


Spring 2022

The Algebra of Type Unification

Verity James Scheel
Bard College

Follow this and additional works at: https://digitalcommons.bard.edu/senproj_s2022

 Part of the [Logic and Foundations Commons](#), and the [Programming Languages and Compilers Commons](#)



This work is licensed under a [Creative Commons Attribution-Share Alike 4.0 License](#).

Recommended Citation

Scheel, Verity James, "The Algebra of Type Unification" (2022). *Senior Projects Spring 2022*. 230.
https://digitalcommons.bard.edu/senproj_s2022/230

This Open Access is brought to you for free and open access by the Bard Undergraduate Senior Projects at Bard Digital Commons. It has been accepted for inclusion in Senior Projects Spring 2022 by an authorized administrator of Bard Digital Commons. For more information, please contact digitalcommons@bard.edu.

The Algebra of Type Unification

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Verity James Scheel

Annandale-on-Hudson, New York
May, 2022

Abstract

Type unification takes type inference a step further by allowing non-local flow of information. By exposing the algebraic structure of type unification, we obtain even more flexibility as well as clarity in the implementation. In particular, the main contribution is an explicit description of the arithmetic of universe levels and consistency of constraints of universe levels, with hints at how row types and general unification/subsumption can fit into the same framework of constraints. The compositional nature of the algebras involved ensure correctness and reduce arbitrariness: properties such as associativity mean that implementation details of type inference do not leak in error messages, for example. This project is a discovery and implementation of these ideas by extending the type theory of the Dhall programming language, with implementation in PureScript.

Contents

Abstract	iii
Acknowledgments	vii
1 Introduction	1
1.1 Type Theory	3
1.1.1 Consistency of a Type Theory	4
1.2 Tools and Methodology	5
2 Background and Setup	7
2.1 Datatypes	7
2.2 Semilattices	8
2.2.1 Monoids	8
2.2.2 Partial orders	9
2.3 Abstract Syntax Trees	10
2.4 Variables	10
2.4.1 Metavariables	11
3 Type Inference	13
3.1 Intrinsic vs Extrinsic Typing	14
3.2 Constraints	15
3.2.1 Polymorphism	17
4 Universes	19
4.1 The Algebra of Universe Levels	20
4.2 Universe Levels in Judgments	23
4.3 Normal Form for Universe Levels	25

4.3.1	Normal Form Without Impredicative Maximum	25
4.3.2	Normal Form With Impredicative Maximum	26
4.3.3	Implementing Impredicative Maximum Through If	30
4.4	Relating Universe Levels	31
4.4.1	Relations Without Impredicative Maximum	32
4.4.2	Relations With Impredicative Maximum	33
4.4.3	Proofs	35
4.5	Constraining Universe Levels	36
4.5.1	Consistency	38
5	Row Types	41
6	Unification	45
7	Properties	49
8	Future Work	51
	Appendices	55
A	Judgments	55
A.1	Syntax	55
A.2	Substitution	57
A.3	Typing	59
A.3.1	Builtins	59
A.3.2	Functions and Variables	59
A.3.3	Rows	60
A.4	Unification/Subsumption	60
A.5	Evaluation	63
B	PureScript Reference	65

Acknowledgments

I would like to thank my advisor, Bob McGrail, for giving me the latitude and encouragement to explore these topics on my own and with his guidance.

Many thanks to Gabriella Gonzalez for not only creating Dhall and maintaining a welcoming community around it but also being supportive of my exploration and meeting with me several times to discuss ideas.

I would also like to thank other members of the Type Theory community for short but interesting conversations each: Robert Harper, Reed Mullanix, Callan McGill, Asad Saeeduddin, Philippa Cowderoy, Gabriel Scherer, and more.

1

Introduction

The Dhall language is designed to be a straightforward, strongly-typed programming language for specifying and generating configurations for system software. Despite having sophisticated type system features such as dependent types, its standard is kept simple for ease of implementation. This is done by requiring each expression to contain extra type information, which is often redundant and tedious for the user to specify. In particular, lambda abstractions have to specify their input type, even though it can often be figured out from context:

```
\(n : Natural) → n + 1
```

In this example, the user has to specify that the variable `n` has type `Natural`, even though that is the only type that makes the expression typecheck: since the addition operator `+` in Dhall is not overloaded, it can only take two arguments of type `Natural`! However, because this information requires looking at context (the usage of `n` as an argument to `+`), it is not covered by the current rules of type inference in Dhall.

Even before inferring whole types, though, there are two subproblems that must be tackled: universe levels and row types. These are aspects of types that can be detached from the structure of the types, and analyzed on their own.

As it stands, the user has to commit to exactly what fields must be present in records and unions, as in the following example:

```
\(r : { x : Natural, y : Natural }) → r.x + r.y
```

This function can only take a record with fields `x` and `y` (of the appropriate type), even though any record with additional fields would work just as well! A formalism of row types will loosen this restriction and allow a most general type to be given to functions like this that will encompass all potential usages.

Universe levels will be introduced later, but they suffer from similar problems. As one example, `Universe 0` here can be replaced with any other universe to produce a valid identity function, but without universe polymorphism these identity functions will all require separate definitions, for `Universe 1` and `Universe 2` and so on:

```
\(T : Universe 0) → \(v : T) → v
```

This project will increase the flexibility of Dhall while still keeping the same philosophy of straightforward type inference rules. In particular this means adding *universe and row polymorphism* plus *general type unification* to the language. Universes are a technical detail of type theory: they are what allow types to be first-class, however, to ensure the theory remains consistent, they require some bookkeeping which is of little interest from a programmer's perspective. Rows are certainly more interesting from a programmer's point of view: they allow coding to open interfaces of data addressed by labels, but they have their own challenges in bookkeeping. In fact, these concepts are already lurking in the existing rules for Dhall, and this project is allowing them to come into their own as concepts represented within the type theory in their own right.

The third idea is the main motivation for this project: general type unification. In order to allow the programmer to omit more types while writing Dhall, it must be possible to infer what type should have been written in the program, and this is done through unification. Every omitted type starts off as an unknown type, and the task of unification is to stitch together what partial information is known about a type from its occurrences scattered throughout the program. But to support the features already in the Dhall language, in this setting of partially unknown types, more or less necessitates the introduction of universe and row variables. And polymorphism is the natural next step from there: not only will the unknown variables stand

for unique-but-unspecified types, but they will actually be able to be instantiated differently across different call sites.

The goal is to maintain bottom-up inference rules in this description of type unification of Dhall. Traditional unification algorithms in major compilers traverse through program source in a linear fashion, mutating the “current” state of unification variables as they go. This means that information cannot be untangled from the evaluation order the compiler takes, and so the error messages that occur differ depending on program order. By maintaining the independence of parallel branches of code, the new ideas in this project ensure that errors remain predictable and clear.

The ideas contained in this project should give clearer type errors than both bidirectional type-checking and the usual unification algorithms (which silently mutate unification variables), and it should keep evaluation safe when given partial type information.

1.1 Type Theory

Type theory studies programs (in the broadest sense) by giving types to expressions in a compositional manner. Type theories are set up as a system of formal judgments that give meaning to programs, considered as terms in some language. Terms are ascribed types in typing judgments, and then evaluation rules describe how terms reduce to other terms in order to run the program. The primary judgment has the form $t : T$ saying that term t has type T . Informally we might say that term t “lives in” type T . It may be thought of as analogous to $t \in T$ in the language of set theory, and indeed, older literature often uses this notation, even though types and sets are conceptually quite different.

While the rules are formally laid out in a logical framework (the metalanguage), the rules often fit well into a computational framework. The process of checking whether a term has a particular type is called type checking. The process of coming up with a type for a term is called type inference. And of course evaluation often has computational meaning, although it

is typically specified as a term rewriting system that need not terminate and may not even be confluent (though these are both desirable properties).

The difference between type inference and type checking is that the former must come up with the type of an expression, whereas the latter is given the type of an expression and has to verify that it does indeed have the type. In some type systems, there is a significant difference between these modes: if the same expression could be assigned different types, type checking has more information to nudge the types in a particular direction, while type inference sort of has to make a guess as to which is intended. They certainly should be compatible in the sense that an inferred type should also satisfy type checking, but type checking in general may give different results. However, for the purposes of this research, type inference is primary, and type checking is implemented in terms of type inference, so they may be conflated. Additionally, since there is no term for a program that carries out type inference, “typechecker” covers both.

1.1.1 Consistency of a Type Theory

A key property of type theories is that of consistency. There are two forms of consistency. As a computational calculus, the evaluation rules of a type theory would be regarded as *inconsistent* if all terms were equatable under the rules of evaluation. This is the sense in which untyped lambda calculus is consistent. As a system of logic, however, the more relevant notion is that a logic system is consistent when not all types are inhabited by terms. (In considering propositions as types, this corresponds to not all propositions being provable.)

Most type theories have one or several types that, if inhabited, would imply all other types are inhabited. For example, in Dhall, the empty union type \diamond would be one example of such a type, since a function `forall (T : Type) → \diamond → T` is derivable in Dhall. By flipping those arguments, `\diamond → (forall (T : Type) → T)` suggests another type that is in fact equivalent: the type `forall (T : Type) → T` obviously implies all other types. These “false” types, then, must be uninhabited in a consistent type theory, and it is sufficient to prove that one is uninhabited.

The goal of consistent type theories, then, is to establish a typing system that preserves this form of consistency.

This consistency is also important from a programming perspective, as it relates to termination of evaluation. There is no normal form for false types, so if there is a term that inhabits them, it must have an infinite chain of reductions, demonstrating that evaluation is not terminating if the theory is inconsistent.

Though Dhall has a formal standard specifying its type theory (with typechecking and evaluation semantics), it does not currently have a proof of consistency. However its rules are based on systems known to be consistent, and the extensions in this paper are simple extensions to that, with constraint solving for metavariables, subsumption, and let-polymorphism. In particular, let-polymorphism should be a conservative extension, since every term written with a polymorphic let can instead be written with the substitution applied directly.

1.2 Tools and Methodology

The gold standard for research like this would be an implementation of the ideas and corresponding proofs formalized in a proof assistant, like Agda or Lean. However, this would be too ambitious: computer-assisted proofs are notoriously exacting, frustrating, and difficult to produce. So for this project, the scope is more modest and colloquial: an implementation in PureScript along with informal proofs in the language of common mathematics practice.

PureScript is a functional programming language that is quite similar to Haskell, but with strict evaluation and compilation to JavaScript. Since PureScript is much newer than Haskell, it has the chance to revisit some of Haskell's design decisions. One of PureScript's innovations is the addition of row types. However, the row types for Dhall discussed in this project differ significantly in implementation and scope from PureScript's, although the basic ideas are similar.

PureScript is great for specifying executable code, but there's an impedance mismatch between PureScript and mathematical practice. PureScript, for the most part, has a concrete

syntax for its datatypes, which is great for clarity of algorithms, but it lacks the ability to form subtypes, which complicates proofs of correctness. For example, from the type of lists in PureScript, one cannot formally construct a type of *ordered* lists in PureScript. Instead, properties like this must be maintained as informal invariants of PureScript programs instead of being bundled into the datatypes and checked for type safety.

On the other hand, set theory (the supposed language of mathematical practice) is great at forming subtypes (i.e. subsets), but is less clear with inductive types. In fact, many things that PureScript models with inductive types (such as the `SemigroupMap` type of unordered associative lists, considered as monoids under keywise appending) are best denoted by very different objects in set theory (like functions of finite support). Similarly, there is also differences of vocabulary between Dhall and PureScript, the same concept goes by different names, for example, `Text` in Dhall is `String` in PureScript.

2

Background and Setup

2.1 Datatypes

Modern programming languages feature inductive types, which are least fixed points of specified constructors. In their simplest form they are called Algebraic Data Types (ADTs), which are what PureScript supports. ADTs encompass both sum types and product types, and more general recursive types.

Product types have one constructor and multiple fields:

```
data Bounds = MkBounds (Max Int) (Maybe (Min Int))
MkBounds :: Max Int -> Maybe (Min Int) -> Bounds
```

This datatype has one constructor `MkBounds` which has the type shown above.

Records in both PureScript and Dhall allow naming the fields of a product type:

```
{ min :: Max Int, max :: Maybe (Min Int) }
```

Sum types have multiple constructors:

```
data Maybe a = Nothing | Just a
Nothing :: forall a. Maybe a
Just :: forall a. a -> Maybe a
```

This datatype has two constructors: `Nothing`, a constructor with zero arguments, and `Just`, a constructor with one argument.

Union types in Dhall allow anonymously creating this pattern with named fields, and a library provides `Variant` for PureScript which serves a similar purpose:

```
< Nothing | Just : a >
Variant ( Nothing :: Unit, Just :: a )
```

One special type in PureScript is `Map`, and its companions `SemigroupMap` and `Set`. Internally they are implemented as balanced trees, but this representation is not visible to the user: all that matters is that they are sorted collections indexed on a fixed type of key. Mathematically they can be viewed as partial functions from the key type to the value type. The semigroup operation on `SemigroupMap` appends two maps in the obvious keywise manner: if the key is present in both, append the two values, otherwise take the single value that is present.

2.2 Semilattices

While groups are optimal for capturing the symmetries of various systems, semilattices are ideal for tabulating knowledge obtained incrementally. Therefore, semilattices are one tool that regularly show up in type inference.

Semilattices can be viewed from an algebraic perspective, as idempotent commutative monoids, and also from an order-theoretic perspective, as a partially-ordered set with finite joins – that is, least upper bounds of finite sets. (Dually they can be thought of as having finite meets/greatest lower bounds, but we will solely consider semilattices as join-semilattices in this work.)

2.2.1 Monoids

Monoids are sets with one associative binary operation that has an identity. That is, they are semigroups with an identity, or groups without inverses.

1. Associativity:

$$(xy)z = xyz = x(yz)$$

2. Two-sided identity:

$$ex = x = xe$$

Most monoids considered here additionally have the property that their identity is adjoined: the identity only factors trivially as $e = ee$ (this is true for semilattices in particular, since if e factors as xy , then $x = xe = xxy = xy = e$ and likewise $y = e$ by idempotence). This means these monoids are really semigroups reflected through the adjunction that freely adjoins an identity element.

Monoids are ubiquitous in programming. For example, strings form a monoid under concatenation. But this is really an instance of a more general fact: lists form the *free monoid*, and strings are abstractly just lists of characters.

Natural numbers form monoids in several ways: under addition (with identity 0), under maximum (also with identity 0), under multiplication (with identity 1), under *lcm* (also with identity 1), and other operations as well. Note that these three operations are commutative, but only maximum and *lcm* are idempotent. This idempotency is the key to semilattices.

Finite sets form the *free semilattice*, with the monoid operation being set unions. They can be constructed by quotienting lists by commutativity and idempotence, or as a subtype by choosing an ordering and requiring that the underlying list appears in sorted order without duplicates.

This is the key underlying reason why we often think of operations like maximum and *lcm* as operating on finite sets.

So when asking how to keep track of information like constraints during typechecking, the most general answer will always be “Freely, by tabulating what constraints have occurred in a finite set”. But there often is more structure to be noticed, and it can be factored into a more specific algebraic structure (carried in some datatype) that precisely captures how constraints interact.

2.2.2 Partial orders

Partial orders have the following axioms:

1. Reflexivity:

$$x \leq x$$

2. Transitivity:

If $x \leq y$ and $y \leq z$ then $x \leq z$.

3. Antisymmetry: If $x \leq y$ and $y \leq x$, then $x = y$.

From a commutative idempotent monoid, we can form a partial order by taking $x \leq y$ to be defined as $xy = y$, which can intuitively thought as y absorbing x under the monoid operation. Transitivity and antisymmetry are immediate from the semigroup properties, and reflexivity comes from idempotence. The identity of the monoid becomes the least element under this ordering. This ordering additionally respects the monoid operation: if $x \leq y$ and $u \leq v$ then $xu \leq yv$ since $xuyv = xyuv = yv$ by commutativity.

2.3 Abstract Syntax Trees

Abstract syntax trees are the bread and butter of computer science research. By thinking of syntax as a tree, the recursion patterns of algorithms (particularly typechecking) is clarified.

All of the algorithms, such as typechecking and evaluation, are best thought of as operating on the abstract syntax. Substitution is particularly straightforward: other than the cases that mention variables, the rest of substitution is the obvious recursion down the syntax tree.

2.4 Variables

Variables are a big topic in programming language design, and this is compounded when formalizing programming languages.

Variables are placeholders for values, so .

Variables in the programming language.

Formalizing variables in a rigorous way is a surprisingly hard issue. The main issue is resolving variables when there are multiple with the same name in scope. If one variable

binding clobbers another already in scope, with no way to disambiguate, then the semantics of the program could drastically change! It can be responsible for subtle bugs in theory and implementation.

Nevertheless, there are many known solutions, such as capture-avoiding substitution, which renames bound variables during the process of substitution, and de Bruijn indices, which replace variables with numbers that track levels of scope, thus allowing `.` We will gloss over the issue here, but in the Dhall standard it is formalized with variable shifting that combines the power of de Bruijn indices with the convenience of named variables. Adding this method of variable shifting onto existing rules is a straightforward process, since it mirrors the obvious structure of how variable contexts are already embedded in the rules.

2.4.1 Metavariables

The term “metavariable” is somewhat ambiguous: It could refer to variables in the metalanguage, like the placeholders for variable names, other syntax fragments, or complete expressions that are used in rules. However, this is not usually talked about, because we are not analyzing the metalanguage, we are employing it.

What they usually refer to, then, is placeholders for undetermined structure. That is, they are implicitly existentially quantified and may have constraints placed on their value, potentially including an exact value being determined later.

Because these metavariables are existentially quantified, with no particular scope nor explicit abstraction/instantiation rules, they are global variables. In order to support polymorphism, therefore, their scope must be contained. We will see rules for determining their scope in this framework of metavariables and constraints during type inference.

Implicit Metavariables and Elaboration

The goal is for the user to not have to supply universes and row types, since to produce the most general type, they could all be metavariables. However, users may want to specify more restric-

tive types – if a definition does not need to be polymorphic and would produce more confusing errors, the user could specify a concrete universe level for it, also for efficiency reasons.

3

Type Inference

Type theory rules are typically written in horizontal bar style, writing assumptions above the line that are required to deduce the judgment below the line.

Most judgments in type theory take place in a context, traditionally denoted Γ . This tracks variables in scope and associates them with their types, and sometimes values (for let-bound variables). Thus when examining the body of a `forall` or lambda term, the context is extended with the variable name and the type declared by the `forall` or lambda. And when examining the body of a `let` term, the context is extended with not only the variable name and type, but also its declaration value (and later we will add more information here to support polymorphism).

The main judgment is denoted $\Gamma \vdash t : T$ and reads “term t has type T in context Γ ”.

Judgments in this sense belong to a logical metatheory, but they may be given a computational interpretation by a type inference algorithm. In the computational interpretation, Γ and t are inputs to the algorithm, and the result is the inferred type T or an error if no proof tree could be constructed for it.

Type inference usually satisfies particular nice properties. For example, if a piece of syntax typechecks in an environment, every sub-piece of syntax also typechecks in same environment, but extended with variables to reflect the deepening of scope.

Another property, useful for implementation, is that each piece of syntax typically only has one rule that could apply to it, so there is no ambiguity. This is called syntax-directed type-checking.

Besides type-theoretical judgments, there are other side-conditions that may appear as assumptions for judgments. Normally they involve values that are static in the source code, so the assumptions are trivially checked immediately while applying the rule.

The goal of this project is to show how these side-conditions can instead be deferred, with less of it being known statically and more being figured out during type inference. In doing so, the side-conditions need to be more tightly integrated with the presentation of the type inference judgments. Now type inference will produce constraints, and these constraints need to be tabulated and checked for consistency. In particular, universe levels will produce arithmetical constraints that are essential to ensuring termination of program evaluation, and row types will produce other kinds of constraints to ensure that when a record has its field accessed, that label definitely exists in the record (and with the right type).

The unification judgment will be written $A \equiv B \mapsto C$ for the unification of A and B resulting in a new unified term C (along with constraints to make them unify), and computationally this is a part of typechecking.

3.1 Intrinsic vs Extrinsic Typing

Each rule for type checking judgments can be viewed as an introduction rule for a formal deductive derivation. A typing derivation, then, is a tree of these judgments arranged in the appropriate way.

There are two extremes: intrinsic typing and extrinsic typing.

Intrinsic typing says that the syntax contains enough information about types that there is only one possible typing derivation for a particular syntax tree. For example, it is common to have lambda terms require a type for their argument. The role of type inference is then to verify that the details check out, and to find the unique type associated with the expression in the end.

Extrinsic typing at the other extreme says that programs exist on their own, independent of types, and types are imposed on top of existing programs via formal typing derivations, which may now involve creativity and choices not written into the structure of the program! For example, an untyped lambda term like $\lambda x \rightarrow x$ can be compatible with many types, like `Natural` \rightarrow `Natural` or `Text` \rightarrow `Text`, since the program itself does not impose one choice of type on its argument and is consistent with all choices.

This project then starts from the intrinsic point of view, and weakens the static requirements of programs slightly to approximate the freedom of extrinsic typing, while still retaining the structure of the intrinsic. In particular, one goal is that once all constraints are satisfied, an intrinsically-typed syntax tree could still be produced.

3.2 Constraints

Now we represent constraints explicitly during type inference. Constraints should be thought of as finite sets of *not incompatible* atomic constraints. These atomic constraints will be constraints on universe levels (like $u \leq \max(v, 2)$), on row types (like $r_1 \bowtie r_2 = r_3$), or a unification constraint between terms.

The new judgment looks like $\Gamma \vdash e : T \Leftarrow C$ and is read as “In context Γ , expression e is inferred to have type T once the constraints C are satisfied”. Algorithmically, Γ and e are the inputs to the type inference procedure and T and C are the outputs. This might be surprising under an interpretation of the judgment as “ e has type T if C is satisfied”. But it is really making the claim “ e can *only* have type T , and that *only* occurs when C is satisfied”.

Note

Looking forward, it would be cool to see if Γ could also be an output of the algorithm. This relates to a concept called principal typings. See the conclusion for more.

Of course we will have judgments that are trivially true, returning the trivial constraint \emptyset , like the following:

$$\frac{}{\Gamma \vdash 0 : \mathbf{Natural} \Leftarrow \emptyset}$$

Other judgments will require combining the constraints produced, under constraint union:

$$\frac{\Gamma \vdash e_1 : \mathbf{Text} \Leftarrow C_1 \quad \Gamma \vdash e_2 : \mathbf{Text} \Leftarrow C_2}{\Gamma \vdash [e_1, e_2] : \mathbf{List Text} \Leftarrow C_1 \cup C_2}$$

Note that we only want to mention constraints that are satisfiable (or not known to be unsatisfiable!). That is, we want type inference to stop on `assert : 2 + 2 == 5` instead of continuing with the unsatisfiable unification constraint $2 + 2 \equiv 5$. So every time we mention constraints in a judgment, there is an implicit check that they are still satisfiable. (Unfortunately, this is an undecidable problem in general, given equality of open terms is undecidable, so we will have to settle for ensuring they are *not unsatisfiable*.)

As written, the Dhall standard suggests inferring the type of the binding, and then substituting it in and inferring the type of the resulting expression:

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2[x := e_1] : T_2}{\Gamma \vdash \mathbf{let } x = e_1 \mathbf{ in } e_2 : T_2}$$

However, in practice what most implementations do is to extend the context by including both the inferred type of the variable and its defined value:

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, (x := e_1 : T_1) \vdash e_2 : T_2}{\Gamma \vdash \mathbf{let } x = e_1 \mathbf{ in } e_2 : T_2[x := e_1]}$$

Notice that this requires the substitution be performed in the output type now, but this is generally less work – in particular, for value-level variables that do not occur in the type! Maybe the output substitution isn't necessary if the type judgment is well-crafted, i.e. producing a normalized type, but it would still require shifting to keep track of variables.

Note

Under lazy evaluation, if the variable x is never used, it will not be evaluated. Thus it would still be safe (with respect to evaluation and type-safety) to wholly omit the first assumption $\Gamma \vdash e_1 : T_1 \Leftarrow C_1$, since it is unnecessary when x does not appear in e_2 and it would merely be deferred to the usage sites when x does occur. However, this is a bad idea for a couple reasons: First it breaks the formal assumption that if an expression typechecks, its subexpressions typecheck (in the appropriate context). Second, it would be surprising for users of the language who rely on this property, even informally through their intuition but also formally. As a particular example, assertion expressions are often used in let bindings to formally assert properties of programs. These assertion expressions are never evaluated: there is no way to force their evaluation in Dhall! Third, it also makes error messages more unpredictable: the error would be deferred and look like it was coming from inside e_2 when it really occurred in e_1 , which just happened to get embedded into e_2 .

3.2.1 Polymorphism

These judgments do not get us polymorphism yet, though, because metavariables are global. Instead, we need to track what metavariables to generalize over, and then instantiate them at use sites.

There are two ways to do this, corresponding to the two styles of let-inference above: If we are doing inference on the substituted expression $e_2[x := e_1]$, we can simply make sure that the metavariables are generalized for each instantiation, which should be direct (once we are sure what metavariables to generalize). The second style requires a bit more work, but it amounts to keeping track of the unresolved constraints on the generalized metavariables, and then generalizing.

We will prefer the second style for the same reasons: the inference work is memoized, and the instantiating work is substantially smaller.

For now, we will only make let-bound variables polymorphic. Higher-order polymorphism, with polymorphic function arguments, will be more difficult (and less clear if it is consistent).

In order to do so, we want to keep a whole type inference constraint in context:

$$\frac{\Gamma \vdash e_1 : T_1 \Leftarrow C_1 \quad \Gamma, (x := e_1 : T_1 \Leftarrow C_1) \vdash e_2 : T_2 \Leftarrow C_2}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : T_2[x := e_1] \Leftarrow C_1 \cup C_2}$$

The magic happens in the variable judgment:

$$\frac{vs \text{ to generalize over with corresponding fresh } us}{\Gamma, (x := e : T \Leftarrow C), \Delta \vdash x : T[vs := us] \Leftarrow C[vs := us]}$$

When we see a variable, we look it up in the context, complete with its type and constraints, and then we reinstantiate the generalizable metavariables while copying the type and constraint.

Note

We may need to inspect e to catch some metavariables to be generalized in the constraints, but maybe they should be already discarded. That is, if there are metavariables that occur in e but not T , we should try to eliminate them from the constraints.

How do we know what metavariables to generalize over? By examining the context and tracking which metavariables already occurred in lambda-bound places.

The distinction is that variables that are let-bound never introduce new bound metavariables, since the metavariables they would introduce are instead generalized over; but lambda-bound variables do bind their metavariables, since we do not yet support polymorphism for them.

$$\overline{\text{metavariables}(e) = \{ \dots \}}$$

$$\overline{\text{bound}(\Gamma, x : T) = \text{bound}(\Gamma) \cup \text{metavariables}(T)} \quad \overline{\text{bound}(\Gamma, x := e : T \Leftarrow C) = \text{bound}(\Gamma)}$$

$$\frac{vs := \text{metavariables}(e : T) \setminus \text{bound}(\Gamma) \quad us := \text{fresh}(vs)}{\Gamma, (x := e : T \Leftarrow C), \Delta \vdash x : T[vs := us] \Leftarrow C[vs := us]}$$

4

Universes

How do you study something in type theory? By giving it a type! Universes are a way to give types to types, and they would be uninteresting if not for some difficulties that arise in ensuring their consistency.

The naïve way of doing it would be to say that all types live in a single universe, call it **Type**. This universe is in fact a type, so it must be the case that **Type** : **Type**. However, this “**Type-in-Type**” rule makes most type theories inconsistent, due to results such as Girard’s paradox and its simplification as Hurkens’ paradox [3]. These are in some ways analogous to Russell’s paradox: just like there can be no set of all sets in a consistent set theory, there can be no type of all types in a consistent type theory. In type theory, the inconsistencies are visible as terms that can be ascribed types, but do not reduce to any normal form in a finite number of steps. This particular type of inconsistency was exhibited in an issue on the Dhall repository when impredicativity was allowed for *two* universes, which is enough to exhibit the above paradoxes, but the issue was quickly fixed [1].

The solution, therefore, is to stratify types into levels. Only *small* types can live in **Type**, while **Type** itself (and other “large” types) will live in **Kind** and **Kind** lives in **Sort**, and so on. In fact it is customary to assume an infinite hierarchy of universes indexed by natural numbers, where

`Type = Universe 0` denotes the smallest, then `Kind = Universe 1`, `Sort = Universe 2`, and the rest are unnamed but exist as `Universe 3`, `Universe 4`, and so on.

However, these universe levels cannot be understood as natural numbers internal to the type theory. They must be accorded special status, because abstracting over them works differently to normal values and also because they should not support all operations that natural numbers do. In fact, four operations suffice as we will see, plus a fifth to obtain a normal form.

It is conventional to elide concrete universe levels from programs, in so-called “typical ambiguity”, and instead assume that there is a consistent assignment of levels that makes it work. This is the role of metavariables here, to keep track of universes with various constraints placed on them during typechecking. The question is how to design a type theory to check consistency of universes in as much generality as possible, and much has been written about this problem, including papers on similar type theories [2]. Here we just dive straight in to the arithmetic and give explicit description of most of the details necessary to check universe constraints.

4.1 The Algebra of Universe Levels

The actual algebraic language that expresses universe levels only has four operations: nullary zero, unary successor, and two binary operations maximum and impredicative-maximum. Thus this language only allows addition by fixed natural numbers: adding two universe levels together is not meaningful.

There are actually three related languages under discussion here: \mathcal{L} , the language of universe levels with only maximum, no impredicative maximum; $\mathcal{L}_{\text{imax}}$, extended with the impredicative maximum; and \mathcal{L}_{if} , with an alternative binary operator that is used to simplify normal forms for impredicative maximum but does not appear directly in the typing judgments.

Obviously we will use natural numbers to denote the appropriate sequence of successors and zero, and addition by a natural number represents the appropriate sequence of successors. The issue of variables in the metatheory already rears its head here: by writing $u + k$ we mean that

u is a variable (or possibly a whole universe level expression), and k is a fixed natural number to shift u by.

For clarity, we will usually denote maximum of u and v by $\max(u, v)$ and their impredicative-maximum by $\text{imax}(u; v)$, but sometimes we will leave the function labels off and treat the comma and semicolon as binary operators, with the comma having higher precedence. Obviously the ordinary maximum will be associative, commutative, idempotent, with identity 0, so there is no disambiguation required. The impredicative maximum, however, is neither associative nor commutative, only idempotent, and so by convention it will be regarded as left-associative. That is, $(u, v; x, y)$ reads as $\text{imax}(\max(u, v); \max(x, y))$ and $(u; v; w)$ as $\text{imax}(\text{imax}(u; v); w)$.

We define $u \leq v$ by $\max(u, v) = v$ as is standard.

Axioms:

1. Successor is injective:

$$u = v \iff u + 1 = v + 1$$

2. Maximum is commutative:

$$\max(u, v) = \max(v, u)$$

3. Maximum is associative:

$$\max(u, \max(v, w)) = \max(\max(u, v), w)$$

4. Maximum is idempotent:

$$\max(u, u) = u$$

5. Zero is least:

$$\max(0, u) = u$$

6. Successor distributes across maximum:

$$\max(u, v) + 1 = \max(u + 1, v + 1)$$

7. For $\mathcal{L}_{\text{imax}}$:

(a) Impredicative maximum:

$$\text{imax}(u; 0) = 0$$

(b) Non-impredicative maximum:

$$0 < v \implies \text{imax}(u; v) = \max(u, v)$$

8. For \mathcal{L}_{if} :

(a) If zero:

$$\text{if}(u; 0) = 0$$

(b) If nonzero:

$$0 < v \implies \text{if}(u; v) = u$$

(c) Definition of impredicative maximum in terms of if:

$$\text{imax}(u; v) = \max(\text{if}(u; v), v)$$

Some key consequences derived from the above fundamental axioms:

$$1. 0 \leq u$$

$$2. u \leq \max(u, v)$$

$$3. u \leq v \iff u + 1 \leq v + 1$$

$$4. \text{imax}(u; \max(v, w)) = \max(\text{imax}(u; v), \text{imax}(u; w))$$

$$5. \max(\text{imax}(u; v), \text{imax}(u; w)) = \text{imax}(\max(u, v); w)$$

$$6. \text{imax}(u, \text{imax}(v, w)) = \text{imax}(\max(u, v); w)$$

In the following, when we speak of a “model”, we mean an assignment of variables to natural numbers. The set of all models is the space which gets carved out by constraints: certain models are compatible with some constraints while others are not. In particular, the challenge is to narrow down .

4.2 Universe Levels in Judgments

Here we go over how the universe levels are used in judgments.

Of course the successor operation, as mentioned above, is used to find the type of a universe, and the zero is used for types like **Natural** and **Text** that automatically live in the lowest universe. The maximum operation is used to find a common universe that contains two (or more) types, particularly in constructions like record and union types.

The impredicative-maximum is used to determine what universe function types live in.

Here is how the universe level constraints come up during type inference:

Universes themselves live in the next highest universe:

$$\frac{}{\Gamma \vdash \mathbf{Universe} \ u : \mathbf{Universe} \ (u + 1) \Leftarrow \emptyset}$$

Function types live in the impredicative-maximum of their input and output universes:

$$\frac{\Gamma \vdash T_1 : \mathbf{Universe} \ u \Leftarrow C_1 \quad \Gamma, x : T_1 \vdash T_2 : \mathbf{Universe} \ v \Leftarrow C_2}{\Gamma \vdash (\forall (x : T_1) \rightarrow T_2) : \mathbf{Universe} \ \text{imax}(u;v) \Leftarrow C_1 \cup C_2}$$

That is, functions with a codomain in the lowest universe live in the lowest universe, regardless of what universe the domain is in.

Impredicativity is particularly useful in programming because it allows polymorphic functions to remain in the lowest universe, if they return data in the lowest universe. In particular, custom recursive types (analogous to the builtin **List**) require an encoding with polymorphic functions that quantify over types (e.g. Böhm–Berrarducci encoding), and impredicativity allows this to live in the lowest universe (assuming all of the data does). Most importantly, it still produces a consistent logic system and terminating programming language, though some have philosophical objections to it.

The obvious rules hold for record types, union types, and the like. In particular, these rules take the maximum of levels of their arguments.

A common additional rule is the universe cumulatvity rule, which says that any type in a particular universe also lives in all larger universes:

$$\frac{\Gamma \vdash T : \mathbf{Universe} \ u \quad u \leq v}{\Gamma \vdash T : \mathbf{Universe} \ v}$$

As stated, it means that terms no longer have a unique type and requires generating fresh variables for universe levels. Instead, a subsumption rule is incorporated in the appropriate places to emulate the cumulatvity. This is because the type information is propagated anyways.

$$\frac{\mathbf{Universe} \ u \trianglelefteq \mathbf{Universe} \ v \Leftarrow \{u \leq v\}}{\frac{A_2 \trianglelefteq A_1 \Leftarrow C_1 \quad B_1 \trianglelefteq B_2 \Leftarrow C_2}{\forall (x : A_1) \rightarrow B_1 \trianglelefteq \forall (x : A_2) \rightarrow B_2 \Leftarrow C_1 \cup C_2}}$$

The unification rule produces an equality constraint:

$$\frac{u \equiv v \mapsto w}{\mathbf{Universe} \ u \equiv \mathbf{Universe} \ v \mapsto \mathbf{Universe} \ w}$$

Because $u \equiv v$ emits as a side-condition, there's no specific expression that denotes w , so we just take $w = \max(u, v)$ as the most symmetric result, though there should be no difference returning either u or v instead.

The subsumption rule acts a lot like the unification rule in that it ensures the two types have similar structure overall, but incorporates an inequality constraint on the universe levels at the leaves according to the variance of the judgment.

Besides cumulatvity, the big challenge is polymorphism: letting the same definitions be instantiated at various universes. For built-in functions (e.g. `Natural`/`fold`) this is easy enough to postulate in the type theory (again, it requires fresh variables for universe levels), but extending this to let-bound user functions is more work, and function arguments even more so (i.e. higher-order polymorphism). Cumulatvity takes care of some examples where polymorphism would ordinarily be required.

4.3 Normal Form for Universe Levels

First we introduce normal forms for the three-operation language without `imax`, then we extend it to a normal form for the full language.

4.3.1 Normal Form Without Impredicative Maximum

The normal form for \mathcal{L} is $\max(u_1 + k_1, \dots, u_n + k_n, c)$, where $c \geq k_1, \dots, k_n \geq 0$ and the u_i occur in order. This is clearly closed under the operations in \mathcal{L} , since `max` is associative, commutative, and idempotent, and successor distributes across it. The PureScript datatype that represents the normal form is as follows:

```
data UnivMax = UnivMax (SemigroupMap String (Max Int)) (Max Int)
```

With this setup, taking the maximum is clearly just the obvious semigroup append operation.

There are some details to worry about here, though. The first is the use of integers instead of natural numbers: for one it is easier to work with integers in PureScript due to built-in support, and later, when subtracting constants from expressions based on their relationships, it will be convenient to temporarily allow negative integers to appear in constraints, although in the global picture nothing will actually be negative: no variables will be assigned negative values and no expressions will evaluate to negative values. Certainly it will be the case that all the inputs are nonnegative integers, and this can be enforced syntactically when parsing programs.

And as was discussed in the introduction, there is no way in PureScript to enforce the restriction corresponding to $c \geq k_1, \dots, k_n$. This restriction comes from the fact that variables (and all expressions) are nonnegative. So instead, there is a normalization function that sets $c' = \max(c, k_1, \dots, k_n, 0)$.

```
normalizeUnivMax :: UnivMax → UnivMax
normalizeUnivMax (UnivMax us u) = UnivMax us
  (fold1 (NonEmpty u us) ◇ Max zero)
```

Thus we see that an expression input as $\max(u+1, v+3)$ has a normal form of $\max(u+1, v+3, 3)$, just by applying the nonnegativity axiom, $\max(u+1, v+3) \geq v+3 \geq 3$. One nice property of

this normalization is that appending two normalized expressions results in another normalized expression.

4.3.2 Normal Form With Impredicative Maximum

Instead of a normal form for $\mathcal{L}_{\text{imax}}$ (which would require making somewhat arbitrary choices), expressions are written in the larger language \mathcal{L}_{if} which admits nice normal forms. One example of why this is crucial is that the expression $(c; b; a), (c; a; b) = ((c; b), b; a), a, ((c; a), a; b), b$ could have two normal forms that are equivalent up to choice of order on the variables: $(c; b; a), a, b = (c; a; b), a, b$. One of the two has to win in this case, but imax does not commute like that: only if does.

In particular, \mathcal{L}_{if} has normal forms that are the maximum of cases with values from \mathcal{L} , satisfying some normality properties. Syntactically an expression in normal form looks like $\text{max}(\text{if}(a_1; b_{1,1}; \dots; b_{1,m_1}); \dots; \text{if}(a_n; b_{1,1}; \dots; b_{1,m_n}))$ where a_i are normal forms for \mathcal{L} and $b_{i,j}$ occur in sorted order, no duplicates, plus some other conditions on how the cases interact. However, this is more clearly expressed in the PureScript type:

```
newtype UnivIMax = UnivIMax (SemigroupMap (Set String) UnivMax)
```

Again, the obvious semigroup operation represents taking the maximum of two expressions, though it may not be in normal form anymore even if its operands are.

Mathematically, this should be seen as a monotonic function from hypotheticals (sets of variables) to the above \mathcal{L} normal form. Indeed comparing any two normal forms of this type essentially requires computing that function at all the relevant hypotheticals. This PureScript function computes that by folding together all the hypotheticals that are included in the desired hypothetical:

```
foundAt :: Set String -> UnivIMax -> UnivMax
foundAt vars (UnivIMax c) = withMinimumAt vars $
  fromMaybe (UnivMax mempty (Max zero)) $
    c # foldMapWithIndex \ks v ->
      if Set.subset ks vars then Just v else mempty
```

However, for the purposes of a finite normal form, it is better to store the *minimal* amount of information to recover the function. So, along with the above conditions, there is the extra

restriction that there is no redundancy: if one hypothetical is a subset of another, then whatever constraints are forced in the smaller (general) hypothetical should not be included in the larger (more specific) hypothetical. More on that later, but for example, $\max(\text{if}(u;v), u + 1, v)$ is not a normal form: the u under the hypothetical $\{v\}$ is redundant, since $u + 1$ already occurs at the hypothetical \emptyset . The correct normal form of that expression is simply $\max(u + 1, v)$. See `reduceBySelf` below for the logic implementing this.

There are two additional details to take care of, relating to variables appearing under their own hypotheticals.

The easier detail is exemplified by the two facts that $x + 3 = \max(3, x + 3)$ but $\text{imax}(x + 3; x) = \text{imax}(\max(4, x + 3); x)$. That is, under a hypothetical that includes $\{v\}$, the variable v is (by definition) strictly positive, so the constant factor representing the minimum value needs to take that account. See `withMinimumAt` below for this step of normalization.

The more subtle detail can be seen in $\max(\text{imax}(x + 5; x), 6) = \max(\text{imax}(x + 5; x), 6, x + 5) = \max(6, x + 5)$, where the second step is handled by the first detail above (removing redundancy), but the first step requires a new approach. Whenever the variable appears in a hypothesis that also mentions it, it is added to the hypothesis that excludes it, shifted by the minimum of its observed shift and the constant factor at the reduced hypothesis. This preserves equality since there it will only be observed as zero anyways as the nonzero case is appropriately handled by the hypothesis that includes it. See `promoting` below for this step of normalization.

Again, these conditions cannot be imposed on values within PureScript's system of datatypes, but they lend themselves to being expressed as a normalization function:

```
normalizeUnivIMax :: UnivIMax → UnivIMax
normalizeUnivIMax (UnivIMax uz) =
  UnivIMax (mapWithIndex withMinimumAt uz) # promoting # reduceBySelf

withMinimumAt :: Set String → UnivMax → UnivMax
withMinimumAt ks (UnivMax us u) = UnivMax us $
  apply maybe append (Max zero ◇ u) $
    us # foldMapWithIndex \k (Max v) →
      Just (Max (v + if Set.member k ks then 1 else 0))

promoting :: UnivIMax → UnivIMax
```

```

promoting (UnivIMax uz) = UnivIMax $ append uz $
  uz # foldMapWithIndex \ks (UnivMax us _) →
    us # foldMapWithIndex \k n →
      let ks' = Set.delete k ks in
          if ks' = ks then mempty else
            case foundAt ks' (UnivIMax uz) of
              UnivMax _ n' →
                let n'' = min n' n in
                    SemigroupMap $ Map.singleton ks' $
                      UnivMax (SemigroupMap (Map.singleton k n'')) n''

-- | Remove information that is implied already.
-- | This is used for normalizing constants.
reduceBy :: UnivMax → UnivMax → Maybe UnivMax
reduceBy
  (UnivMax (SemigroupMap source) (Max sv))
  (UnivMax (SemigroupMap target) (Max tv)) =
  let
    target' = target # Map.mapMaybeWithKey \k v →
      case Map.lookup k source of
        Just v2 | v2 >= v → Nothing
        _ → Just v
  in if Map.isEmpty target' && sv >= tv then Nothing
     else Just (UnivMax (SemigroupMap target') (Max tv ◇ Max sv))

-- | Reduce by itself.
reduceBySelf :: UnivIMax → UnivIMax
reduceBySelf (UnivIMax (SemigroupMap source)) =
  UnivIMax $ SemigroupMap $ source #
    Map.mapMaybeWithKey \ks →
      reduceBy (foundBy ks (UnivIMax (SemigroupMap source)))

```

Unlike `normalizeUnivMax`, however, `normalizeUnivIMax` is not preserved by the naïve semigroup operation, since redundancy may be introduced with the expressions in combination that was not present individually.

A normal form N in \mathcal{L}_{if} is in $\mathcal{L}_{\text{imax}}$ exactly when under each nontrivial hypothetical, the variables of that hypothetical are included. This can be characterized more directly in terms of the normal form. We will see that this property is maintained throughout, to ensure the normal form of $\mathcal{L}_{\text{imax}}$ is indeed closed.

Proof of Normal Forms

To show that we have normal forms for the respective languages, for two expressions with different normal forms, we need to demonstrate a model where they differ.

For \mathcal{L}_{\max} , the proof of normal form is simple:

1. If the normal forms differ at a constant (for example, $\max(x + 1, 4)$ versus $\max(x + 1, 7)$), because the normal form guarantees that those constants are greater than the shifts of the variables, a model witnessing their difference is found by simply setting all variables to zero.
2. If the normal forms differ at a variable, a model witnessing their difference is found by setting all variables to zero except the one they differ at, which can be set to the maximum of the constants appearing in the expressions. This makes it so that that variable dominates and the difference can be observed. For example, for the expressions $\max(x + 3, 13)$ and $\max(x + 4, 13)$, they are the same for $x \leq 9$, but of course once $x > 9$ the $x + 3$ and $x + 4$ terms will dominate, showing the difference, so the model $x = 13$ certainly suffices. Note that this model works even if the variable is omitted from one side.

For $\mathcal{L}_{\text{imax}} \subset \mathcal{L}_{\text{if}}$, the proof of normal form is more intricate.

The first step is to find an *inclusion-minimal* hypothesis where they differ. This is to ensure that the difference corresponds to an actual difference in the functional realization.

The basic model is to set the variables in the hypothesis to 1, and other variables to 0, but there are three slightly different cases:

1. If the normal forms differ at a constant, then the basic model works, in particular since the normalized constant factor takes into account the basic model! (Recall that $\text{imax}(x + 3; x)$ has the normal form $\text{imax}(\max(4, x + 3); x)$.)
2. If the normal forms differ at a variable *which appears in the hypothesis already*, the model simply needs that variable to dominate like above, which again can be found by setting it to be larger than the constants found within.

3. If the normal forms differ at a variable *which is not mentioned in the hypothesis*, this variable can still be made to dominate, but it needs a little more explanation why this difference is still visible at the hypothesis including this new variable. Two examples, $\max(x + 4, 8)$ versus $\max(x + 6, 8)$, or $\max(x + 4, 8)$ versus 8 (note that since we are not in case 1, the constants must be the same in both expressions!). Thus the variable must be missing from the hypothesis with the new variable added, since otherwise normalization would move it to this smaller hypothesis.

4.3.3 Implementing Impredicative Maximum Through If

This is the part that shows that the language is closed for \mathcal{L}_{if} and $\mathcal{L}_{\text{imax}}$.

First we set up two helper functions. Then we calculate the combination of variables that have to be zero for the constant to be zero. Note that this.

An expression is zero when:

1. $\text{imax}(a; b) = 0$ iff $b = 0$.
2. $\text{if}(a; b) = 0$ iff $b = 0$ or $a = 0$.
3. $\max(a, b) = 0$ iff $a = 0$ and $b = 0$.
4. 0 is always zero, $a + 1$ is never zero.

Knowing the exact combination of variables that make an expression zero, it can be used to construct the appropriate expression out of if operations. In particular, an expression e under the disjunction of some variables vs is expressed by $\text{if}(e; vs)$ (if one of those variables is zero, then that expression is), and conjunction is further expressed by taking the maximum of those expressions (if any of the conjuncts are nonzero, then the expression is has value e). Thus it is convenient to have this information in conjunctive normal form (CNF). The disjunctions are not even needed if the expression is in $\mathcal{L}_{\text{imax}}$, so a simpler implementation can be given for that special case, though it is omitted below.

```
zeroableUnivMax :: UnivMax → Boolean
zeroableUnivMax (UnivMax us (Max u)) =
```

```

u <= zero && all (\(Max v) → v <= zero) us

skimUnivMax :: UnivMax → Maybe (Set String)
skimUnivMax t@(UnivMax (SemigroupMap us) _) =
  if zeroableUnivMax t then Just (Map.keys us) else Nothing

-- | The expression is only zero when combinations of variables are zero.
peruse :: UnivIMax → Set (Set String)
peruse (UnivIMax (SemigroupMap uz)) =
  let
    -- distr :: CNF → DNF → CNF
    distr k1s k2s =
      k2s # Set.map \k2 →
        Set.insert k2 k1s
  in uz # foldMapWithIndex \k1 t →
    -- k1 is DNF
    -- skimUnivMax is CNF
    -- we need CNF
    case skimUnivMax t of
      Just k2 → distr k2 k1
      Nothing → Set.singleton k1

ifop :: UnivIMax → UnivIMax → UnivIMax
ifop (UnivIMax (SemigroupMap us)) v
= fromMaybe uempty $
  peruse v # foldMap \ks →
    us # foldMapWithIndex \ks' u →
      Just $ UnivIMax $ SemigroupMap $ Map.singleton (ks ◇ ks') u

```

4.4 Relating Universe Levels

Relating two level expressions is one of the key components of the constraint solving.

Besides equality of normal forms (that is, equality across all models), there are three fundamental relations that work together to inform about how expressions relate across models. The simplest relation is one is always strictly less than the other: e.g. $\max(u, v) < \max(u+1, v+1)$. The next simplest relation is that one is always less than or equal to the other *and* there are models where they are equal and arbitrarily large: e.g. $\max(u, v) \lesssim \max(u, v+1)$. The third and final relation is that one is always less than or equal to the other, but equality is only achieved when both expressions are small: $0 \leq \max(u, v)$. The bonus relation is $u \bowtie v$, for expressions that

are uncomparable: sometimes greater, sometimes less than, and necessarily arbitrarily large in either direction.

4.4.1 Relations Without Impredicative Maximum

These three relations can be wrapped up into a single semigroup that describes how two level expressions are related. The difference between the last two relations, then, is how they combine: the former is infectious, in that it takes priority over the strict inequality, while the latter is subsumed by it.

```
data Rel
  = H_EQ -- equal (and arbitrarily large)
  | S_LT | S_GT -- strict inequality
  | H_LE | H_GE -- weak inequality, with arbitrarily large equality
  | L_LE | L_GE -- weak inequality, but only equal at small values
  | UNCOMP -- uncomparable
```

This has a commutative, idempotent semigroup structure. A selection of key cases are shown below:

```
instance semigroupRel :: Semigroup Rel where
  append UNCOMP _ = UNCOMP

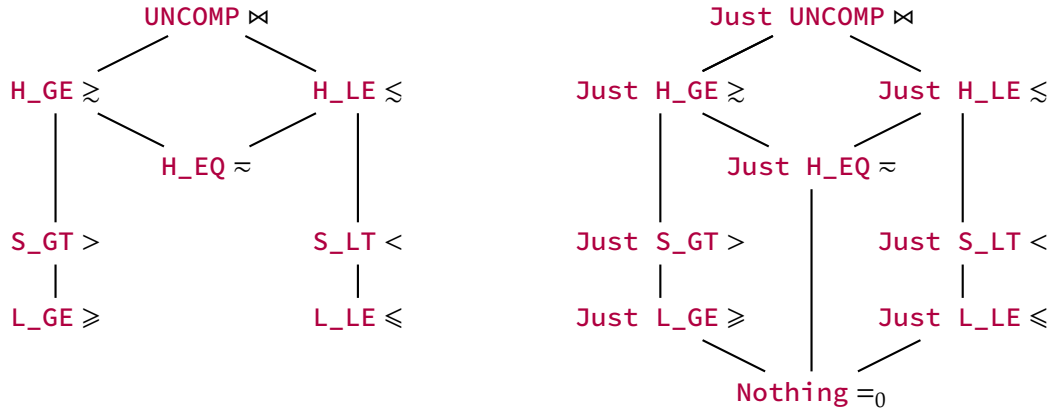
  append S_LT H_EQ = H_LE
  append H_LE H_EQ = H_LE
  append L_LE H_EQ = L_LE

  append H_LE L_LE = H_LE

  append S_LT H_LE = H_LE
  append S_LT L_LE = S_LT
```

To get a monoid out of this, we just adjoin an identity. The functor in PureScript that canonically does this is `Maybe`, which has two constructors: `Nothing :: forall a. Maybe a` and `Just :: forall a. a -> Maybe a`. Of course the monoid `Maybe Rel` is still commutative and idempotent.

In fact, a commutative, idempotent semigroup is a semilattice. Adjoining the identity for the semigroup operation is the same as adjoining a bottom element for the semilattice, producing a bounded semilattice. These are the Hasse diagrams for `Rel` and `Maybe Rel`.

Figure 4.4.1: Hasse diagrams for **Rel** and **Maybe Rel**

For the \mathcal{L} normal form **UnivMax**, the expressions are compared variablewise and these results are combined together with the semilattice operation. There is one annoying detail: this information cannot be combined with the information from the constant using the same semilattice operation (which should be **Nothing**, **Just S_LT**, or **Just S_GT** as appropriate). It is almost correct, except for one small detail: on cases where the constants are equal but the rest of the expression is strictly comparable, as when comparing $\max(x + 4, 5)$ with $\max(x, 5)$, the result would be a high inequality **H_LE** or **H_GE** instead of the expected low inequality **L_LE** or **L_GE**, since those two example expressions are only equal for $x = 0, 1$ and not arbitrarily high. In fact, for the rest of this logic, the difference is not essential, but it is nice to maintain it. Thus we add this small check as a special case, since it goes against the monotonicity of the semilattice.

Just L_LE

4.4.2 Relations With Impredicative Maximum

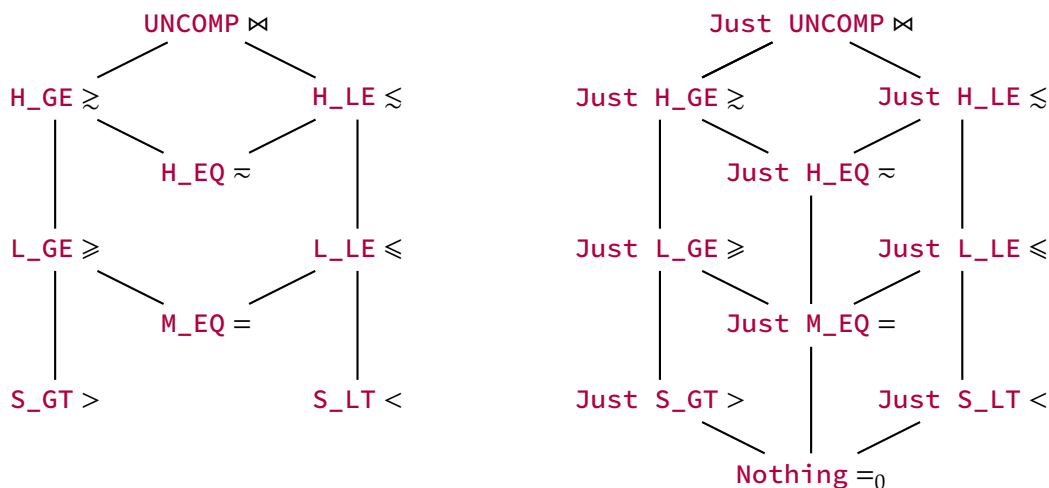
For the \mathcal{L}_{if} normal form **UnivIMax**, the expressions are compared for each relevant hypothetical and combined with **new** semilattice operation. A new operation is needed because if two impredicative expressions can be equal at one hypothetical, they of course can be equal overall, not matter if it is a low equality or a high equality. So in this new operation, **L_GE** dominates **S_GT** and **L_LE** dominates **S_LT** instead of the other way around. That is, the distinction between **L_GE** and **H_GE** is not so useful anymore (they are adjacent in the semilattice), but it might as

well be kept around since the information is available! In addition there is a “middle” notion of equality `M_EQ` for when constants are equal in the empty hypothetical (this is somewhat analogous to the above discussion of promoting from `S_LT` to `L_LE` and `S_GT` to `L_GE`).

This means that there are some annoying details, but the core of the algorithm is comparing the expressions hypothetical-by-hypothetical, and in fact it all fits into the semilattice structure this time. The first annoying detail is the aforementioned one about constants at the empty hypothetical being considered `Just M_EQ` instead of `Nothing`. The other is that a minor correction of the same type: if two expressions are being compared at hypothetical h and at $h \cup \{x\}$, then the comparison at the hypothetical h can discard the variable x , since it will contribute zero at the smaller hypothetical and will be instead picked up at the larger one. This corrects some comparisons from `H_GE` to `L_GE` and `H_LE` to `L_LE`, such as when comparing $\max(\text{if}(x + 2; x), x)$ and $\max(\text{if}(x + 3; x), x)$, which are only equal when $x = 0$, not at arbitrarily high values. Again, this is not a concern for correctness of the overall algorithm, where the difference between `H_GE` and `L_GE` is no longer relevant.

```
data IRel = IRel Rel | M_EQ
```

Figure 4.4.2: Hasse diagrams for `IRel` and `Maybe IRel`



```
compareUnivIMax :: UnivIMax → UnivIMax → Maybe IRel
compareUnivIMax uz1' uz2' =
  foldMap compRel hypotheses
```

```

where
  Pair uz1 uz2 = normalizeUnivIMax <$> Pair uz1' uz2'
  hypotheses =
    -- always compare at the empty hypothetical
    Set.singleton Set.empty
    -- and at the hypotheticals from each expression
    ◇ Map.keys (unwrap (unwrap uz1))
    ◇ Map.keys (unwrap (unwrap uz2))
    -- Don't include a variable in the comparison if we are going to
    -- explicitly include it in the next hypothesis
    -- (This will only correct some comparisons from H_LE to L_LE)
  discarding ks (UnivMax (SemigroupMap us) u) =
    UnivMax (SemigroupMap (Map.mapMaybeWithKey (discardKey ks) us)) u
  discardKey ks k =
    if not (Set.member k ks) && Set.member (Set.insert k ks) hypotheses
    then \_ → Nothing
    else Just
  compRel ks = liftAt ks $ compareUnivMax
    (discarding ks (foundAt ks uz1))
    (discarding ks (foundAt ks uz2))
  liftAt ks Nothing | Set.isEmpty ks = Just M_EQ
  liftAt _ v = IRel <$> v

```

4.4.3 Proofs

Here we present proofs that justify why these relations combine in the ways they do. Some parts of the proofs are trivial: for example, if $x < y$ and $a \leq b$, then $\max(x, a) \leq \max(y, b)$ by monotonicity. This example goes halfway towards showing that $S_LT \diamond H_LE = H_LE$ is a valid deduction: it covers the half that states “if x is always less than y across models, and a is always less than or equal to b , then $\max(x, a)$ is always less than or equal to $\max(y, b)$ ”. The other half of the propositions is more tricky: “if there are no models where x equals y , but there are models where a equals b arbitrarily large, then there are models where $\max(x, a)$ equals $\max(y, b)$ arbitrarily large”. The ability to manipulate models like this is not true without additional assumptions: for example, it fails when $x = u$ and $y = u + 1$ and $a = \max(u, v)$ and $b = \max(u, v + 2)$, since the composite is $\max(u, v)$ versus $\max(u + 1, v + 2)$ which are strictly incomparable!

What went wrong? The fact that the expressions had overlapping variables causes the comparison to behave unpredictably. As a result, simple assumption that x, y have disjoint variables from a, b makes the lattice structure work as it should.

For example, this immediately shows that **UNCOMP** is the absorbing element of the semilattice: of $x \bowtie y$ and x, y have disjoint variables from a, b , then since x can be made arbitrarily large with y under it, and vice-versa with y arbitrarily large over x , then $\max(x, a) \bowtie \max(y, b)$ clearly holds since x can dominate a and y can dominate b .

Let x, y, a, b be universe expressions. Clearly if $x \leq y$ and $a \leq b$ then $\max(x, a) \leq \max(y, b)$, since $\max(\max(x, a), \max(y, b)) = \max(\max(x, y), \max(a, b)) = \max(y, b)$. Similarly if $x < y$ and $a < b$ then $\max(x, a) < \max(y, b)$. This justifies **S_LT** \diamond **S_LT** = **S_LT**.

Compare $\max(v + k, e_1)$ and $\max(v + k, e_2)$, where v does not occur in e_1 or e_2 : add **H_EQ** to comparing e_1 and e_2 . Compare $\max(v + k, e_1)$ and $\max(v + k + m + 1, e_2)$ where v does not occur in e_1 or e_2 : add **S_LT** to comparing e_1 and e_2 .

The idea is that by comparing variable against variable, we don't run into conflicts in the models we want to produce. If we have $u = u$, then the models we want to produce to obtain arbitrarily large equalities hold after considering $\max(u, e_1) = \max(u, e_2)$. This is because we can always consider u to be larger than $\max(e_1, e_2)$ – as long as e_1 and e_2 do not mention u . More exactly: to construct a model where $\max(u, e_1) = \max(u, e_2)$ we set all the other variables to 0 and then take $u = \max(e_1, e_2)$, where e_1 and e_2 can now be evaluated in the model with all the other variables being 0.

Note

Normalization of **UnivMax** expressions helps, but is an orthogonal concern, merely enforcing that $u \geq 0$ for all variables/expressions.

4.5 Constraining Universe Levels

As type inference progresses, more constraints are added to universe levels that need to be checked for consistency. The actual state that is kept is a map from level expressions to level

expressions, where each key-value pair represents the constraint that the key is *greater than* the value.

```
newtype GEConstraints = GEConstraints (Map UnivIMax UnivIMax)
```

The reason for this choice is that the conjunction of $u \geq v$ with $u \geq w$ is $u \geq \max(v, w)$. So having a single level expression as a lower bound for each key suffices.

The primary aspect of solving is just saturating all of the known relations between expressions, starting with reflexivity. For example, if $u \geq v$ is a constraint and $v \geq w$ is also a constraint, then $u \geq w$ is a constraint, by transitivity. In practice, this means looking at all pairs of key-value pairs and adding $k_i \geq v_j$ when $v_i \geq k_j$. However, this does not capture the distributivity of successor. In general we want to adjust values by a constant that expresses when k_j becomes less than v_i : for the largest $c \in \mathbb{Z}$ such that $v_i \geq k_j + c$, we can add $k_i \geq v_j + c$ as a constraint, justified by the chain $k_i \geq v_i \geq k_j + c \geq v_j + c$.

If $k_i < v_i$ is always true (across all models) for some pair, then an error must be thrown since it is no longer consistent (this is important for termination of the saturation algorithm too). In fact, more generally one wants to reduce the key if there are parts of v_i that are strictly greater than the corresponding parts of k_i . For example, if there is the constraint $\max(u, v) \geq \max(w, v + 1)$, that is $\max(u, v) \geq w$ and $\max(u, v) \geq v + 1$. But the latter can only be satisfied by $u \geq v + 1$, since $v \not\geq v + 1$. So $\max(u, v) = u$ in fact, thus the constraint can be reduced to the entry $u \geq \max(w, v + 1)$.

Finally there are a couple wrinkles to be figured out with `imax`. For example, if $\text{imax}(u; v) > 0$ then $\text{imax}(u; v) \geq \max(u, v)$, but recall that it is encoded in \mathcal{L}_{if} as $\max(\text{if}(u; v), v)$. As a more complicated situation with two variables, $\max(\text{imax}(l; a), \text{imax}(r; b)) > 0$ implies that $\max(a, b) > 0$ so $\max(\text{imax}(l; a), \text{imax}(r; b)) \geq \min(l, r)$, it would be weird to have to add a minimum operator, and again, it is not so obvious to see how this applies in the \mathcal{L}_{if} normal form in full generality and whether it does influence satisfiability.

However, by brute-force case analysis, it is possible to decide $\mathcal{L}_{\text{imax}}$ in at worst exponential time over the plain \mathcal{L} algorithm. So that is what we do. This involves looking for constraints of $v + k \geq k + 1$ (forcing v to be positive) or $k \geq v + k$ (forcing v to be zero) in the context.

4.5.1 Consistency

The goal is for the algorithm to detect the collective consistency of the constraints. Using the ingredients discussed above (normal forms, seeing how levels compare across all models, and applying all known axioms to derive new constraints), this should be the case, but we only sketch the direction of a proof here, especially since the challenge is intertwined with verifying that the algorithm terminates, given that there is no fixed base case, simply looping until it generates no new constraints.

The first direction is the most reasonable: since only known axioms are applied at each stage of the algorithm, it should be the case that no consistent set of constraints is ruled inconsistent. The other direction is a bit more tricky: did we cover all possible combinations of axioms and constraints, in order to detect the inconsistent sets of constraints? It is a bit tricky to identify where each axiom is used, since some are embedded in the very structure of the normal forms (like how successor distributes across maximum), or in the normalization functions (like how zero is the least element), and the rest do appear at various points during the constraint solving process. But there is reason to think that it is complete: all applicable axioms are used.

There is a more direct way to show this too: we should be able to exhibit at least one model that satisfies the constraints, if it is consistent. We can do this relatively easily from a saturated constraint context, which essentially does the work of finding lower bounds for each variable. However, there are two challenges: the saturated constraint context will give lower bounds not for variables, but for expressions, and generally speaking, simply setting all variables to their lower bounds all at once will result in a model that does not satisfy the constraints anymore.

The algorithm then is to look at the saturated constraint context and pick the smallest possible move towards a model that still makes progress. Then that change is added to the constraint context, which propagates the new information to create a new saturated context and step towards creating a model with all variables assigned.

The smallest possible move consists of finding a constraint that has some variables on the left (preferring ones with the fewest variables) and some expression on the right, and then setting

one of the variables to the smallest value that makes the left expression larger than the constant on the right. For example, if $\max(u + 2, v, 5) \geq \max(w, v + 1, 8)$ is a constraint in the saturated context, then $u = 6$ may be chosen (reducing it to $\max(v, 8) \geq \max(w, v + 1, 8)$) or $v = 8$ may be chosen (reducing it to $\max(u + 2, 8) \geq \max(w, 8)$). If no moves are found, the remaining variables are set to 0, since they only appear with upper bounds.

The key claim is that if a claimed lower bound $v = k$ is actually inconsistent with the other constraints, the saturated constraint context should have “detected” that and included a constraint equivalent to $v \geq k + 1$. If this is true, then the new constraint context will retain its consistency after every choice, and eventually terminate because the algorithm should make progress with each choice, even though more constraints may be added as a result each time during saturation.

At least empirically these claims all seem true: in the test cases run, they all check out in that the algorithm terminates, correctly detects inconsistencies, accepts consistent contexts, and produces concrete models for those consistent contexts.

5

Row Types

Typechecking records and unions is one of the most complex parts of the Dhall standard already. Dhall mitigates some of this complexity by requiring that all records and unions have statically known shapes. Nevertheless, hiding behind the scenes are the outlines of a concept called row types that capture the shared shapes of records and unions. Row types have been explored before, such as by Didier Rémy who explores a similar system of constraints [6]. Here we sketch what it would look like for the operators Dhall supports in particular.

Row types consist of an unordered set of labels associated with types. “Open” rows have some known labels at the “head” along with a “tail” row of unknown fields. There may also be some labels that are known to *be absent* from the tail of the row. “Closed” rows are completely known, with an empty tail.

For simple constraints on rows, open rows over row variables are sufficient to express them. For example, getting a single field from an open record is trivial with open rows.

However, merging two open rows cannot be solved in the language of open rows. To allow more complex programs to be polymorphic over these shapes, we need to introduce a system of constraints similar to the case of universe polymorphism.

In order for information to flow backwards, instead of introducing functions on row types, we introduce constraint relations that track inputs and outputs equally. Of course, the primary mode of learning information is learning about presence and absence of labels.

Since row types contain other types, this requires some machinery on types (namely unification and apartness); see the next chapter for that. In particular, apartness will direct the path of solving, while unification constraints are part of the resultant data generated by solving.

Unlike universe constraints, which do not necessarily “solve” for universe level variables, row constraints are best expressed by solving for variables and filling in information partially. One challenge is the fact that information comes in in a quasi-ordered fashion, while rows have to conceptually be completely unordered.

Todo

How to disallow duplicate labels? Just show it never introduces duplicate labels?

Todo

Generate the “backwards” rules from the forwards?

We write $\{ \dots r \}$ for the record type with fields specified by the record row $(\dots r)$. A record row $(l : t, \dots r)$ contains type t at a known label l in the head, and r as the tail.

We write $\langle \dots rm \rangle$ for the union type with fields specified by the union row $\langle \dots rm \rangle$. A union row $\langle l : ? mt, \dots r \rangle$ contains an optional type mt at a known label l in the head, and r as the tail. That is, $\langle l, \dots r \rangle$ denotes the union row with a label l but no type associated to that label, and $\langle l : t, \dots r \rangle$ associates to it a type as usual.

We write $r \setminus l$ to denote the row r with label l removed.

First we describe the constraint $r_1 \bowtie r_2 = r_3$ for typing the recursive record merge operator using a fresh row metavariable r_3 :

$$\frac{a : \{ \dots r_1 \} \quad b : \{ \dots r_2 \} \quad r_1 \bowtie r_2 = r_3}{a \wedge b : \{ \dots r_3 \}}$$

Learning that the label is *not* in the output tells us that the label is in neither of the inputs, and vice-versa:

$$\frac{r_1 = (r_1 \setminus l) \quad r_2 = (r_2 \setminus l)}{r_1 \wp r_2 = (r_3 \setminus l)} \quad \frac{r_3 = (r_3 \setminus l)}{(r_1 \setminus l) \wp (r_2 \setminus l) = r_3}$$

Learning the label is in the left or right side tells us that the label is in the output, but the constraint is stuck because the types cannot be related yet:

$$\frac{r_3 = (l : t_3, \dots(r_3 \setminus l))}{(l : t_1, \dots r_1) \wp r_2 = r_3} \quad \frac{r_3 = (l : t_3, \dots(r_3 \setminus l))}{r_1 \wp (l : t_2, \dots r_2) = r_3}$$

Learning the label is in the left or right side and absent from the other does make progress:

$$\frac{r_3 = (l : t_3, \dots(r_3 \setminus l)) \quad t_1 = t_3 \quad r_1 \wp r_2 = (r_3 \setminus l)}{(l : t_1, \dots r_1) \wp (r_2 \setminus l) = r_3} \quad \frac{r_3 = (l : t_3, \dots(r_3 \setminus l)) \quad t_2 = t_3 \quad r_1 \wp r_2 = (r_3 \setminus l)}{(r_1 \setminus l) \wp (l : t_2, \dots r_2) = r_3}$$

Finally, learning the label is in both sides triggers the recursive case:

$$\frac{r_3 = (l : t_3, \dots(r_3 \setminus l)) \quad t_1 = \{ \dots r'_1 \} \quad t_2 = \{ \dots r'_2 \} \quad t_3 = \{ \dots r'_3 \} \quad r'_1 \wp r'_2 = r'_3 \quad r_1 \wp r_2 = (r_3 \setminus l)}{(l : t_1, \dots r_1) \wp (l : t_2, \dots r_2) = r_3}$$

Next we describe the constraint $r_1 \wp r_2 = r_3$ for typing the right-biased record merge operator:

$$\frac{a : \{ \dots r_1 \} \quad b : \{ \dots r_2 \} \quad r_1 \wp r_2 = r_3}{a \wp b : \{ \dots r_3 \}}$$

Again, the label is absent from the output if and only if it is absent from both inputs:

$$\frac{r_1 = (r_1 \setminus l) \quad r_2 = (r_2 \setminus l)}{r_1 \wp r_2 = (r_3 \setminus l)} \quad \frac{r_3 = (r_3 \setminus l)}{(r_1 \setminus l) \wp (r_2 \setminus l) = r_3}$$

On the left side it gets stuck until it is known to be absent from the right:

$$\frac{r_3 = (l : t_3, \dots(r_3 \setminus l))}{(l : t_1, \dots r_1) \wp r_2 = r_3} \quad \frac{r_3 = (l : t_3, \dots(r_3 \setminus l)) \quad t_1 = t_3 \quad r_1 \wp r_2 = (r_3 \setminus l)}{(l : t_1, \dots r_1) \wp (r_2 \setminus l) = r_3}$$

But it always makes progress on the right side:

$$\frac{r_3 = (l : t_3, \dots(r_3 \setminus l)) \quad t_2 = t_3 \quad r_1 \wp r_2 = (r_3 \setminus l)}{r_1 \wp (l : t_2, \dots r_2) = r_3}$$

Finally we describe the constraint $r_1 \$ rm_2 \rightarrow t_3$ for typing the record–union merge expression:

$$\frac{a : \{ \dots r_1 \} \quad b : \langle \dots rm_2 \rangle \quad r_1 \$ rm_2 \rightarrow t_3}{\text{merge } a \ b : t_3}$$

Both rows must have the same labels, but it gets partially stuck if it is not know whether the union type has data at a label:

$$\frac{rm_2 = \langle l: ? mt_2, \dots (rm_2 \setminus l) \rangle}{(l: t_1, \dots r_1) \$ rm_2 \rightarrow t_3} \quad \frac{r_1 = (l: t_1, \dots (r_1 \setminus l))}{r_1 \$ \langle l: ? mt_2, \dots rm_2 \rangle \rightarrow t_3}$$

Knowing that makes progress:

$$\frac{t_1 = t_3 \quad r_1 \$ rm_2 \rightarrow t_3}{(l: t_1, \dots r_1) \$ \langle l, \dots rm_2 \rangle \rightarrow t_3} \quad \frac{t_1 = t_2 \rightarrow t_3 \quad r_1 \$ rm_2 \rightarrow t_3}{(l: t_1, \dots r_1) \$ \langle l: t_2, \dots rm_2 \rangle \rightarrow t_3}$$

Since the type problem is undecidable (and computationally expensive), the row type problem is no better. However, it might still be possible to make an algorithm for the row type problem that is complete with respect to an oracle for the type problem or to restrict types to a decidable subset (say, one base type like **Text**, function types, and record types).

6

Unification

Unification is a relation on terms that indicates what needs to be the case for two types to be equal, and what their unified result will be. It expresses a demand by the typechecker: the term will not be well-typed unless it can find evidence that the types to be unified are in fact equal in context. Thus it fits well into the constraint system of this paper, as each unification constraint can be kept in the constraints produced by typechecking, and then later solved (partially or fully). Because unification can affect things like evaluation, unlike universe and row constraints, all unification constraints must be solved fully to produce a program: there can be no metavariables left.

A related problem is to quickly determine some cases in which two values can never be equal. This is used above to make progress on row constraints, and in fact it may use unification constraint context to make these determinations because it is operating on the constraint context instead of creating constraints.

Unification is a powerful tool for implementing typechecking, but also involves difficult trade-offs because it ultimately faces an impossible problem: determining whether two values might be equal is undecidable, and so is the same problem for types, since types can mention values in dependently-typed systems. So the actual evidence that unification may search for depends on the theory and implementation.

There are many papers on the topic. One approach is “Type checking through unification”, which uses unification in key ways to do much of the work of type checking via unification, and navigates the tight weave between typechecking, unification, and evaluation [5].

Subsumption adds a twist on unification by allowing for an order, as if one type is larger than another. The philosophy of subsumption is that one type is subsumed by a second type if all values of the first type may be used where values of the second type are expected.

In the system of this paper then, the only substantial difference between unification and subsumption is for the universe judgment, which introduces the inequality $u \leq v$ for `Universe` $u \trianglelefteq$ `Universe` v . This asymmetry propagates through type constructors, including contravariantly in function arguments while covariantly in function outputs and also record and union types.

The key detail here is that the structure of the terms is still required to be the same for subsumption, it is just the values of universe levels that may differ (and recursively through rows too). This means that the unification algorithms still apply.

Unification and subsumption are reflexive and transitive. Unification and apartness are symmetric.

Unification and subsumption are the same judgment, just parameterized over a relationship ($=$) for unification, (\leq) or (\geq) for subsumption) which is only used as the constraint for comparing universe levels right now. This means that it does not handle row expansion or contraction – but hopefully row polymorphism is sufficient for that.

Maybe apartness should be called disunifiability, since it does not serve the same purpose as it does in PureScript/Haskell. Namely, it will consider distinct (λ -)bound variables apart, because they could never be unified, though they could be instantiated to the same.

We use metavariables for unification, to represent types that used to be required but can now be omitted. Technically they can also represent non-types, but unification is much less useful for non-types because computations are not necessarily injective or generative, unlike type constructors.

Metavariables can of course be unified with anything (as long as scope matches up?), and so will be apart from nothing.

7

Properties

To deliver on the promise that universes really are the type of types, it must be the case that each typing judgment produces a type whose type is a universe, and this is easily verified by inspection of the typing rules:

$$\frac{\Gamma \vdash e : T \Leftarrow C_1}{\Gamma \vdash T : \mathbf{Universe} \ u \Leftarrow C_2 \quad C_2 \subseteq C_1}$$

This is true in the current Dhall standard except for the lack of universes above `Sort` (that is `Universe 2`).

All subterms typecheck, and their constraints appear in the result. This is true from the structure of the typechecking judgments, which typecheck all immediate subterms and include those constraints in the result. Thus transitive subterms are also included.

All type-level functions automatically respect subsumption, because all universe-level expressions are monotonic, and term structure is otherwise preserved.

Let-substitution typechecks. This requires the substitution to respect the variable instantiations chosen during typechecking, but then with this set-up it is mostly trivial. I believe the resultant type and constraints will be verbatim, just with the same substitution applied.

$$\frac{\Gamma, (x := e_2 : T_2 \Leftarrow C_2), \Gamma' \vdash e_1 : T_1 \Leftarrow C_1}{\Gamma, \Gamma'[x := e_2] \vdash e_1[x := e_2] : T_1[x := e_2] \Leftarrow C_1[x := e_2]}$$

Context subsumption. A term typechecks under a more specific context too, resulting in a similar but potentially more specific type/constraints. This is true mainly for structural reasons like the above property: since the constraints propagate through type inference in a way that respects subsumption, it is enough that typechecking each variable produces a more specific type – but this is exactly what the assumption of context subsumption gives us, possibly under some extra constraints C_3 that will help ensure that the resultant constraints C_2 are more specific than the original ones C_1 .

$$\frac{\Gamma_1 \vdash e : T_1 \Leftarrow C_1 \quad \Gamma_2 \sqsubseteq \Gamma_1 \Leftarrow C_3}{\Gamma_2 \vdash e : T_2 \Leftarrow C_2 \quad (T_2 \Leftarrow C_2) \sqsubseteq (T_1 \Leftarrow C_1) \Leftarrow C_3}$$

Type preservation. For a given context Γ , if e_1 typechecks with type T_1 under constraints C_1 , and it evaluates to e_2 , then e_2 also typechecks, possibly with a more general type/constraints. More specifically, e_2 will have type T_2 which is subsumed by T_1 (that is, e_2 can be used everywhere an expression of type T_1 is expected) under the constraints C_3 . The constraints $C_2 \cup C_3$ must be implied by the first constraints C_1 . (Note: this is semantic: they may not literally be a subset of the atomic constraints from C_1 .)

$$\frac{\Gamma \vdash e_1 : T_1 \Leftarrow C_1 \quad \Gamma \vdash e_1 \mapsto e_2}{\Gamma \vdash e_2 : T_2 \Leftarrow C_2 \quad T_2 \sqsubseteq T_1 \Leftarrow C_3 \quad C_2 \cup C_3 \subseteq C_1}$$

$$\frac{\Gamma_2 \sqsubseteq \Gamma_1 \Leftarrow C_3 \quad \Gamma_1 \vdash e_1 : T_1 \Leftarrow C_1}{\Gamma_2 \vdash e_2 : T_2 \Leftarrow C_4}$$

Everything that typed before still types, since the rules are strictly more general.

Constraints don't affect evaluation.

8

Future Work

Some obvious technical challenges remain with universes. In order to enable better handling of the contravariance of function inputs, it would be great to extend the arithmetic with a minimum operation in addition to the maximum, making it a lattice (not merely a semilattice), but that represents a significant extension in scope. It would also be beneficial to come up with a clear way to present universe errors to users, since they can be arcane and hard to debug, especially if the user is given some random-seeming universe level constraint(s) no specific indication of why they occurred, how they were derived, and where the conflicts are in the source code.

One key optimization is minimizing constraint contexts. When a universe variable is mentioned in a term but not the type of the term, the variable can still appear in the constraint context, even though it will have no interaction since it does not appear in the type (the public interface)! In many cases these variables can be eliminated, but setting them to appropriate values (usually lower bounds). The implementation of universe polymorphism in Coq is much more aggressive about this kind of minification, but it has led to some bugs and unexpected behavior by making assumptions that reduce the generality of the polymorphism, whereas an ideal algorithm would preserve full generality [7].

To give an example of how common it is to have constraints that can be minimized, polymorphic functions like the built-in `Natural/fold` take a type of any universe level in as an argu-

ment, and the subsumption rule invoked during function application says this universe level instantiated by **Natural**/fold just needs to be large enough to contain the argument, potentially being larger as well. But once it is applied, that universe variable does not appear in the output anymore, and in fact it is always enough to assume that it is in fact the same level as the argument, thus eliminating it from the constraints.

Another direction is the obvious one of higher-order polymorphism, where arguments that are functions can in fact be invoked as polymorphic functions, and their constraints tracked in the body of the function and then checked when an argument is applied. The main challenge here is consistency: whereas let-polymorphism was clearly consistent since it is just syntactic sugar for substitution with a sprinkle of polymorphism, it's not clear what universe to assign to polymorphism functions such that it remains consistent. One option is to then index universes by infinite ordinals, such as $0 < 1 < \dots < \omega$, but that seems like a lot more work to stratify universes across that new layer.

At a more practical level, higher-order polymorphism will require understanding functions over universe levels and row types. For example, if $F : \text{Universe } ? \rightarrow \text{Universe } ?$ is a input to a function, it is reasonable to expect it to still be polymorphic: that is, instead of having type $\text{Universe } u \rightarrow \text{Universe } v$ for some fixed u and v , it rather would have type $\text{Universe } u \rightarrow \text{Universe } v(u)$, implicitly quantified over the input level u , with the output level $v(u)$ depending on the input level. Particular values for F will have different behaviors:

1. Constant functors like $\text{Const } l = \text{Natural}$ will have $v(u)$ also constant (0 in this case), since the output does not depend on the input.
2. The identity functor and builtin functor **List** will both have $v(u) = u$.
3. Some functors may increase the level, e.g. $v(u) = \max(u, 1)$.

Nevertheless, all universe level functions will have certain properties: they will be monotonic, in fact they will distribute over maximum as all operators in the algebra do: $v(\max(u, w)) = \max(v(u), v(w))$.

Similarly, row types will need to be extended handle row functions. Again, there are only a few building blocks that dictate how the output will relate to the input, so the structure of row functions is pretty restricted.

There is clearly much more work to do with unification/subsumption and integrating them into the framework of the other constraints, though it should be possible. It is obvious that they produce constraints of the other kind, but it is not clear how this process interact with polymorphic variables. That is, if the type of a let-binding or higher-order argument is (initially) unknown, how can the system know what metavariables to generalize over, if it is later discovered to be a function?

Finally, it would be amazing if this work could be extended to a principal typing algorithm for Dhall. The idea of principal typing is that each term can produce the constraints that its context must satisfy to make it well-typed, and it feels like if the unification algorithm is good enough, it should work for Dhall. There is some work on doing this for other type theories, though mostly in the absence of higher-order polymorphism [4][8]. Hopefully this technique of pushing the polymorphism out to the leaves (universe levels and row types exclusively) makes it more tractable.

Appendix A

Judgments

A.1 Syntax

Definition of the reduced Dhall syntax. The e_i denote expressions, n denotes a natural number literal, and x denote names (both variable names and labels for records/unions). Note that the `merge` keyword cannot occur on its own; instead, it must always occur applied to two arguments, with an optional type annotation (this is necessary when the union and record are empty).

$$\begin{aligned}
e_i, T_i ::= & \text{Universe } u \mid \text{Natural} \mid \text{List} \\
& \mid n \mid [] : e \mid [e_1, e_2, \dots, e_n] \\
& \mid \backslash(x : e_1) \rightarrow e_2 \mid \text{forall}(x : e_1) \rightarrow e_2 \\
& \mid \text{let } x = e_1 \text{ in } e_2 \\
& \mid e_1 e_2 \mid e_1 : e_2 \\
& \mid \{ \} \mid \{ x_1 : e_1, x_2 : e_2, \dots, x_n : e_n \} \\
& \mid \{ \equiv \} \mid \{ x_1 = e_1, x_2 = e_2, \dots, x_n = e_n \} \\
& \mid \langle \rangle \mid \langle x_1 : e_1, x_2 : e_2, \dots, x_n : e_n \rangle \\
& \mid \langle x = e \rangle \mid \langle x = e, x_1 : e_1, x_2 : e_2, \dots, x_n : e_n \rangle \\
& \mid x \mid e.x \mid \text{merge } e_1 e_2 \mid \text{merge } e_1 e_2 : e_3 \\
& \mid e_1 \wedge e_2 \mid e_1 \bowtie e_2 \mid e_1 \# e_2 \mid e_1 \# \# e_2 \\
& \mid \text{Natural} / \text{fold} \mid \text{List} / \text{fold} \\
u_i ::= & n \mid x \mid u_1 + n \mid \max(u_1, u_2) \mid \text{imax}(u_1; u_2) \\
n ::= & 0 \mid 1 \mid 2 \mid \dots
\end{aligned}$$

Contexts

$$\Gamma, \Delta ::= \cdot \mid \Gamma, (x : T_1) \mid \Gamma, (x := e_1 : T_2 \Leftarrow C_1)$$

Constraints C_i have the form of a set of atomic constraints \hat{C}_i , which can be universe constraints or row constraints or unification/subsumption constraints:

$$C_i ::= \emptyset \mid \{ \hat{C} \} \mid C_1 \cup C_2$$

$$\hat{C} ::= u_1 = u_2 \mid r_1 \bowtie r_2 = r_3 \mid r_1 \# r_2 = r_3 \mid r_1 \$ m r_2 \rightarrow e_3 \mid e_1 \trianglelefteq e_2$$

A.2 Substitution

The only nontrivial rules for substitution are for applying a substitution to a variable: if the variable matches, substitution needs to return the substitution value instead. This simplicity is because we assume all variables are distinct, otherwise variable bindings would need to be handled more carefully. This means that aside from the base case, all expressions are substituted recursively in the obvious manner.

However, just substituting the literal value upon seeing a variable does not do what we want, since this will preserve metavariables that should be instantiated to fresh variables for polymorphism. What needs to happen is that during typechecking, the decision of how to instantiate the metavariables needs to be encoded in the syntax tree at each variable occurrence, and this substitution performed when a value is chosen for the variable. Of course, this seems redundant since polymorphism is only for let-bound variables at the moment, but supporting higher-order

polymorphism would require it for lambda- and forall-bound variables.

$$x[x := v] = v$$

$$x[y := v] = x$$

$$n[y := v] = n$$

$$([], e)[y := v] = [] : e[y := v]$$

$$[e_1, e_2, \dots, e_n][y := v] = [e_1[y := v], e_2[y := v], \dots, e_n[y := v]]$$

$$(\lambda(x : e_1) \rightarrow e_2)[y := v] = \lambda(x : e_1[y := v]) \rightarrow e_2[y := v]$$

$$(\text{forall}(x : e_1) \rightarrow e_2)[y := v] = \text{forall}(x : e_1[y := v]) \rightarrow e_2[y := v]$$

$$(\text{let } x = e_1 \text{ in } e_2)[y := v] = \text{let } x = e_1[y := v] \text{ in } e_2[y := v]$$

$$(e_1 : e_2)[y := v] = e_1[y := v] : e_2[y := v]$$

$$\{\}[y := v] = \{\}$$

$$\{x_1 : e_1, \dots, x_n : e_n\}[y := v] = \{x_1 : e_1[y := v], \dots, x_n : e_n[y := v]\}$$

$$\{\} \Rightarrow [y := v] = \{\}$$

$$\{x_1 = e_1, \dots, x_n = e_n\}[y := v] = \{x_1 = e_1[y := v], \dots, x_n = e_n[y := v]\}$$

$$\langle \rangle [y := v] = \langle \rangle$$

$$\langle x_1 : e_1, \dots, x_n : e_n \rangle [y := v] = \langle x_1 : e_1[y := v], \dots, x_n : e_n[y := v] \rangle$$

$$\langle x = e \rangle [y := v] = \langle x = e \rangle$$

$$\langle x = e, x_1 : e_1, \dots, x_n : e_n \rangle [y := v] = \langle x = e[y := v], x_1 : e_1[y := v], \dots, x_n : e_n[y := v] \rangle$$

$$e.x[y := v] = e[y := v].x$$

$$(\text{merge } e_1 \ e_2)[y := v] = \text{merge } e_1[y := v] \ e_2[y := v]$$

$$(\text{merge } e_1 \ e_2 : e_3)[y := v] = \text{merge } e_1[y := v] \ e_2[y := v] : e_3[y := v]$$

$$\text{Natural}/\text{fold}[y := v] = \text{Natural}/\text{fold}$$

$$\text{List}/\text{fold}[y := v] = \text{List}/\text{fold}$$

A.3 Typing

A.3.1 Builtins

$$\begin{array}{c}
\frac{}{\Gamma \vdash n : \mathbf{Natural}} \quad \frac{}{\Gamma \vdash \mathbf{Natural} : \mathbf{Universe} 0} \\
\\
\frac{u := \mathit{fresh}}{\Gamma \vdash \mathbf{Natural}/\mathit{fold} : \mathbf{Natural} \rightarrow \forall (r : \mathbf{Universe} u) \rightarrow (r \rightarrow r) \rightarrow r \rightarrow r} \\
\\
\frac{T := \mathit{fresh} \quad \Gamma \vdash e_1 : T_1, \dots, e_n : T_n \Leftarrow C_1 \quad T_1 \leq T, \dots, T_n \leq T \Leftarrow C_2}{\Gamma \vdash [e_1, \dots, e_n] : \mathbf{List} T \Leftarrow C_1 \cup C_2} \quad \frac{\Gamma \vdash e \rightarrow \mathbf{List} e_1}{\Gamma \vdash ([] : e) : \mathbf{List} e_1} \\
\\
\frac{u := \mathit{fresh}}{\Gamma \vdash \mathbf{List} : \mathbf{Universe} u \rightarrow \mathbf{Universe} u} \\
\\
\frac{u := \mathit{fresh} \quad v := \mathit{fresh}}{\Gamma \vdash \mathbf{List}/\mathit{fold} : \forall (a : \mathbf{Universe} v) \rightarrow \mathbf{List} a \rightarrow \forall (r : \mathbf{Universe} u) \rightarrow (a \rightarrow r \rightarrow r) \rightarrow r \rightarrow r}
\end{array}$$

A.3.2 Functions and Variables

Concrete and abstract variables:

$$\frac{vs := \mathit{metavariables}(e : T) \setminus \mathit{bound}(\Gamma) \quad us := \mathit{fresh}(vs)}{\Gamma, (x := e : T \Leftarrow C), \Delta \vdash x : T[vs := us] \Leftarrow C[vs := us]}$$

$$\frac{}{\Gamma, (x : T), \Delta \vdash x : T \Leftarrow \emptyset}$$

Let bindings:

$$\frac{\Gamma \vdash e_1 : T_1 \Leftarrow C_1 \quad \Gamma, (x := e_1 : T_1 \Leftarrow C_1) \vdash e_2 : T_2 \Leftarrow C_2}{\Gamma \vdash \mathbf{let} x = e_1 \mathbf{in} e_2 : T_2[x := e_1] \Leftarrow C_1 \cup C_2}$$

Lambdas and forall:

$$\frac{\Gamma \vdash T_1 : \mathbf{Universe} u_1 \Leftarrow C_1 \quad \Gamma, (x : T_1) \vdash e_1 : T_2 \Leftarrow C_2}{\Gamma \vdash (\lambda(x : T_1) \rightarrow e_1) : \forall (x : T_1) \rightarrow T_2 \Leftarrow C_1 \cup C_2}$$

$$\frac{\Gamma \vdash T_1 : \mathbf{Universe} u_1 \Leftarrow C_1 \quad \Gamma, (x : T_1) \vdash T_2 : \mathbf{Universe} u_2 \Leftarrow C_2}{\Gamma \vdash (\forall (x : T_1) \rightarrow T_2) : \mathbf{Universe} \mathit{imax}(u_1; u_2) \Leftarrow C_1 \cup C_2}$$

Function application:

$$\frac{\Gamma \vdash e_1 : \forall (x : T_1) \rightarrow T_2 \Leftarrow C_1 \quad \Gamma \vdash e_2 : T_3 \Leftarrow C_2 \quad \Gamma \vdash T_2 \leq T_3 \Leftarrow C_3}{\Gamma \vdash e_1 e_2 : T_2[x := e_2] \Leftarrow C_1 \cup C_2 \cup C_3}$$

A.3.3 Rows

$$\begin{array}{c}
\frac{}{\Gamma \vdash \{\} : \{\}} \quad \frac{}{\Gamma \vdash \{\} : \mathbf{Universe} \ 0} \quad \frac{}{\Gamma \vdash \langle \rangle : \mathbf{Universe} \ 0} \quad \frac{\Gamma \vdash e : \{l : T, \dots r\}}{\Gamma \vdash e.l : T} \\
\frac{\Gamma \vdash e_1 : T_1 \ \dots \ \Gamma \vdash e_n : T_n}{\Gamma \vdash \{l_1 = e_1, \dots, l_n = e_n\} : \{l_1 : T_1, \dots, l_n : T_n\}} \quad \frac{\Gamma \vdash T_1 : \mathbf{Universe} \ u_1 \ \dots \ \Gamma \vdash T_n : \mathbf{Universe} \ u_n}{\Gamma \vdash \{l_1 : T_1, \dots, l_n : T_n\} : \mathbf{Universe} \ \max(u_1, \dots, u_n)} \\
\frac{\Gamma \vdash e : T \ \Gamma \vdash T_1 : \mathbf{Universe} \ u_1 \ \dots \ \Gamma \vdash T_n : \mathbf{Universe} \ u_n}{\Gamma \vdash \langle l = e, l_1 : T_1, \dots, l_n : T_n \rangle : \langle l : T, l_1 : T_1, \dots, l_n : T_n \rangle} \\
\frac{\Gamma \vdash e_1 : \{\dots r_1\} \ \Gamma \vdash e_2 : \{\dots r_2\}}{\Gamma \vdash e_1 \wedge e_2 : \{\dots r_1 \ \& \ r_2\}} \\
\frac{\Gamma \vdash e_1 : \mathbf{Universe} \ u_1 \ e_1 \equiv \{\dots r_1\} \ \Gamma \vdash e_2 : \mathbf{Universe} \ u_2 \ e_2 \equiv \{\dots r_2\}}{\Gamma \vdash e_1 \ \& \ e_2 : \mathbf{Universe} \ \max(u_1, u_2)} \\
\frac{\Gamma \vdash e_1 : \{\dots r_1\} \ \Gamma \vdash e_2 : \{\dots r_2\}}{\Gamma \vdash e_1 \ // \ e_2 : \{\dots r_1 \ // \ r_2\}} \\
\frac{\Gamma \vdash e_1 : \mathbf{Universe} \ u_1 \ e_1 \equiv \{\dots r_1\} \ \Gamma \vdash e_2 : \mathbf{Universe} \ u_2 \ e_2 \equiv \{\dots r_2\}}{\Gamma \vdash e_1 \ // \ e_2 : \mathbf{Universe} \ \max(u_1, u_2)} \\
\frac{a : \{\dots r_1\} \ b : \langle \dots rm_2 \rangle \ r_1 \ \$ \ rm_2 \rightarrow t_3}{\mathbf{merge} \ a \ b : t_3}
\end{array}$$

A.4 Unification/Subsumption

We extend subsumption to cover contexts, for stating properties of the type theory. Of course the empty context is subsumed by the empty context with no constraints, and it is extended for abstract variables (those bound by forall and lambdas) in the obvious way:

$$\frac{}{\cdot \trianglelefteq \cdot \Leftarrow \emptyset} \quad \frac{\Gamma_1 \trianglelefteq \Gamma_2 \Leftarrow C_1 \quad T_1 \trianglelefteq T_2 \Leftarrow C_2}{\Gamma_1, (x : T_1) \trianglelefteq \Gamma_2, (x : T_2) \Leftarrow C_1 \cup C_2}$$

We can also specialize abstract variables to concrete variables: this is needed for showing that function application preserves typing.

$$\frac{\Gamma_1 \trianglelefteq \Gamma_2 \Leftarrow C_2 \quad T_1 \trianglelefteq T_2 \Leftarrow C_3}{\Gamma_1, (x := e : T_1 \Leftarrow C_1) \trianglelefteq \Gamma_2, (x : T_2) \Leftarrow C_1 \cup C_2 \cup C_3}$$

Context subsumption can also specialize the type of a concrete variable (which only occurs if an annotation is changed or removed), but its value cannot change since it might be involved in unification constraints.

$$\frac{\Gamma_1 \trianglelefteq \Gamma_2 \Leftarrow C_3 \quad (T_1 \Leftarrow C_1) \trianglelefteq (T_2 \Leftarrow C_2) \Leftarrow C_4}{\Gamma_1, (x := e : T_1 \Leftarrow C_1) \trianglelefteq \Gamma_2, (x := e : T_2 \Leftarrow C_2) \Leftarrow C_3 \cup C_4}$$

Similarly for convenience, subsumption is extended to pairs of types combined with the constraints they need to typecheck:

$$\frac{T_1 \trianglelefteq T_2 \Leftarrow C_3 \quad C_1 \subseteq C_2 \cup C_3}{(T_1 \Leftarrow C_1) \trianglelefteq (T_2 \Leftarrow C_2) \Leftarrow C_3 \cup ((C_2 \cup C_3) \setminus C_1)}$$

The intuition here is that the constraints C_3 determine how T_1 and T_2 must be compatible (since they might mention different metavariables that need to be aligned somehow), and this forces. Ugh maybe we want to know what variables are bound. Or emit $C_1 \setminus (C_2 \cup C_3)$ to answer “what needs to be added to make C_1 satisfied”. Maybe it’s okay to make it semantic like that because it’s only a meta property.

Examples:

$$\frac{\text{Universe } u \Leftarrow \text{Universe } v \Leftarrow \{u \leq v\}}{(\text{Universe } u \Leftarrow \{u \geq 4\}) \trianglelefteq (\text{Universe } v \Leftarrow \{v \leq 2\}) \Leftarrow \text{false}}$$

$$\text{Universe } u \equiv \text{Universe } v \mapsto \text{Universe } \max(u, v) \Leftarrow \{u = v\}$$

$$\text{Universe } u \trianglelefteq \text{Universe } v \Leftarrow \{u \leq v\}$$

Row type subsumption requires the *same* labels in each row, merely subsuming the corresponding types (covariantly). This is because the row merge operators in Dhall would have different behavior depending on whether subsumption is performed on their result or their operands.

$$\frac{\frac{T_1 \trianglelefteq E_1 \Leftarrow C_1 \quad \dots \quad T_n \trianglelefteq E_n \Leftarrow C_n}{\{l_1 : T_1, \dots, l_n : T_n\} \trianglelefteq \{l_1 : E_1, \dots, l_n : E_n\} \Leftarrow C_1 \cup \dots \cup C_n}}{< l_1 : T_1, \dots, l_n : T_n > \trianglelefteq < l_1 : E_1, \dots, l_n : E_n > \Leftarrow C_1 \cup \dots \cup C_n}$$

Functions are contravariant in their domain and covariant in their codomain.

$$\frac{A_2 \trianglelefteq A_1 \Leftarrow C_1 \quad B_1 \trianglelefteq B_2 \Leftarrow C_2}{\forall (x : A_1) \rightarrow B_1 \trianglelefteq \forall (x : A_2) \rightarrow B_2 \Leftarrow C_1 \cup C_2}$$

The list functor preserves subsumption:

$$\frac{T_1 \trianglelefteq T_2 \Leftarrow C_1}{\text{List } T_1 \trianglelefteq \text{List } T_2 \Leftarrow C_1}$$

Natural is only subsumed by itself:

$$\overline{\text{Natural} \trianglelefteq \text{Natural} \Leftarrow \emptyset}$$

Helpers for keeping track of free and bound metavariables, in terms, universe expressions, and row expressions:

$$\text{metavariables}(\text{Universe } u) = \text{metavariables}_u(u)$$

$$\text{metavariables}_u(x) = \{ x \}$$

$$\text{metavariables}_u(u + k) = \text{metavariables}_u(u)$$

$$\text{metavariables}_u(\max(u, v)) = \text{metavariables}_u(u) \cup \text{metavariables}_u(v)$$

$$\text{metavariables}_u(\text{if}(u, v)) = \text{metavariables}_u(u) \cup \text{metavariables}_u(v)$$

...

$$\text{metavariables}(\{ \dots r \}) = \text{metavariables}_r(r)$$

$$\text{metavariables}_r(x) = \{ x \}$$

$$\text{metavariables}_r((l : T_1, \dots r)) = \text{metavariables}(T_1) \cup \text{metavariables}_r(r)$$

...

$$\text{metavariables}(e_1 e_2) = \text{metavariables}(e_1) \cup \text{metavariables}(e_2)$$

$$\text{bound}(\cdot) = \emptyset$$

$$\text{bound}(\Gamma, x : T) = \text{bound}(\Gamma) \cup \text{metavariables}(T)$$

$$\text{bound}(\Gamma, x := e : T \Leftarrow C) = \text{bound}(\Gamma)$$

It is extended in the obvious way to the remaining cases.

A.5 Evaluation

Evaluation is extended to take place in a context, simply by substituting let-bound variables.

$$\frac{e_1 \rightarrow e_2}{\cdot \vdash e_1 \rightarrow e_2} \quad \frac{\Gamma \vdash e_1 \rightarrow e_2}{\Gamma, (x : T_1) \vdash e_1 \rightarrow e_2} \quad \frac{e_1[x := e_0] = e_2 \quad \Gamma \vdash e_1 \rightarrow e_3}{\Gamma, (x := e_0 : T_1 \Leftarrow C_1) \vdash e_1 \rightarrow e_3}$$

Evaluation rules need to preserve typing, so we record proofs of that right after each rule.

The most important evaluation rule is applying a lambda literal to an argument, and we go through the proof that it preserves typing in the most detail:

$$\frac{e_1 \rightarrow \lambda(x : T_1) \rightarrow e_2 \quad e_2[x := e_3] = e_4 \quad e_4 \rightarrow e_5}{e_1 e_3 \rightarrow e_5}$$

By the typing derivation of application, e_1 must have a function type and e_2 must have an argument that fits its codomain, so there are three main steps working from that hypothesis: ensuring that the type of the evaluated lambda expression matches the original type of e_1 , ensuring that the substitution is well-typed, and then the final step of evaluating the substitution.

1.

$$\frac{\frac{\Gamma \vdash e_1 e_3 : T_2[x := e_3] \Leftarrow C_2}{\Gamma \vdash e_1 : \forall(x : T_0) \rightarrow T_2 \Leftarrow C_3} \quad e_1 \rightarrow \lambda(x : T_1) \rightarrow e_2}{\Gamma \vdash \lambda(x : T_1) \rightarrow e_2 : \forall(x : T_1) \rightarrow T_3 \Leftarrow C_4} \quad T_0 \trianglelefteq T_1 \quad T_3 \trianglelefteq T_2}{\Gamma, (x : T_1) \vdash e_2 : T_3 \Leftarrow C_4}$$

2.

$$\frac{1. \quad \Gamma, (x : T_1) \vdash e_2 : T_3 \Leftarrow C_4 \quad \frac{\Gamma \vdash e_1 e_3 : T_2[x := e_3] \Leftarrow C_2}{\Gamma \vdash e_3 : T_4 \Leftarrow C_5} \quad T_4 \trianglelefteq T_0 \Leftarrow C_6}{\Gamma \vdash e_2[x := e_3] : T_2[x := e_3]}$$

3.

$$\frac{2. \quad \Gamma \vdash e_2[x := e_3] : T_2[x := e_3] \quad e_2[x := e_3] = e_4 \rightarrow e_5}{\Gamma \vdash e_5 : T_7[x := e_3]}$$

By the rule for typing function application, it must be that the type of e_4 is subsumed by the codomain of the function.

The fallback rule for function application, when the function does not normalize to a lambda yet, is well-typed by the contravariance of subsumption on the input, meaning that evaluating the function actually makes its input potentially larger:

$$\frac{e_1 \rightarrow e_3 \quad e_2 \rightarrow e_4}{e_1 e_2 \rightarrow e_3 e_4}$$

$$\frac{e_2[x := e_1] = e_3 \quad e_3 \rightarrow e_4}{\mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \rightarrow e_4}$$

The proof for this is very similar to function application, but with fewer details.

$$\frac{e_1 \rightarrow e_3}{e_1 : e_2 \rightarrow e_3}$$

This proof is trivial, since the typing assumption of $e_1 : e_2$ is that the intrinsic type of e_1 is subsumed by e_2 via constraints already included in the typing judgment.

$$\frac{e_1 \rightarrow e_2}{\{x = e_1, \dots\}.x \rightarrow e_2}$$

Instead of laboring through the syntax for the row operators, we describe their behavior in prose. The right-biased merge operators \parallel (for values) and $\parallel\!\!\parallel$ (for types) simply return the combination of the two rows, with the right side taking priority in terms of duplicates. The unbiased merge operators \wedge (for values) and \bowtie (for types) are a bit more complicated because they act recursively: if the same label appears in both sides, it must be a record (value/type respectively), and so the same operator is applied recursively to it.

$$\frac{e_1 \rightarrow \{l = e_3, \dots\} \quad e_2 \rightarrow \langle l = e_4, \dots \rangle \quad e_3 e_4 \rightarrow e_5}{\mathbf{merge} \ e_1 e_2 \rightarrow e_5}$$

$$\frac{}{e \rightarrow e}$$

Appendix B

PureScript Reference

List of datatypes, functions, typeclasses, and typeclass instances referenced in the code listings in this paper.

```
apply :: forall a b. (a -> b) -> a -> b
infixr 0 apply as $
applyFlipped :: forall a b. a -> (a -> b) -> b
infixl 1 applyFlipped as #

data Map k a
data Set k
instance (Ord k) => Semigroup (Set k)
instance (Ord k) => Monoid (Set k)
newtype SemigroupMap k a = SemigroupMap (Map k a)
instance (Ord k, Semigroup a) => Monoid (SemigroupMap k a)
Set.map :: forall a b. Ord b => (a -> b) -> Set a -> Set b
Set.insert :: forall a. Ord a => a -> Set a -> Set a
Set.member :: forall a. Ord a => a -> Set a -> Boolean
Set.delete :: forall a. Ord a => a -> Set a -> Set a
Set.singleton :: forall a. a -> Set a
Map.singleton :: forall k v. k -> v -> Map k v
Map.lookup :: forall k v. Ord k => k -> Map k v -> Maybe v
Map.isEmpty :: forall k v. Map k v -> Boolean
Map.mapMaybeWithKey :: forall k a b. Ord k =>
  (k -> a -> Maybe b) -> Map k a -> Map k b

newtype Max a = Max a
instance (Ord a) => Semigroup (Max a)
instance (Ord a) => Ord (Max a)
data NonEmpty f a = NonEmpty a (f a)
instance (Foldable f) => Foldable (NonEmpty f)
instance (Foldable f) => Foldable1 (NonEmpty f)
```

```

data Maybe a = Nothing | Just a
instance Foldable Maybe
instance (Semigroup a) => Monoid (Maybe a)
maybe :: forall a b. b -> (a -> b) -> Maybe a -> b
fromMaybe :: forall a. a -> Maybe a -> a

class Eq a where
  eq :: a -> a -> Boolean
infix 4 eq as ==
data Ordering = LT | GT | EQ
class (Eq a) <= Ord a where
  ord :: a -> a -> Ordering
lessThanOrEq :: forall a. Ord a => a -> a -> Boolean
infixl 4 lessThanOrEq as <=
greaterThanOrEq :: forall a. Ord a => a -> a -> Boolean
infixl 4 greaterThanOrEq as >=

class Semigroup m where
  append :: m -> m -> m
  infixr 5 append as <
class Semigroup m <= Monoid m where
  mempty :: m
class Foldable (t :: Type -> Type) where
  fold :: forall m. Monoid m => t m -> m
  foldMap :: forall a m. Monoid m => (a -> m) -> f a -> m
class (Foldable t) <= Foldable1 t where
  fold1 :: forall m. Semigroup m => t m -> m
  all :: forall a. (a -> Boolean) -> f a -> Boolean
class Functor (f :: Type -> Type) where
  map :: forall a b. (a -> b) -> f a -> f b
class (Functor f) <= Apply f where
  apply :: forall a b. f (a -> b) -> f a -> f b
class (Functor t, Foldable t) <= Traversable t where
  traverse :: forall a b m. Applicative m => (a -> m b) -> t a -> m (t b)
class (Foldable f) <= FoldableWithIndex i f | f -> i where
  foldMapWithIndex :: forall a m. Monoid m => (i -> a -> m) -> f a -> m
class (Functor f) <= FunctorWithIndex i f | f -> i where
  mapWithIndex :: forall a b. (i -> a -> b) -> f a -> f b
class
(
  FunctorWithIndex i t,
  FoldableWithIndex i t,
  Traversable t
) <= TraversableWithIndex i t | t -> i where
traverseWithIndex :: forall a b m. Applicative m =>
  (i -> a -> m b) -> t a -> m (t b)

```

Note: (apply maybe append) `:: forall m. Semigroup m => m -> Maybe m -> m` is an idiom equivalent to `\x y -> maybe x (append x) y` that appends the two arguments in order if the second is **Just**, otherwise it returns just the first argument. If `m` is a **Monoid**, this is equivalent to `\x y -> x \diamond fromMaybe mempty y`.

Bibliography

- [1] Fabrizio Ferrai, *Hurkens' paradox? · issue 250 · dhall-lang/dhall-lang*, Dhall-Lang, 2018.
- [2] Robert Harper and Robert Pollack, *Type checking with universes*, Theoretical Computer Science **89** (1991), no. 1, 107–136.
- [3] Antonius J. C. Hurkens, *A simplification of girard's paradox*, Proceedings of the second international conference on typed lambda calculi and applications, 1995, pp. 266–278.
- [4] Trevor Jim, *What are principal typings and what are they good for?*, Proceedings of the 23rd acm sigplan-sigact symposium on principles of programming languages, 1996, pp. 42–53.
- [5] Francesco Mazzoli and Andreas Abel, *Type checking through unification*, 2016.
- [6] Didier Rémy, *Type inference for records in natural extension of ml*, MIT Press, Cambridge, MA, USA, 1994.
- [7] Matthieu Sozeau and Nicolas Tabareau, *Universe polymorphism in coq*, Interactive theorem proving, 2014, pp. 499–514.
- [8] J. B. Wells, *The essence of principal typings*, Proceedings of the 29th international colloquium on automata, languages and programming, 2002, pp. 913–925.