */},
                BinOp::Add => {/* .. */},
                /* .. */
            }
        }

        // Language constructs
        Ast::AttrSet { .. } => { },
        Ast::LetBinding { .. } => { }
        Ast::Lambda { .. } => { }
        Ast::Conditional { .. } => { }
        Ast::Assertion { .. } => { }
        /* .. */
    }
}
```

The recursive function `type_term` takes a term whose type should be determined as an argument together with a context. The context stores the types of known bindings that the algorithm can later look up. The type inference algorithm then matches the AST's variants and computes the type of the given expression. As an example, we take the application operation, where some argument `ty2` is applied to the function with type `ty1`.

```rust
BinOp::Application => {
    let res = Type::Var(ctx.fresh_var(lvl));
    constrain(
        ctx,
        &ty1,
        &Type::Function(Box::new(ty2), Box::new(res.clone())),
    )
    .map_err(|e| e.span(lhs.get_span()))?;
    Ok(res)
}
```

As a first step, a fresh type variable `res` is created, which is then used in the call to `constrain`. The constrain function takes two types as arguments and constrains the first type to be a subtype of the second one. In this case, the function type `ty1` is constrained to be a subtype of the new function type `ty2 -> res`, which simply means the function `ty1` must be able to take an argument of type `ty2`. In the constrain function this constraint is checked and enforced.
The novelty of SimpleSub is that it adds subtyping by only constraining type variables with lower and

upper bounds during AST traversal. All other types, like lists, records, and primitives, are constant and do not change. The lower and upper bounds are stored in mutable vectors and are the only source of mutability in the algorithm.

```
pub struct Var {
    pub lower_bounds: Vec<Type>,
    pub upper_bounds: Vec<Type>,
    pub level: usize,
    pub id: usize,
}
```

Type variables are created not only for applications but also for function arguments, return types, and let-bindings.

The `constrain` function also handles the constraints for functions and their applied arguments as well as all other kinds of constraints. Its shortened definition is given below.

```
fn constrain_inner(lhs: &Type, rhs: &Type, cache: &mut HashSet<(&Type, &Type)>) ->
InferResult<()> {
    if lhs == rhs {
        return Ok(());
    }

    // A cache is needed as type variables can contain themselves in their
    // bounds which would lead to infinite recursion.
    match (lhs, rhs) {
        (Type::Var(..), _) | (_, Type::Var(..)) => {
            if cache.contains(&(lhs, rhs)) {
                return Ok(());
            }
            cache.insert((lhs, rhs));
        }
        _ => (),
    }

    match (lhs, rhs) {
        // Constraining two functions
        (Type::Function(l0, r0), Type::Function(l1, r1)) => {
            constrain_inner(l1, l0, cache)?;
            constrain_inner(r0, r1, cache)?;
        }
        // Constraining two records (attrset)
        (Type::Record(fs0), Type::Record(fs1)) => {
            for (n1, t1) in fs1 {
                match fs0.iter().find(|(n0, _)| *n0 == n1) {
                    Some((_, t0)) => constrain_inner(t0, t1, cache)?,
                    None => return Err(InferError::MissingRecordField { field:
n1.clone() }),
                }
            }
        }
        // Constraining a type variable.
        // The bounds are the only source of mutability of the algorithm.
        (Type::Var(lhs), rhs) if rhs.level() <= lhs.level => {
            lhs.upper_bounds.push(rhs.clone());
            for lower_bound in &lhs.lower_bounds {
                constrain_inner(lower_bound, rhs, cache)?;
```

```
            }
        }
        // Constraining a type variable.
        (lhs, Type::Var(rhs)) if lhs.level() <= rhs.level => {
            rhs.lower_bounds.push(lhs.clone());
            for upper_bound in &rhs.upper_bounds {
                constrain_inner(lhs, upper_bound, cache)?;
            }
        }
        // Constraining a type variable.
        (Type::Var(_), rhs) => {
            let rhs_extruded = extrude(rhs, false, lhs.level(), &mut HashMap::new());
            constrain_inner(lhs, &rhs_extruded, cache)?;
        }
        // Constraining a type variable.
        (lhs, Type::Var(_)) => {
            let lhs_extruded = extrude(lhs, true, rhs.level(), &mut HashMap::new());
            constrain_inner(&lhs_extruded, rhs, cache)?;
        }

        /* More cases... */

        // For any case that is not handled, an error is returned.
        _ => {
            return Err(InferError::CannotConstrain {
                lhs: lhs.clone(),
                rhs: rhs.clone(),
            })
        }
    }
    Ok(())
}
```

Functions are subtyped as usual where $f_1 : \tau_1 \to \tau_2$ is a subtype of $f_2 : \tau_3 \to \tau_4$ if and only if $\tau_1 < \tau_3$ and $\tau_4 < \tau_2$. Records use the usual depth and width subtyping; variables are constrained based on usage. One important thing to note is the usage of levels to handle let-polymorphism correctly.

4.1.1. Levels

Let-bindings and λ-bindings are two ways to introduce polymorphism to a type system. While let-bindings allow polymorphism by writing an expression once and using it in different places, λ-binding types are inferred from their usage in the function body.
To make let-bindings applicable in different places, they must be formalized with generalized (∀-quantified) type variables that can be instantiated at different types based on the let-bindings usage. Otherwise, it would not be possible to apply a function like $\mathrm{id} : a \to a$ to different types because the first application would lock the let-bindings type.

As soon as a let binding refers to λ-bindings, it can no longer be fully generalized as the λ-binding must not be variable. The only solution to this is a partial generalization that only generalizes a subset of all available expressions. Dolan [3] solves this problem by prefixing a type with bounds $[\Delta]\tau$ so that only the none-referred types are generalized. Parreaux [1] takes another approach with levels, which is the practical way to achieve the same thing in a programming language. The initial level is 0 and increases once a let-binding is created. Every variable of that let-binding is then initialized with level + 1 to mark them as generalized. Upon instantiating that type scheme, only type variables with a level higher than the threshold are generalized (newly created variables).

4.1.2. Intersection and Union Types

Union and intersection types are used to model subtype relations between types on a distributive lattice where types are order $\tau_1 \leq \tau_2$ when τ_1 is subsumed by the type of τ_2, i.e., is less general. The union between two types (also join) is the least upper bound between those types, and the intersection (meet) is the greatest lower bound. $\top$ and $\bot$ are the biggest (union) and smallest (empty) types, respectively, which engulf all other types. Subtyping with arbitrary bounds forms a directed constraint graph that may contain cycles and is hard to reduce. Using constraint types like $(\alpha \to \text{bool}) \to \alpha \to \beta \to \gamma \mid \alpha \leq \gamma, \beta \leq \gamma$ only redirects the problem to the programmer [3], which also not solves the problem. In his Ph.D. thesis, Pottier showed that when restricting intersections to negative (input) and unions to positive (output) positions, the constraint graph is bipartite and can be solved efficiently. To make this distinction, the distributive type lattice $\mathcal{T}$ is split into two subsets of polar variables τ^+ and τ^-, which are annotated by their negative or positive position. This limitation also restricts the manual type definitions that programmers can write as a type definition like $\text{str} \sqcup \text{int} \to \text{int}$ that describes a function taking an argument of either type `str` or `int` is in an illegal negative position.

4.1.3. Recursive Types

Recursive types are types of the form $\mu\alpha.(T \to \alpha)$ that references themselves in their definition and can be unrolled indefinitely $T \to \mu\alpha.(T \to \alpha)$, $T \to T \to \mu\alpha.(T \to \alpha)$, $T \to T \to T \to \ldots$ [1]. A practical example of a recursive type is the call-by-value Y-Combinator of ML that throws away its arguments and returns itself [3]. This function is not typeable in ML at all, but SimpleSub and SimpleNix support it and could type it with any of the following types $T \to T, T \to (T \to T), T \to (T \to (T \to T), \ldots$. To concisely express this type, the recursive type $\mu\alpha.(T \to \alpha)$ is needed. In MLsub, as specified by Lionel Parreaux [1], recursiveness is introduced with recursive let-bindings that can reference themselves in their definition. SimpleNix supports the same let-recursion and adds recursion in set definitions. By allowing recursion in sets and multi-let bindings, two types of recursion can happen: primitive recursion of the shown form that can be typed using $\mu\alpha.(T \to a)$ and mutual recursion between two types. While primitive recursion is allowed in Nix, mutual recursion is detected by the evaluator and results in an error.

4.1.4. Type Simplification

Even though type-coalescing is enough to create principle types, types produced by type inference contain unnecessary structures and type variables, making the types bloated and hard to comprehend. A simplification step is thus needed to create usable types. While Dolan [3] draws the line between types and finite automata, which enables one to leverage existing techniques for automata reduction, the SimpleSub Paper by Parreaux [1] uses a more naive approach with a collection of hands-on reductions that can directly be applied to the output types. Even though the code was not shown in the paper and simplification was only briefly discussed, the associated repository provides simplification code that SimpleNix can leverage in the future.

4.1.5. Specification of SimpleSub

What follows is the specification of SimpleSub [1] that refers to the syntax and type definition in Figure 1. The typing rules in Table 1 contain the usual λ-calculus rules for functions T-Abs, T-App and add record and let-binding rules T-Rcd, T-Proj, and T-Let. T-Sub is needed for subtyping and the T-var rule is specialized in that it allows any number of generalized variables α.

$$\text{T-Var}\ \frac{x : \forall\vec{\alpha}.\ \tau \in \Gamma}{\Gamma \vdash x : \tau[\vec{\alpha} \setminus \vec{\tau}]} \qquad \text{T-Abs}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} \qquad \text{T-App}\ \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{t_1 t_2 : \tau_2}$$

$$\text{T-Rcd}\ \frac{\Gamma \vdash t_0 : \tau_0 \quad \ldots \quad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash \left\{\vec{l} : \vec{t}\right\} : \left\{\vec{l} : \vec{\tau}\right\}} \qquad \text{T-Proj}\ \frac{\Gamma \vdash t : \{l : \tau\}}{\Gamma \vdash t.l : \tau} \qquad \text{T-Sub}\ \frac{\Gamma \vdash t : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash t : \tau_2}$$

$$\text{T-Let}\ \frac{\Gamma, x : \tau_1 \vdash t_1 : \tau_1 \quad \Gamma, x : \forall\vec{\alpha}.\tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \text{let rec } x = t_1 \text{ in } t_2 : \tau_2}$$

Table 1: Typing rules for SimpleSub [1], slightly adjusted for notation.

For subtyping, the subtyping context Σ is used and extended with subtyping hypotheses H, which are added during constraining. The subtyping rules are given in Table 1. The respective rules S-Weaken, S-Assum and S-Hyp are used to handle such typing hypotheses. The rules S-Refl and S-Trans are common subtyping rules, and S-Rec is needed to subtype recursive types. The remaining rules, S-Or, S-And, S-Fun, S-Rcd and S-Depth, are needed for unions, intersections, functions, and records.

$$\text{S-Refl}\ \frac{}{\tau \leq \tau} \qquad \text{S-Trans}\ \frac{\Sigma \vdash \tau_0 \leq \tau_1 \quad \Sigma \vdash \tau_1 \leq \tau_2}{\Sigma \vdash \tau_0 \leq \tau_2} \qquad \text{S-Weaken}\ \frac{H}{\Sigma \vdash H} \qquad \text{S-Assume}\ \frac{\Sigma, \triangleright H \vdash H}{\Sigma \vdash H}$$

$$\text{S-Hyp}\ \frac{H \in \Sigma}{\Sigma \vdash H} \qquad \text{S-Rec}\ \frac{}{\mu\alpha.\tau \equiv [\mu\alpha.\tau/\alpha]\tau} \qquad \text{S-Or}\ \frac{\forall i, \exists j, \Sigma \vdash \tau_i \leq \tau'_j}{\Sigma \vdash \sqcup_i \tau_i \leq \sqcup_j \tau'_j} \qquad \text{S-And}\ \frac{\forall i, \exists j, \Sigma \vdash \tau_j \leq \tau'_i}{\Sigma \vdash \sqcap_j \tau_j \leq \sqcap_i \tau'_i}$$

$$\text{S-Fun}\ \frac{\lhd \Sigma \vdash \tau_0 \leq \tau_1 \quad \lhd \Sigma \vdash \tau_2 \leq \tau_3}{\Sigma \vdash \tau_1 \longrightarrow \tau_2 \leq \tau_0 \longrightarrow \tau_3} \qquad \text{S-Rcd}\ \frac{}{\left\{\vec{t} : \vec{\tau}\right\} \equiv \sqcap_i \{l_i : t_i\}} \qquad \text{S-Depth}\ \frac{\lhd \Sigma \vdash \tau_1 \leq \tau_2}{\Sigma \vdash \{l : \tau_1\} \leq \{l : \tau_2\}}$$

$$\lhd (H_0, H_1) = \lhd H_0, \lhd H_1 \qquad \lhd (\triangleright H) = H \qquad \lhd (\tau_0 \leq \tau_1) = \tau_0 \leq \tau_1$$

Table 2: Subtyping rules of SimpleSub[1], slightly adjusted for notation.

It is important to note that newly assumed subtyping hypotheses must not be applied directly as this would obviously be unsound. To circumvent this, the later modality $\triangleright$ is used to delay the application of hypotheses. When adding a hypothesis with S-Assum, the later modality is inserted in the context and stops the immediate application of S-Hyp until the assumption passes through a function or record rules. The semantic rules for the later modality and its counterpart $\lhd$ are given at the bottom of Table 2.

4.2. Static Type Inference for Nix

The Nix language is a much stronger language than MLSub in that it supports many more primitive types as well as language constructs and makes the core features like sets, let-bindings, and functions more expressive. The following section discusses the differences between the type inference algorithm for MLSub, Simple-Sub, and the type inference algorithm for Nix, SimpleNix.

The Nix language supports the same primitive operations as SimpleSub, so the basic syntax and type definition look very similar to SimpleSub. The only changes are slight syntax extensions and the removal of `primitive` and functions, as they will be added later with advanced syntax.

$$t ::= x \mid t\ t \mid \{l_0 = t; ...; l_n = t;\} \mid t.l \mid \text{let } x_0 = t; ...; x_n = t; \text{in } t$$
$$\tau ::= \tau \to \tau \mid \{l_0 : \tau; ...; l_n : \tau\} \mid \alpha \mid \top \mid \bot \mid \tau \sqcup \tau \mid \tau \sqcap \tau \mid \mu\alpha.\tau$$

Figure 2: Basic syntax and type definition for SimpleNix.

4.2.1. Primitives

The Nix language, instead of only one primitive, supports booleans, strings, paths, floats, and integers. The syntax and type definition is given in Listing 7. From a typing and semantic perspective, there is no difference between integers and floats, so a combined type `num` is used which represents both `int` and `float`. The string and path syntax definitions are given as regex; the string definition is slightly simplified.

Syntax (bools): $b ::= \text{true} \mid \text{false}$

Syntax (strings): $s ::=$ `[a-zA-Z_]*`

Syntax (Paths): $p ::=$ `(./|~/|/)([a-zA-Z.]+/?)+`

Syntax (numbers): $n ::=$ `([0-9]*\.)?[0-9]+`

Syntax extension (primitives): $t ::= ... \mid b \mid s \mid p \mid n$

Type syntax extension (list): $\tau ::= ... \mid \text{bool} \mid \text{string} \mid \text{path} \mid \text{num}$

Listing 7: Syntax and typing syntax for primitives.

T-Bool	T-Path	T-String	T-Num
$\Gamma \vdash b : \text{bool}$	$\Gamma \vdash s : \text{path}$	$\Gamma \vdash p : \text{string}$	$\Gamma \vdash n : \text{num}$

Table 3: Primitives typing rules.

4.2.2. Records

Records are primitives in [1] and can be typed with S-Rcd, Table 1 from SimpleSub. The only distinction between MLSub and Nix is that Nix allows self-referential records with the `rec` keyword, which has two important implications. Firstly, the order of evaluation can no longer be arbitrary because fields can reference each other in arbitrary order. To account for this, the context has to be extended with all record fields up-front, and inference has to jump to unevaluated names in case they are referenced. Secondly, by allowing self-references, it is possible to create two forms of recursion: primitive and mutual recursion. While the first one is allowed, the second is not as explained in Section 4.1.3.

Syntax extension (recursive records): $t ::= ... \mid \text{rec } \{l_0 : t; ...; l_n : t\}$

4.2.3. Lists

The list type in Nix is very versatile in that it allows the concatenation of any arbitrary elements to each other. While this gives the most power to the programmer, it is notoriously hard to derive a type for these lists. A list type that aggregates all item types to a list type best describes the underlying structure but is very rigid and impossible to apply to generic functions like maps that would expect a homogenous list with elements of type T. The other option is to optimistically check every list item

and create a homogenous list type [T] for arrays that only inhabit one element type. While this is possible, it would already be useless for lists that consist of string and path elements. An aggregate type would be created for these lists because string and path are principally different, even though they are used interchangeably in practice. SimpleNix sticks to the latter approach and tries to create a pure list but resorts to an enumeration otherwise.

$$\textit{Syntax extension (lists)}: t ::= \ldots \mid [\ t_0\ t_1\ \ldots\ t_n\ ]$$

$$\textit{Type syntax extension (list)}: \tau ::= \ldots \mid [\ \vec{\tau}\ ]$$

The simple subtyping rule for homogenous lists and a typing rule for arrays is is also added:

$$\text{S-Lst}\quad \frac{\Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash [\tau_1] \leq [\tau_2]} \qquad \text{T-Lst-Hom}\quad \frac{\Gamma \vdash t_0 : \tau \quad \ldots \quad \Gamma \vdash t_n : \tau}{\Gamma \vdash [\ t_0\ t_1\ \ldots\ t_n\ ] : [\tau]} \qquad \text{T-Lst-Agg}\quad \frac{\Gamma \vdash t_0 : \tau_0 \quad \ldots \quad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash [\ t_0\ t_1\ \ldots\ t_n] \leq [\tau_0\ \tau_1\ \ldots\ \tau_n]}$$

4.2.4. Operators

The Nix language has arithmetic, logic, comparison, and some specialized operators for sets and lists. Operators need special handling because of their runtime semantics, which allows them to combine and mix some of their operator types but not all.

The logic operators $\wedge, \vee, \rightarrow, \neg$ and arithmetic operators $+, -, /, *$ are trivial as they only allow numbers and, respectively, booleans as operands. T-Op-Arith and T-Op-Logic (Table 4) rules are used to handle these operators. Addition $(+)$ is defined for strings, paths, and numbers, and it is also possible to mix strings and paths under addition. The first operand is always the deciding factor for the type of the operation. Due to this heavy overloading, three typing rules T-Add-Num, T-Add-Str, T-Add-Path are needed.

The comparison operators $<, \leq, >, \geq, \neq$, and $==$ behave as expected for numbers. Paths and strings are determined lexicographically, but it is worth noting that cross-comparisons between strings and paths are not allowed perpendicular to how addition behaves. Arrays are compared element-wise, up to the first deciding element, and records do not support comparisons even though a depth and width-based comparison would, in theory, be possible.

To type and constrain the above monotone operators with no crossed types, the first operand is checked, and if it is a primitive type, the second type is constrained to the same type. If the first operand is a type variable, the second operand might give a hint upon which the type variable can be constrained. The only unhandled case is when both operands are type variables, leaving us with the only option to add lower bounds for all supported operands on both variables.

Besides the common operators, Nix adds operators for specialized use cases, namely `_++_` (concat), which concatenates two lists, and `_//_` (update), which updates the first record with fields from the second one. Both operators are easy in that they force their operands to be of type list or record, respectively, but they have non-trivial meanings for type inference, which is further discussed in Section 4.2.5.

Finally, Nix introduces helper operators like `?` and `or` that operate on records. The check operator (?) returns a bool, signaling whether the given record has the requested field. If the left operand is a proper record, the check operator is a no-op that trivially returns a bool. If it is a type variable, a record-field constraint with the value `optional<undefined>` can be made on the type variable. While this is not really needed, it can improve record-field completions during configuration, making it unnecessary for Nix developers to look up possible fields manually.

The `or` operator extends the check operator in that it returns a default value in case the requested field of a set does not exist. The type of this operator is $\text{or} : \alpha \to \beta \to \alpha \sqcup \beta$.

Syntax extension (arithmetic): $t ::= \ldots \mid t + t \mid t - t \mid t * t \mid t \;/\; t$

Syntax extension (logic): $t ::= \ldots \mid (t \;\&\&\; t) \mid (t \mid\mid t) \mid (t \to t) \mid !t$

Syntax extension (comparisons): $t ::= \ldots \mid t < t \mid t \leq t \mid t \geq t \mid t > t \mid t == t \mid t \neq t$

Syntax extension (helpers): $t ::= \ldots \mid t \;?\; l \mid t \;//\; t \mid t \;++\; t \mid t.l \text{ or } t$

Figure 4: Operator Syntax extension.

T-Op-Arith

$$\frac{\Gamma \vdash t_1 : \text{num} \quad \Gamma \vdash t_2 : \text{num} \quad \text{op} \in [-, +, /, *]}{\Gamma \vdash t_1 \text{ op } t_2 : \text{num}}$$

T-Op-Logic

$$\frac{\Gamma \vdash t_1 : \text{bool} \quad \Gamma \vdash t_2 : \text{bool} \quad \text{op} \in [\to, \vee, \wedge]}{\Gamma \vdash t_1 \text{ op } t_2 : \text{bool}}$$

T-Add-Num

$$\frac{\Gamma \vdash t_1 : \text{num} \quad \Gamma \vdash t_2 : \text{num}}{\Gamma \vdash t_1 + t_2 : \text{num}}$$

T-Add-Str

$$\frac{\Gamma \vdash t_1 : \text{str} \quad \Gamma \vdash t_2 : \text{str} \sqcup \text{path}}{\Gamma \vdash t_1 + t_2 : \text{str}}$$

T-Add-Path

$$\frac{\Gamma \vdash t_1 : \text{path} \quad \Gamma \vdash t_2 : \text{path} \sqcup \text{str}}{\Gamma \vdash t_1 + t_2 : \text{path}}$$

T-Compare

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2 \quad \tau_1 \equiv \tau_2 \quad \text{op} \in [<, \leq, \geq, >, ==, \neq]}{\Gamma \vdash t_1 \text{ op } t_2 : \text{bool}}$$

T-Negate

$$\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash !e : \text{bool}}$$

T-Check

$$\frac{\Gamma \vdash e : \{l : \tau\}}{\Gamma \vdash e?\, l : \text{bool}}$$

T-Or

$$\frac{\Gamma \vdash t_1 : \{l : \tau_1\} \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1.l \text{ or } t_2 : \tau_1 \sqcup \tau_2}$$

Table 4: Operator typing rules 1.

4.2.5. Mutating Lists and Records

Concatenating two lists is straightforward if both types are proper lists, as a new list type with all elements of the first and second list can be concatenated. Type variables complicate the matter because they don't have a single associated type. During type inference, it is optimistically checked if the variable has a list constraint, and if yes, that constraint is used. Otherwise, the partial list type from the know operands is returned, and the type variable is constrained with an empty list type.

S-List-Concat-Hom

$$\frac{\Gamma \vdash a : [\tau] \quad \Gamma \vdash b : [\tau]}{\Gamma \vdash a \;++\; b : [\tau]}$$

S-List-Concat-Multi

$$\frac{\Gamma \vdash a : [\vec{\tau_1}] \quad \Gamma \vdash b : [\vec{\tau_2}]}{\Gamma \vdash a \;++\; b : [\vec{\tau_1}\vec{\tau_2}]}$$

T-Rec-Update

$$\frac{\Gamma \vdash a : \{l_i : \tau_i\} \quad \Gamma \vdash b : \{l_j : \tau_j\}}{\Gamma \vdash a \;//\; b : a \setminus b \cup b}$$

Table 5: Operator typing rules 2.

4.2.6. Let-Bindings

Let-bindings in the Nix language supersede let-bindings in MLsub because Nix allows multiple variable bindings as part of one let binding instead of only one. Normally, one could construct an isomorphism between the two let-bindings by breaking apart an n-multi-let and creating a chain of n let-bindings to introduce all bindings. This, however, is inapplicable in practice because the name bindings in a multi-let can refer to each other. These references can come in arbitrary order, and form cycles similar to the ones records form. To solve this, every new identifier of the let-binding has to be added to the context up-front so that it can be referred to during typing.

$$\text{T-Multi-Let}\quad \frac{\Gamma[\overline{x_i : \tau_i \vdash t_i : \tau_i}^{\,i}] \qquad \Gamma[\overline{x_i : \forall \vec{\alpha}.\tau_i}^{\,i}] \vdash t : \tau}{\Gamma \vdash \text{let } x_0 = t_1; ...; x_n = t_n \text{ in } t : \tau}$$

Table 6: Let typing rule.

4.2.7. Functions

Primitive function definitions, like the ones in MLSub, only allow single identifiers to be function arguments. Nix extends this by allowing destructuring set patterns as function arguments as well. To add them to the language, a new type, `Pattern,` is added. This type mirrors the record type but adds a boolean flag that expresses whether the pattern allows any non-enumerated record fields, i.e., if the pattern is a wildcard pattern like `{x, y, ...}`. In this pattern, the ellipsis signals that the supplied record can have more than the enumerated fields x and y. Pattern constraining flows contrary to how identifier arguments are constrained. Identifiers are introduced as empty type variables and constrained based on their usage in the function's body. Patterns already define a structure that only partially changes with the function's body. All fields of the pattern must be added as fresh variables to the context because they might be referred in the function's body just as normal top-level arguments. These new variables are either added empty if they define no default value or with a constraint for the default value if it exists. The pattern argument x of g `g: {x ? 1, y}` would be added as a variable with a constraint on `num` while the identifier for y would be added without constraints.
The alias that can be specified with the @-syntax refers to the whole argument that was supplied to the function. Trivially, it also has to be added to the context, and in the simple case of a wildcard pattern, it only needs a record constraint. If it is not a wildcard pattern, it must be added with a pattern constraint so that the constraint function can throw an error if too many fields are supplied.

Due to the new pattern type, constraining has to account for the case that a function with a pattern argument is called. If the given value is a record itself, the provided fields can be compared one by another, and if all fields exist and subsume the pattern fields, constraining succeeds. In case the pattern is not a wildcard pattern, it has to be checked that the given record does not have any additional fields, and an error has to be raised otherwise. If the value given to the function is a type variable, we can only constrain the type variable to have all requested fields.

Pattern Element Syntax: $\text{elem} ::= x \mid x \ ? \ t$

Pattern Syntax: $\text{pat} ::= \{\ \text{elem}_0, ..., \text{elem}_n\ \} \mid \{\ \text{elem}_0, ..., \text{elem}_n, \ ...\ \} \mid x$

Language extension(function): $t ::= ... \mid \text{pat} : t$

Type syntax extension(pattern): $\tau ::= ... \mid (\{l_0 : \tau; ...; l_n : \tau\}, \text{bool})$

Figure 5: Function syntax definition.

4.2.8. Conditionals

Conditionals are not part of the core language specification for SimpleSub, as they can easily be added to the language by prefilling the context with `f: bool → α → α → α` and rewriting the if construct as an application to this function [1]. It has been shown by Dolan [3] that `f: bool → α → α → α` is a subtype of the more natural-looking type $f : \text{bool} \to \gamma \to \beta \to \gamma \sqcup \beta$ that explicitly allows both branches to have different types by substituting $\alpha = \gamma \sqcup \beta$. Nix has the same syntax and semantics for if statements, so a similar approach can be used. In practice, it is important to create errors referencing proper code locations so that conditionals have to be handled explicitly, but that is only an implementation decision.

Syntax extension (conditionals): $t ::= \ldots \mid \text{if } t \text{ then } t \text{ else } t$

T-If

$$\frac{\Gamma \vdash t_1 : \text{bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \tau}$$

Table 7: Syntax and typing rules for conditionals.

4.2.9. Inherit Statements

The inherit statement is syntactic sugar to reintroduce bindings from the context Γ into a let-binding or record. Inherit statements of the form `inherit (path) x` can be rewritten to an assignment `x = path.x`, where `x` is an identifier and path is a sequence of field accesses to a deeply nested record. From a typing perspective, rewriting is the proper solution as it does not introduce any new typing rules, but for a language server, that is not quite enough. To create good errors, it is necessary to relate to the source code instead of internal rewrites. That is why inherits must be handled more cautiously, delaying lookup to the latest possible point so that if identifiers do not exist, a respective error referencing the proper location in code can be thrown.

Syntax Path: $p ::= x \mid p.x$

Syntax Inherit: $s ::= \text{inherit } x; \mid \text{inherit } (p)\ x;$

Syntax overwrite(records): $t ::= \ldots \mid \{\ s_i\ \}$

Let variable assignment: $a ::= x = t; \mid s$

Syntax extension(let): $t ::= \ldots \mid \text{let } a_i \text{ in } t$

Figure 6: Syntax rules for let-bindings.

4.2.10. With Statements

Another language extension of Nix is the `with` statement. The with statement brings all fields from a record into scope for the following expression. If the imported record has a proper record type, all its fields can be brought into scope, similar to how a let binding would do. The only distinction is that bindings introduced by a with-statement never shadow variables introduced by any other means, meaning every new name has to be checked if it is already part of the context and only be added if it is not.

If the added record is a type variable, the constraint direction reverses, such that the type variable is constrained based on its usage in the following expression. This prohibits the existence of free variables in the sub-expression because every variable will be associated with the new type variable. To account for this in code, a single field `with-statement` is added to the context struct that remembers the last with-statement. Because fields from a with-statement never shadow variables introduced by other means, with-statements are not mere syntax sugar and need to be handled by their own typing rule T-With.

Syntax set: $\text{set} ::= \{l_i : t\} \mid x$

Syntax extension(with): $t ::= \ldots \mid \text{with set}; t;$

T-With

$$\frac{\Gamma \vdash t_1 : \left\{\vec{l} : \vec{\tau}\right\} \quad \Gamma, l_0 : \tau_0, ..., l_n : \tau_n \vdash t_2 : \tau \quad l_i \notin \Gamma}{\Gamma \vdash \text{with } t_1; t_2 : \tau}$$

Table 8: Syntax and typing rules for the with statement.

4.2.11. Assert Statements

Assertions precede expressions and allow for early program exit if some condition does not hold. From a typing perspective, assertions do not differ from normal sequential execution and can be inferred by just evaluating the first and then the second expression.

Syntax extension(assert): $t ::= \ldots \mid \text{assert } t; t;$

T-Assert

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \text{assert } t_1; t_2 : t_2}$$

Table 9: Syntax and typing rules for assert statements.

4.2.12. Large Expressions and Lazy Type Inference

The Nix package repository contains over 80.000 packages and is the largest package repository in existence. The public nixpkgs Github mono repository provides all packages, the standard library, NixOS, and a lot of utility functions from its root Nix file. Importing this root file and evaluating the whole package tree takes an unfeasible amount of time, making type inference unpractical slow. This is why SimpleNix is not *yet* applicable for common practical tasks like configuration writing and flaking. For now, the Garnix project restricts itself to single-file type inference without multi-file support, as adding even the most basic import for the nixpkgs would make type inference unusable and slow.

To eventually overcome this, the following simple approach should suffice. Import statements usually aren't located at the top of the file but rather in code wherever they are needed. This makes import statements a good candidate for code splitting during type inference. Only when an uninitialized field is needed during tree traversal should the related expression be imported and inferred. This should, in theory, make package completion possible as packages are defined in one big record(mapping) between the package name and import statements for that package. A smart language server could thus create the full list of packages and lazily load more information about them upon package selection, greatly improving the developer experience. Similar approaches should be possible for option and module auto-completion, which have similar structures.

4.3. Examples

The repository associated with this bachelor thesis at https://github.com/Septias/garnix contains the parser, inference algorithm, and language server. The inference algorithm and parser are heavily tested and are good sources for determining what SimpleNix is capable of. What follows is a short showcase of type inference in SimpleNix.

The trivial cases for primitive types produce proper types, and list types are reduced to homogenous lists.

```
coalesced("{ x = true; y = false;}") // {y: Bool}, {x: Bool}
coalesced("{ x = true; y = false;}") // {y: Bool}, {x: Bool}
coalesced(r#"{ x = ["hi" 1];}"#) // {x: [ String Number ]}
coalesced("{ x = [1 1];}") // {x: [ Number ]}
coalesced("{ x = {y = 1;};}") // {x: {y: Number}}
```

Functions get the correct function type and properly constrain records. The last function is an example from [1] that showcases type inference and type coalescing. Notice that occurrences of `Var(num)` have been replaced by proper type variable names $\alpha, \beta, \gamma, \delta, \ldots$ to improve notation. Also, proper type simplification still has to be added, which is why some types still have unnecessary type variables.

```
coalesced("x: x + "hi";") // "α ∧ String -> (String)"
coalesced("x: x.y") // "α ∧ {y: β} -> β"
coalesced("f: x: f (f x)") // "α ∧ (β -> γ) ∧ (γ -> δ) -> β -> δ"
```

Let bindings create new variables, and inherit statements preserve type information. The last example also shows the proper reduction of with-statements.

```
coalesced("let x = 1; in {inherit x;};") // {x: α ∨ Number}
coalesced(r#"let t = { x = 1; }; f = t; in {inherit f t;};"#) // {t: Var ∨ Record},
{f: Var ∨ Record}
coalesced("with {y = 1;}; {z = y;};") // {z: Number}
```

Function types properly propagate constraints through the newly added pattern type, and optional values are properly installed for the check operator.

```
coalesced("let f = { x, ... }: x; in f {x = 1; y = 2;}";) // Var ∨ Number
coalesce("x: x ? y"); // α ∧ {y: Optional<Undefined>} -> (Bool)
```

The test cases for the parser and inference algorithm in this project's repository provide many more examples of what SimpleNix can infer.

5. Language Server

Part of my contribution is a language server for the Nix language. It acts as a frontend to play with the implementation. The language server protocol supports various code actions, from which Garnix supports type hints and error reporting. Type hints are supported in the form of `Inlay hints`, small text boxes that are shown behind every identifier and display the type of variables. Errors reported during type inference and parsing are collected together with their position in the document. These errors are then displayed in the document for quick and interactive error fixing.

6. Discussion and Further Work

This thesis implements and extends the SimpleSub type inference algorithm, finally bringing type inference to the Nix programming language by providing a language server and a rust parser for the Nix language. This, together with other great recent improvements in documentation, language servers, and evaluators, lays the first stepping stones for great developer experience in Nix.

To make Garnix applicable in general, the remaining problems of type simplification and unbound variables, which were out of the scope of this thesis, need to be resolved. The Garnix repository is left with a permissive Apache 3.0 license, making it a public domain, and will undergo continuous development in the future to eventually overcome these problems.

7. Acknowledgements

Thanks to all peer reviewers, and especially Marius Weidner and Christian Weber, for their great support during and after writing this thesis.

Bibliography

[1] L. Parreaux, "The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl)," *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, Aug. 2020, doi: 10.1145/3409006.

[2] J. Dieguez Castro, "NixOS," in *Introducing Linux Distros*, Berkeley, CA: Apress, 2016, pp. 301–327. doi: 10.1007/978-1-4842-1392-6_14.

[3] S. Dolan and A. Mycroft, "Polymorphism, subtyping, and type inference in MLsub," *SIGPLAN Not.*, vol. 52, no. 1, pp. 60–72, Jan. 2017, doi: 10.1145/3093333.3009882.

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids other than those listed have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Freiburg den 27.03.2024

Place, Date

Signature