Contents

1. Polymorphizing NAGs 1
Figure 1: The structure of this section. 1
1.1. NAGs 2
1.2. Moving the language check to the type system 2
1.3. Polymorphic equations 5
1.4. Polymorphic attributes 6
1.5. Polymorphic NAGs 6
2. Basic Type System 6
2.1. Relations 8
2.2. Typing Equations 9
2.3. Typing Expressions 9
2.4. Defining Languages 11
3. TODOs 11
3.1. Can Q contain $=$-constraints against a skolem? 11
3.2. What do we need for good editor tooling? 12
4. Miscellanea 12
4.1. Property-testing Unification 12

1. Polymorphizing NAGs

In this section, we will present a series of slightly different formalisms, starting with the one from Nanopass Attribute Grammars and culminating in the one we will discuss for the rest of the paper. The structure of this section is presented in Figure 1. We'll present these formalisms informally for now, and describe the overall "Polymorphic NAGs" system more completely in the next section.

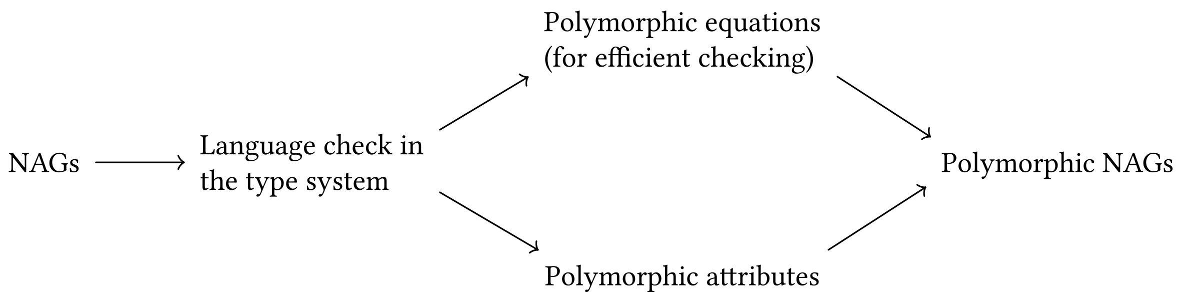

Figure 1: The structure of this section.

1.1. NAGs

We'll start with approximately what's in the Nanopass Attribute Grammars paper. Our metalanguage's grammar is as follows:

```
Root ::= AttrDecl* ProdDecl* LangDecl*
AttrDecl ::= ("syn" | "inh") Name Type
ProdDecl ::= Name ProdSig Equation*
ProdSig ::= Name Type "::=" (Name Type)*

LangDecl ::= TODO
Type ::= TODO
Equation ::= TODO
```

1.2. Moving the language check to the type system

In nanopass attribute grammars as originally presented, languages are not represented in the type system. Instead, we determine what productions and attributes are available based on the context of the equation in which they're used. This check is a separate check that happens after type checking.

The first step in integrating this check with the type system is to give every nonterminal type a language as well. Instead of the nonterminal x being a type on its own, we now have $l.x$, where x still names a nonterminal and l names a language.

$$\frac{l : \mathsf{lang} \quad x : \mathsf{nt}}{l.x : \mathsf{type}}$$

If our supporting system is something like System F_ω, we could make the dot itself be a type constructor whose kind is $\mathsf{lang} \to \mathsf{nt} \to \mathsf{type}$, but this isn't necessary to make the language check work.

This allows us to express language polymorphism using ordinary type polymorphism. For example, we might have an inherited attribute representing the typing context of our object language, mapping names to object language types, which might originally be declared like `inh lexicalCtx: Map[String, Type];`. However, this doesn't specify what language we're expecting the types to be in.

In the original NAG system, we'd implicitly TODO.

> **FIXME**: It's been allows us to be polymorphic. Over languages. Or even entrepreneurs if we wanted. Although, that seems less useful. For example. We might have an inherited attribute. It represents the taping context. Or a turn.

This inherited attribute. Might be a map. Napping strings. Terrible names. To types. However, types of which language

In. If we are using this inherited attribute across, Several languages. We would expect the types would be for the particular language. That we're currently in. However, we might normalizing or lowering types. As we go between languages. Therefore, It would make more sense. For our inherited attributes to high context.

To be parametric over a language l. And then have the type now. String. Comma. L dot type.

FIXME: Next. We have the problem. Of efficiently tape checking. These declarations. Oh yeah. First, you have to actually define the typing rules. Here. We're sort of.

Um, Here, one approach. Is to introduce constraints. Similar to type class constraints. Or other. Constraints, we might have for constrained point orgasm. Such as subtype, constraints. Um, They're reflect. The language definitions themselves. End of the type system. For instance. If we have a negation, Node. Which is an expression.

The type of an actual call, the negation node might be something like L dot expert. Where, L. Dot expert has the production negate? If we have some larger term stored in a Constant declaration. That we still wish to be polymorphic. Over the language. It could have more complicated constraints.

Such as The language. Uh, it would be an l dot expert. Or the language l. In non-terminal expert. As productions, integer negate and app. We also need to add constraints. To define attributes. To find the bad words. It declare the presence of attributes. Or non-terminals.

So we do that. And then we also have low type questions. Womp. Um,

Well, since attributes. Oh, since actually equations.

I guess the flow type thing would just be That we say, Um, that when we're decorating. Some non-terminal root. Then. We say that the set of inherited attributes, That occur on it. Is equal to the ones that were decorating it with. Equal could also be a subset. But, Yeah, it could be.

Um, But i suppose in monomorphic cases, The subset might be confusing. Because then extra. Declarations provide by the user wouldn't be used. So we do a quality instead. Similarly, if you pattern match, On a term. To figure out what production it is. We can strain the set of productions.

In the appropriate on terminal to be exactly equal. To the Um, set of productions that the pattern mashup sets. Unless there's a wild card case. If there's

a wild card. We simply say, it has to be a super set or equal to And maybe there could be some annotation, but where I don't need any annotation.

Actually, no definitely don't need any addiction.

The last little bit of complication there. Is that when we're type checking? We have to deal with polymorphic attributes. If we don't actually need to support language fly more prism in attributes, And we're just moving. The language sex system, we don't need to do this. But since we're tempting to gain language point, morphism.

We should think about it. And yeah, that means they should probably get moved. Two, one. Dot. Three or something. Or whatever the sections of being k. Dot three. Um,

Yeah, getting this attribute. Here, take the family. It takes the language on terminal, name and attribute name. And, Returns the Type, the attribute has

Principle. This could. The. Simplified to the language instead if we only want support language, finance and attributes. Not take polymorphism. The language within only decline. Query for And attributes. That are quite more fit.

However, overall this doesn't simplify The system very much. Although maybe it makes checking more decidable. Um, More decidable and practice, not in theory.

Uh, A gross little complication here. Is that we're putting attribute names? Into the types of stuff. Something prior work. Something something happened data. Kind something something. They're interpreted strings. Well. They're only interpreted for the purposes of equality.

Then. The type of, actually access. Is the result of that type family. Which in any cases, should be able to be locally solved for Um,

To do the answers. Probably, yes. But check that it's not possible. To introduce a hypothetical. Constraint. It isn't actually. Drivable in terms of the type family, like having one type, family definition, That. Computes f of n equals n. And then being under a hypothetical, Where evident equals unit.

Maybe. Yeah, this definitely belongs in the section. Kate off three. Because once you've introduced the type family, We have all sorts of complications related to solving families. And in general, Completely losing out on injectivity of typeformers. Um,

FIXME: Next up, our issues of efficiency. And the original man of passage the grammar's formalism the type checking process is performed. Once while, The language, check. It's performed. Once per language. This is deemed to be

acceptable during this simplicity of the language, check It's a simple syntax direction. Analysis.

It deals only with first order terms as a result, it should be very efficient. Uh, calculate. Even if it needs to be calculated multiple times. However, if we're putting the language analysis into the type system, We should also be sure that the type checker still only needs around once rather than what's for every language.

Not only was running the language analysis once per defined language, be much less efficient. Then running at once. Hi all the overhead attack checking as well. Would be present. Given the type checking isn't much more complicated process involving proper first. Order nominal or a higher order unification. We really want to avoid this.

The constraint problems involved. Are also much larger. And performance would suffer as a result.

So it is absolutely critical. That. The overall type checker. Only needs to traverse terms once. But how can we do this when we might need to use an equation in several languages? And we don't know which languages up front. Parenthetical. We don't know which language is up front. Because we want to support we use in modularity.

Wish. Production or attribute, should be able to be better than the library. Every years later by other code. Without having to rerun the type checking process. And parenthetical.

The way we do this is by deferring, constraints that we need more information from the language in order to satisfy

A language, an equation. When checked. In a system like, outside in of x. A functional definition. Well. Oh actually. Hold on. Introduce and quantifiers.

Most of the time. When tape shocking. We have a known set of constraints that are true. Global axioms and local assumptions. For example. Went hype. Checking the function. F of x goes to. Show x, comma 5. Note to change five, to char five. We have a global set of accidents.

Such as show char. And show a conjunction, show the implies show pair a commodity. We also then might have a local assumption such as show a To let this type check this overall function as a arrow string. Contrastically in our scheme.

He might. Do something like,

We already know the type. That's an attribute. So check against that with the set of global wax gyms. However, myself upload straights. And recording.

1.3. Polymorphic equations

1.4. Polymorphic attributes

1.5. Polymorphic NAGs

2. Basic Type System

> **NR**: Assume enough of the background has already been presented for the reader to be familiar with nanopass attribute grammars and the language check.

We might wonder if the language check can be moved into the type system. It is, after all, a syntax-directed check.

We might also want to run the language check as a single analysis over all the equations, rather than once per equation per language.

This section will describe a simple type system based on System F_ω with constrained polymorphism whose type-checking serves as the language check.

The essence of the approach is to no longer consider nonterminal names types, but rather their own kind. Language names are also their own kind.

$$\frac{L; \sigma; \Gamma; Q \vdash x \text{ names a nonterminal}}{L; \sigma; \Gamma; Q \vdash x : \mathsf{nt}} \text{ K-NT}$$

$$\frac{L; \sigma; \Gamma; Q \vdash x \text{ names a language}}{L; \sigma; \Gamma; Q \vdash x : \mathsf{lang}} \text{ K-LANG}$$

We can also turn a pair of a language name and nonterminal name into a type. This type characterizes terms of the nonterminal built in the language. (We do very similar things for reference types, which characterize trees built in the language.)

$$\frac{L; \sigma; \Gamma; Q \vdash l : \mathsf{lang} \quad L; \sigma; \Gamma; Q \vdash n : \mathsf{nt}}{L; \sigma; \Gamma; Q \vdash \mathsf{Term}[l, n] : \mathsf{type}} \text{ K-TERM}$$

$$\frac{L; \sigma; \Gamma; Q \vdash l : \mathsf{lang} \quad L; \sigma; \Gamma; Q \vdash n : \mathsf{nt}}{L; \sigma; \Gamma; Q \vdash \mathsf{Tree}[l, n] : \mathsf{type}} \text{ K-TREE}$$

> **NR**: It's more expressive in the absence of type-level lambdas to instead have:
>
> $$\frac{}{L; \sigma; \Gamma; Q \vdash \mathsf{Term} : \mathsf{lang} \to \mathsf{nt} \to \mathsf{type}} \text{ K-TERM}$$

$$\frac{}{L; \sigma; \Gamma; Q \vdash \mathsf{Tree} : \mathsf{lang} \to \mathsf{nt} \to \mathsf{type}} \text{ K-TREE}$$

Once these exist, the normal T-TAPP can be used to implement the above.

(We might not want type-level lambdas because type unification will then be higher-order unification.)

If the nonterminal does not appear in the language, these types are equivalent to the empty type, but no unsoundness results from being able to write them. (It's probably worth a compiler warning, though.)

We keep the standard metatheoretic property that all expressions' types are of kind type.

$$\frac{L; \sigma; \Gamma; Q \vdash e : \tau}{L; \sigma; \Gamma; Q \vdash \tau : \mathsf{type}}$$

As a result, no expression is of nonterminal type. Instead, the language an expression is in must be known as a result of type-checking.

This poses a complication for production signatures. Recall that previously, production signatures σ were of the form $(x_0 : N_0 ::= x_1 : \tau_1, \ldots, x_k : \tau_k)$, where $\tau_i : \mathsf{type}$. However, with our changes, a nonterminal cannot appear as a child, since the kinds will not match.

The simplest fix here is to lift the restriction that $\tau_i : \mathsf{type}$, and instead allow it to be nt or type.

NR: This assumes that nonterminals and productions can't have polymorphism (universal or existential quantification)!

If we ever solve the "list nonterminals" problem, we'll need polymorphic nonterminals. Silver currently supports existentials. I dunno if we actually use them anywhere that shouldn't be a datatype, though.

(I *do* want to support existentials in datatypes, but the hard problems there are mostly around type inference.)

Language Polymorphism: The typical System F_ω-like rules exist to allow expressions to depend on types, and types to be applied to expressions. Since the type may be of any kind, it can be of kind lang or nt as well. This gives us a notion of *language polymorphism*.

$$\frac{L; \sigma; \Gamma, x : \kappa; Q \vdash e : \tau}{L; \sigma; \Gamma; Q \vdash (\Lambda(x : \kappa).\ e) : \forall(x : \kappa).\ \tau} \text{ T-FORALL}$$

$$\frac{L;\sigma;\Gamma;Q \vdash e : \forall(x : \kappa).\ \tau' \quad L;\sigma;\Gamma;Q \vdash \tau : \kappa}{L;\sigma;\Gamma;Q \vdash e[\tau] : \tau'[x := \tau]}\ \text{T-TAPP}$$

NR: I'm kind of hand-waving around this, but I'm assuming the expression language is pretty much System F_ω as presented in Figure 30-1 of TAPL, plus the rules presented here. T-FORALL and T-TAPP are essentially exactly what's there.

This lays the groundwork for us to talk about the typing of the actual attribute grammar constructs.

2.1. Relations

To start off, we define a relation that establishes that an equation is well-typed:

$$L;\sigma;Q \vdash (x.a := e)\ \textbf{well-typed}$$

This relation states that inside the body of a production with signature σ, the equation $x.a := e$ is well-typed in language L if the constraints Q hold.

We also have relations to establish the types of expressions and the kinds of types:

$$L;\sigma;\Gamma;Q \vdash e : \tau \qquad\qquad L;\sigma;\Gamma;Q \vdash \tau : \kappa$$

This relation states that inside the body of a production with signature σ, with other expression or type variables in scope whose types or kinds are given by Γ, the expression e (or type τ) has the type τ (or the kind κ) if the constraints Q hold.

NR: There'll probably be more relations here if I attempt to prove anything about what type inference can or can't do.

While these judgements might typically be read in terms of Q and L being inputs to a type-checking procedure, our overall type-checking will be performed by treating L as a Skolem variable and inferring a minimal Q necessary for the judgements to be provable. We'll talk more about this later.

NR: Should I write these like $L;\sigma;\Gamma \vdash e : \tau \rightsquigarrow Q$ instead? OutsideIn(X) does $Q;\Gamma \vdash e : \tau$ for the specification, $\Gamma \mapsto e : \tau \rightsquigarrow Q$ for the algorithm.

2.2. Typing Equations

The rules for typing equations are fairly typical. An equation for a synthesized attribute is well-typed if its receiver is the left-hand-side of the production and its expression has the correct type. An inherited attribute equation is well-typed if it's receiver is on the right-hand-side of the production and its expression has the right type as well.

$$\frac{\sigma = (x : N ::= \ldots) \quad a \in A_S \quad L; \sigma; \cdot; Q \vdash e : \Gamma_A(a)}{L; \sigma; Q \vdash (x.a := e) \ \textbf{well-typed}} \text{ W-S}\textsc{yn}$$

$$\frac{\sigma = (\ldots ::= x : X) \quad a \in A_I \quad L; \sigma; \cdot; Q \vdash e : \Gamma_A(a)}{L; \sigma; Q \vdash (x.a := e) \ \textbf{well-typed}} \text{ W-I}\textsc{nh}$$

NR: This also ignores the possibility of an attribute being polymorphic. We want to support that too!

Note that we don't actually check if a occurs on N (or X) at this point. The type system doesn't assume that all languages are known at the point equations are checked. This fits well with existing notions of modularity.

Unlike in the original formulation of nanopass attribute grammars, we don't provide rules for checking transform attribute equations. In fact, we don't even have an A_T!

NR: The "not having A_T" will be in an earlier section.

We get enough flexibility from this new approach that we can instead reasonably make transform attributes simply be another kind of automatic attribute based on synthesized attributes, and use the W-Syn rule.

NR: Automatic attributes will be in the background section.

2.3. Typing Expressions

NR: TODO: Describe the variable-looking rules; none of these are too complicated.

Also, decide if σ should even be in the expression typing judgement; if instead, the W-Inh and W-Syn rules construct an initial Γ from both sides of the signature, maybe that's fine?

It saves us from muttering about the Barendregt convention, at least.

$$\frac{(x : X) \in \sigma \quad L; \sigma; \Gamma; Q \vdash X : \mathsf{nt}}{L; \sigma; \Gamma; Q \vdash x : \mathsf{Tree}[L, X]} \text{T-Child}$$

$$\frac{(x : X) \in \sigma \quad L; \sigma; \Gamma; Q \vdash X : \mathsf{type}}{L; \sigma; \Gamma; Q \vdash x : X} \text{T-DataChild}$$

$$\frac{(x : X) \in \Gamma}{L; \sigma; \Gamma; Q \vdash x : X} \text{T-Var}$$

NR: Ugh, I guess it's maybe reworking Γ_A as a whole to support polymorphic attributes. I'd be fine if "version zero" of this didn't support normal polymorphism on types, but I'd really like to be able to have attributes that are polymorphic on language to describe normal higher-order attributes (e.g. `syn type[l]: l.Type;`).

The problem is then that we might have any or all of:

- `lang L0 { Expr.attrs := { type[L0] } }`
- `lang L1 { Expr.attrs := { type[L1] } }`
- `lang L2 { Expr.attrs := { type[L0] } }`

Plus, we don't want to look at the languages at this point.

One solution might be to make a kind of attributes attr, and have Γ_A as a type family(!) of kind $\mathsf{lang} \to \mathsf{nt} \to \mathsf{attr} \to \mathsf{type}$. Then, we could bubble up constraints like $\Gamma_A[L, \texttt{Foo}, \texttt{bar}] = \texttt{int}$ into Q.

Maybe also ones like:

$$\begin{aligned} \exists L'. \quad & (\Gamma_A[L, \texttt{Expr}, \texttt{type}] = L'.\texttt{Type}) \\ & \wedge (L'.\texttt{Type}.\mathsf{prods} \subseteq \{\texttt{intType}\}) \end{aligned}$$

(This would be subject to Skolemization so I think L' would be a Skolem variable in practice?)

$$\frac{L; \sigma; \Gamma; Q \vdash e : \mathsf{Tree}[L', N] \quad \Gamma_A(a) \in \Gamma}{L; \sigma; \Gamma; Q \vdash e.a : \Gamma_A(a)} \text{T-Attr}$$

NR: Gonna give the previous box some time to marinate; if I go for it, I'll be rewriting 1.2 and 1.3, so I'll pause continuing to work on them.

The below should still work with that change though.

2.4. Defining Languages

Once equations have Q's computed, we can efficiently check language definitions. This process is pretty simple:

- Define the language provisionally (making all the $\mathtt{L_n}.N.\mathsf{attrs} \subseteq \{a\}$, $\mathtt{L_n}.N.\mathsf{prods} \subseteq \{a\}$, etc. constraints for the concrete $\mathtt{L_n}$).
- For every equation that should be included, check that $Q[L \coloneqq \mathtt{L_n}]$ is solvable.
- …you're done!

3. TODOs

This is some stuff to remember to think about that's longer than what'd conveniently fit in a box.

3.1. Can Q contain $=$-constraints against a skolem?

It seems like the following code is pretty reasonable:

```
syn foo: L0.N;
prod bar(): N {
  this.foo := this;
}
lang L0 {
  N.attrs := { foo },
  N.prods := { bar }
}
```

However, the type-checking algorithm (at least the version in my head as of 2023-11-09) will find minimal well-typedness conditions of:

$$L; (\mathtt{this} : \mathtt{N} \mathrel{{:}{:}{=}}); \{L = \mathtt{L0}\} \vdash (\mathtt{this.foo} \coloneqq \mathtt{this})\ \textbf{well-typed}$$

In OutsideIn(X), we'd bail out as soon as we saw the $L = \mathtt{L0}$ constraint rather than adding it to Q:

```
f :: _ => [a] -> _ -> a
f [] x = x
f (h:_) True = h
f (_:t) False = f t True
```

In this Haskell program, we could infer $\mathtt{f} :: (\mathtt{a} \sim \mathtt{Bool}) \Rightarrow [\mathtt{a}]\ \text{->}\ \mathtt{Bool}\ \text{->}\ \mathtt{a}$, but does anyone expect that? I guess this kind of gets to their position on let generalization there; it's kind of **PartQual** from there?

3.2. What do we need for good editor tooling?

If we're checking equations first, then languages, we're kind of assuming the equations are correct and the languages they're being put into are the possibly-wrong thing.

However, this is kind of backwards for editor tooling; the languages are the overall design that a lot of thought goes into and infrequently change; the equations are the bulk of the code and often change.

We might want to support a different type-checking algorithm for use in the editor, where we check equations against every known language (including in downstream modules) that use them.

This is of course asymptotically slower, but it's still polynomial, and it's incrementalizable.

Example:

```
syn foo: Int;
lang L0 {
  Expr.prods := { addExpr, intExpr },
}

aspect addExpr(l: Expr, r: Expr): Expr {
  this.foo = l.match {
    addExpr(_, _) => 0,
  };
}
```

This should ideally point an error at the `match` in addition to or instead of the `lang`.

4. Miscellanea

4.1. Property-testing Unification

The full Sylvan type system is rather complicated. To increase our confidence in the correctness of the implementation, it makes sense to randomly test against carefully selected properties. In this section, we describe a rather complex property used to test the unification algorithm, which is complex mainly due to the presence of binding constructs (i.e., `forall`) in types. The most general unifier of `forall("x", named("Type"), lexical("x"))` and `forall("y", named("Type"), ?0)` is *not* $\{?0 \mapsto$ `lexical("x")`$\}$ as might be expected in a type system whose types lack binding structure, but rather is $\{?0 \mapsto$ `lexical("y")`$\}$.

In order to define our property, we first define the notion of a path inside a term. (In this section, terms are the abstract syntax of types, but this section

generalizes to other terms we might wish to unify against.) A path is a sequence of natural numbers identifying the indices of a term's children to traverse. This notion should become clearer with examples. Let t be the term:

```
app(named("Foo"),
    forall("x", named("Type"), arrow(lexical("x"), named("Type"))))
```

- The path $\langle\rangle$ is valid, and identifies the entire term t (this is the case for any t).
- The path $\langle 0 \rangle$ identifies the term `named("Foo")`.
- The path $\langle 1, 2, 0 \rangle$ identifies the term `lexical("x")`.
- The path $\langle 1, 4 \rangle$ is invalid.

We can then define an auxiliary function `replace(termFrom, path, termTo)` that replaces a subterm of `termFrom` identified by `path` with `termTo`. For example, `replace`$(t, \langle 1 \rangle,$ `named("Bar")`$) =$ `app(named("Foo"), named("Bar"))`.

This can be extended to a function `freshenAll(paths, term)` that accepts a set of paths that do not prefix each other and replaces the subterms each of them identify in `term` with a fresh unification variable. For example, the result of `freshenAll`$(\{\langle 0 \rangle, \langle 1, 2, 0 \rangle\}, t)$ is the term `app`$(\alpha,$ `forall("x", named("Type"), arrow`$(\beta,$ `named("Type")))`$)$.

We define the path partitioning of a term as a triple of sets of paths, $\langle P_s, P_1, P_2 \rangle$, such that:

- $\langle\rangle \in P_s$
- P_s, P_1, and P_2 are disjoint.
- $P_s \cup P_1 \cup P_2 \cup \{p' \mid p \in (P_1 \cup P_2), p \text{ strictly prefixes } p'\}$ is the complete set of paths in the term.
- For any path p in $P_s \cup P_1 \cup P_2$, no path in $P_1 \cup P_2$ strictly prefixes p.

As an example, one valid path partitioning of t is $\langle \{\langle\rangle, \langle 1 \rangle, \langle 1, 0 \rangle, \langle 1, 2 \rangle, \langle 1, 2, 1 \rangle, \langle 1, 2, 1, 0 \rangle\}, \{\langle 0 \rangle, \langle 1, 1 \rangle\}, \{\langle 1, 2, 0 \rangle\} \rangle$. If we color the subterms of t identified by paths prefixed by paths in P_s black, P_1 red, and P_2 blue, then we get:

```
app(named("Foo"),
    forall("x", named("Type"), arrow(lexical("x"), named("Type"))))
```

Finally, we define an auxiliary function `alpha(names, term)` that alpha-converts a term according to a source of names. We will leave the details of sources of names abstract for the time being, beyond asserting the existence of at least one deterministic source of names N_d and two unpredictable sources $N_{u,1}$ and $N_{u,2}$. Note that if `eq`(t_1, t_2) is syntactic comparison of terms, `eq(alpha`(N_d, t_1)`, alpha`(N_d, t_2)`)` is comparison modulo alpha. We abbreviate this `alphaEq`(t_1, t_2).

With these preliminaries, we can define our property.

Property: For any ground term t and path partitioning on it $\langle P_s, P_1, P_2 \rangle$, let

- t_1 be $\texttt{alpha}\big(N_{u,1}, \texttt{freshenAll}(P_1, t)\big)$,
- t_2 be $\texttt{alpha}\big(N_{u,2}, \texttt{freshenAll}(P_2, t)\big)$, and
- t_u be the result of unifying t_1 with t_2.

Then, $\texttt{alphaEq}(t, t_c)$.

The main complexity in implementing this as a property test is in generating good random path partitionings. If a path that prefixes many others is added to P_1 or P_2, the choices available to add further paths to the other are limited.

On the other hand, we do need to ensure that this occurs sometimes; if every path in P_1 and P_2 doesn't strictly prefix any others, constructors that take term arguments will never be unified.

The overall schema of our path partitioning generation process looks like the following:

1. We start with four sets, P_c, P_r, P_1, and P_2.

   - P_c is the set of candidate paths, from which paths are moved to P_1 and P_2. Initially, it contains all valid paths except the empty path.
   - P_r is the set of removed paths, into which paths that cannot be moved to P_1 and P_2 are moved. Initially, it contains only the empty path.
   - P_1 and P_2 are the corresponding sets of paths from the eventual path partitioning. Initially, they are both empty.

   Throughout the generation process, we maintain the invariant that $P_c \cup P_r \cup P_1 \cup P_2$ is equal to the full set of valid paths.

2. We repeatedly move paths from P_c to P_1 or P_2, then move all paths it strictly prefixes and that strictly prefix it from P_c to P_r.

3. At any point, we can stop moving paths and extract a path partitioning $\langle P_s, P_1, P_2 \rangle$. P_s can be uniquely determined by taking all the paths that strictly prefix any path in $P_1 \cup P_2$.

   TODO: Prove that we always can do this. We probably need a stronger invariant to do so…

Within this framework, we just need a procedure from to choose from paths in P_c, and a means of deciding when to stop choosing paths.

NR: Is this something that's typically solved with game theory or something? I feel like this can be modeled as a game, where taking the last valid path from P_c makes you lose. If we can model perfect play, we do that, randomly introducing blunders. On the other hand, I don't know game theory, so I'm going to do the other thing and trust in AFL to explore the state space well.

The simplest thing to do, and the one we actually do, is to repeatedly choose random paths from P_c. The stopping criterion is somewhat trickier. We might miss some cases if we always move paths until P_c is empty.

To get a good balance of P_1 and P_2 cardinalities, we use a probabilistic stopping criterion. Before we start the path partitioning generation process, we designate a random path in P_c as our sentinel path. We only move paths until the sentinel path is no longer in P_c.

TODO: Do an actual analysis of the behavior we get here.