A Type Theoretic Specification of Type Inference

Kenichi Asai Kyoko Kadowaki

Ochanomizu University, Japan

asai@is.ocha.ac.jp, kado@pllab.is.ocha.ac.jp

Abstract

This paper presents the complete formalization of type inference for simply-typed lambda calculus, including unification, and proves its soundness as well as completeness in the dependently-typed programming language Agda. The formalization clearly shows the interaction between allocation of new metavariables and substitution, something that was not observed in the previous formalizations. After introducing metavariables generically using the two-level types approach by Sheard and Pasalic, we extend McBride's unification algorithm to work on generic data defined in the sum-of-product form. We then define a type inference function that, given an untyped term, returns either a proof that the term is untypable or a corresponding well-typed term together with the proof that the inferred type is most general. During the proof development, we introduce a parallel relation between two inequalities, the key relation to keep the proof simple and clear. As a result, we could summarize the soundness of the type inference as typing rules that are intuitively clear but reflect all the details of the type inference in Agda.

General Terms Languages

Keywords Type inference, unification, mechanized proof, Agda, dependent type, generic programming

1. Introduction

Internal verification of programs (Altenkirch 1996; Stump 2016) utilizes types to express various properties on data structures or programs and enables us to maintain or prove those properties directly and ingeniously. With internal verification, we can keep variety of properties, such as the length of vectors, the balanced property of Braun trees, and even ordering invariants for a generic data in which most of the proofs are done automatically (McBride 2014). In his keynote talk at ICFP 2013, Ulf Norell showed us how the internal encoding of λ terms leads to a beautiful type checking algorithm.

However, internal verification has not been used for type inference so far, partly because type inference, unlike type checking, requires unification of metavariables. Although we have various nice techniques of internal verification, e.g., the automatic assurance of type soundness by writing a typed interpreter, we have not been able to use them for type inference, and hence many static analyses that are often formalized as type inference problems. Thus, one can discuss correctness of offline partial evaluation (Asai et al. 2014), but not the binding time analysis it depends on. This is unfortunate, especially in the presence of the structurally recursive unification algorithm (McBride 2003), which could form a basis for the whole development.

In this paper, we present the complete formalization of type inference for simply-typed λ-calculus, including unification, where the soundness and completeness properties are internally expressed in the type inference algorithm. The formalization is done in Agda (Norell 2008), which is based on dependent type theory (Martin-Löf 1984). Our formalization of type inference relies on three techniques: McBride's unification algorithm, generic programming, and a novel *parallel relation* between two inequalities (i.e., types).

We exploit McBride's unification algorithm to formalize type inference, thereby establishing the solid mechanized foundation of type inference that uses unification. Since we want our formalization to be applicable to various type systems, we extend McBride's unification algorithm to work on any data defined generically in the sum-of-product form. We then formalize type inference on top of it as a function from an untyped term to the corresponding well-typed term, where both soundness and completeness properties are built into types.

Since McBride's unification algorithm keeps track of the number of metavariables, a naive formalization on top of it requires many calculations on the number of metavariables, disturbing the essence of the underlying type inference algorithm.[1] To maintain the proof simple and clear, we introduce inequalities on the number of metavariables and the parallel relation that must hold between two inequalities. The parallel relation not only enables us to avoid maintaining the exact number of metavariables but also allows us to prove necessary but easy inequalities only.

The formalization clearly shows the interaction between allocation of new metavariables and substitution, something that was not observed in the previous formalizations. Although the proof itself is non-trivial, we can keep the structure of the type inference very close to type checking thanks to the parallel relation. The resulting type inference is simple enough to understand without sacrificing the fine details of the proof including the number of metavariables.

The contributions of this paper are summarized as follows.

- We extend the structurally recursive unification algorithm by McBride (2003) so that it works for arbitrary sum-of-product type of generic data.
- We formalize type inference including unification for simply-typed λ-calculus and prove its correctness in Agda.
- We introduce the parallel relation between two inequalities and show how it simplifies the correctness proof considerably.
- We clarify the interaction between allocation of new metavariables and substitution in the type inference.

[1] This is reminiscent of the proof with α-equivalence where the freshness condition disturbs the essence of the proof.

types: $t ::= \mathsf{unit} \mid t_1 \to t_2$
type environments: $\Gamma ::= \text{(empty)} \mid \Gamma, x : t$
terms: $e ::= x \mid \lambda x.\, e_1 \mid e_1\, e_2 \mid \bullet$
typing rules:

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ (\mathsf{TVar}) \qquad \frac{\Gamma, x : t_2 \vdash e_1 : t_1}{\Gamma \vdash \lambda x.\, e_1 : t_2 \to t_1}\ (\mathsf{TLam})$$

$$\frac{\Gamma \vdash e_1 : t_2 \to t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\, e_2 : t_1}\ (\mathsf{TApp}) \qquad \frac{}{\Gamma \vdash \bullet : \mathsf{unit}}\ (\mathsf{TUnit})$$

Figure 1. Simply-typed λ-calculus with unit

- We formulate the type inference as typing rules that incorporate substitution of matevariables explicitly and that precisely correspond to Agda implementation.

In the next section, we describe the uses of metavariables in type inference, and introduce metavariables to the generic programming framework. In Section 3, we extend McBride's unification algorithm to generic data and introduce the parallel relation. After defining the simply-typed λ-calculus in Agda in Section 4, we show type inference in Section 5. Related work is in Section 6 and the paper concludes in Section 7.

In Agda, lexical tokens are separated by spaces, parentheses, and braces only. All the other characters (including unicode characters) can constitute an identifier. For example, $l{\leq}l'$ is a single identifier, while $l \leq l'$ (with spaces around $\leq$) is a predicate (type) stating that l is less than or equal to l'. Although we try to explain various features of Agda as we proceed, we assume basic familiarity with Agda. For thorough introduction, see (Norell 2008) for example.

The complete Agda code is submitted as the anonymous supplementary material for interested reviewers.

2. Metavariables, Generically

When we specify a language, we use metavariables. For example, Figure 1 defines types, type environments, terms, and typing rules for the standard simply-typed λ-calculus, extended with unit $\bullet$ of type unit. In the figure, t, Γ, x, and e (possibly with subscripts) are metavariables, representing types, type environments, variables, and terms, respectively.

Metavariables play an important role in type inference. Given a type environment and a term, type inference returns the type of the term under the type environment, according to the typing rules. This view of type inference goes without problems for units, variables, and applications. To infer the type of $\lambda x.\, e_1$, however, we need to infer the type of e_1 under the type environment extended with the type of x. But what is the type of x? We don't know yet. It will be fixed during the type inference of the body e_1. To represent the yet unknown type of x, we use a metavariable, meaning that the type of x can be any type at this moment. During the type inference of the body e_1, it will be instantiated to a required type.

The presence of metavariables during type inference means that we need to somehow support metavariables in the type inference. One way to do it is to extend the grammar with metavariables. For example, we can redefine types as

$$t \quad ::= \quad \mathsf{unit} \mid t_1 \to t_2 \mid m$$

where m represents a metavariable. However, this method works only for this particular type. Since we want to implement type inference not only for the simply-typed λ-calculus but also for various other languages, we employ generic programming and introduce metavariables to any generically specified data definition. We can then define (and prove correct) unification once and for all for any generic data.

In the generic programming, data is defined in the sum-of-product form as a pattern functor and arbitrary large data is constructed by closing the recursive position by a fixed-point operator. We use a simplified version of Regular (van Noort et al. 2008) as formulated by Magalhães and Löh (2012). In Regular, a pattern functor is defined as follows:

```
data Code : Set where
  U   : Code -- unit
  I   : Code -- recursive position
  _⊕_ : (F G : Code) → Code -- sum
  _⊗_ : (F G : Code) → Code -- product
```

The type Code is the type of pattern functors. The first two constructors, U and I, represent unit and a recursive position. In Agda, texts after -- up to the end of line are comments. The latter two constructors are for sum and product. In Agda, underscores show the position of the arguments. Thus, _⊕_ and _⊗_ are infix operators. We assume that _⊗_ has higher operator precedence than _⊕_. For example, the pattern functor for the simple types becomes as follows:

```
TypeF : Code
TypeF = U ⊕ I ⊗ I
```

where U is for unit and two I's are the argument and return types of a function type.

Before constructing an arbitrary large (recursive) data, we relate a pattern functor with an Agda type, by defining the following interpretation function:

```
⟦_⟧ : (F : Code) → (A : Set) → Set
⟦ U ⟧ A = ⊤
⟦ I ⟧ A = A
⟦ F ⊕ G ⟧ A = ⟦ F ⟧ A ⊎ ⟦ G ⟧ A
⟦ F ⊗ G ⟧ A = ⟦ F ⟧ A × ⟦ G ⟧ A
```

Given a pattern functor F and a type A representing the interpretation of the recursive position, the interpretation function returns a corresponding Agda type. The constructor U is mapped to the Agda unit type ⊤, which has a single inhabitant tt. The recursive position I is mapped to the supplied type A. The sum _⊕_ is mapped to the disjoint sum type ⊎ in Agda, which comes with two injection functions inj_1 and inj_2. Finally, the product _⊗_ is mapped to the non-dependent product type × in Agda, whose constructor is _,_.

Using the interpretation function, we can now create a recursive data, following the two-level types approach by Sheard and Pasalic (2004).

```
data μ (F : Code) (m : ℕ) : Set where
  ⟨_⟩ : ⟦ F ⟧ (μ F m) → μ F m -- fixed point
  ⟪_⟫ : (x : Fin m) → μ F m -- metavariable
```

Given a pattern functor F and a natural number m (having Agda type ℕ of natural numbers), $\mu\ F\ m$ represents the type of data specified by F with up to m metavariables. The first constructor ⟨_⟩ receives a value of type ⟦ F ⟧ ($\mu\ F\ m$) to create a value of type $\mu\ F\ m$. Notice that the recursive position of ⟦ F ⟧ is filled with $\mu\ F\ m$ itself, tying the knot of recursion. By supplying a data of type $\mu\ F\ m$ at the recursive position, we can construct an arbitrarily large recursive data.

Before presenting examples of simple types, we define the following constructor-like functions to avoid writing injection functions:

```
TUnit : {m : ℕ} → μ TypeF m
TUnit = ⟨ inj₁ tt ⟩
```

```
_⇒_ : {m : ℕ} → (t₁ t₂ : μ TypeF m) → μ TypeF m
t₁ ⇒ t₂ = ⟨ inj₂ (t₁ , t₂) ⟩
```

The parameters in the braces are implicit arguments whose values are inferred by Agda type checker. Using these functions, we can easily construct, for example, (unit → unit) → unit:

```
TypeEx₁ : μ TypeF 0
TypeEx₁ = (TUnit ⇒ TUnit) ⇒ TUnit
```

To introduce a metavariable, we use the second constructor ⟪_⟫ of $\mu\ F\ m$. Its argument is of type Fin m, which is a type of finite natural numbers from 0 to $m-1$ (constructed by zero and suc). When $m = 0$ (as in TypeEx₁), Fin 0 has no inhabitants and thus no metavariables can be used. When m is greater than 0, we can use metavariables. For example, TypeEx₂ below uses one metavariable ⟪ zero ⟫:

```
TypeEx₂ : μ TypeF 1
TypeEx₂ = (TUnit ⇒ ⟪ zero ⟫) ⇒ ⟪ zero ⟫
```

We can use more than one metavariable:

```
TypeEx₃ : μ TypeF 3
TypeEx₃ = (TUnit ⇒ ⟪ suc (suc zero) ⟫) ⇒ ⟪ zero ⟫
```

In TypeEx₃, two metavariables are used among the allowed three. We could also use ⟪ suc zero ⟫ in the type if we wanted.

The type $\mu\ F\ m$ indicates that there is *up to m* metavariables, not exactly m metavariables. Thus, if a term has type $\mu\ F\ m$, the same term has also type $\mu\ F\ m'$, for any m' greater than m:

```
TypeEx₄ : μ TypeF 4
TypeEx₄ = (TUnit ⇒ ⟪ suc (suc zero) ⟫) ⇒ ⟪ zero ⟫
```

Later, we will define a function that inject a term of type $\mu\ F\ m$ into the same term of type $\mu\ F\ m'$.

Having defined a generic data, we next define a generic function. We could first define fmap′ that applies f to all the recursive positions of a given term by descending down the data as follows:

```
fmap′ : (G : Code) → {A B : Set} →
    (f : A → B) → (⟦ G ⟧ A → ⟦ G ⟧ B)
fmap′ U f tt  = tt
fmap′ I  f d  = f d -- apply f to recursive position
fmap′ (G₁ ⊕ G₂) f (inj₁ d) = inj₁ (fmap′ G₁ f d)
fmap′ (G₁ ⊕ G₂) f (inj₂ d) = inj₂ (fmap′ G₂ f d)
fmap′ (G₁ ⊗ G₂) f (d₁ , d₂) = (fmap′ G₁ f d₁ , fmap′ G₂ f d₂)
```

We could then define a generic catamorphic function that folds over a given generic type.

```
cata′ : {F : Code} → {m : ℕ} → {A : Set} →
    (⟨f⟩ : ⟦ F ⟧ A → A) → (⟪f⟫ : Fin m → A) → (μ F m → A)
cata′ {F} ⟨f⟩ ⟪f⟫ ⟨ d ⟩ = ⟨f⟩ (fmap′ F (cata′ ⟨f⟩ ⟪f⟫) d)
cata′ {F} ⟨f⟩ ⟪f⟫ ⟪ x ⟫ = ⟪f⟫ x
```

Given a generic data of type $\mu\ F\ m$, cata′ ⟨f⟩ ⟪f⟫ computes a value of type A by first applying itself to all the recursive positions of d using fmap′. The result (of type ⟦ F ⟧ A) is then passed to ⟨f⟩ that designates how to handle each case of the pattern functor F. Since the data can be a metavariable, we supply another function ⟪f⟫ that designates what to produce for metavariables.

Although the termination of cata′ ⟨f⟩ ⟪f⟫ t is clear since the recursive call is made by fmap′ only at the recursive positions of t, the Agda termination checker does not infer this fact, because the recursive call cata′ ⟨f⟩ ⟪f⟫ textually appears without its third argument. To make the termination of cata′ clear, we instantiate the argument f of fmap′ with the recursive call cata′ ⟨f⟩ ⟪f⟫ and define these two functions mutually recursively:

```
mutual
  fmap : {F : Code} → (G : Code) → {m : ℕ} → {A : Set} →
      (⟨f⟩ : ⟦ F ⟧ A → A) → (⟪f⟫ : Fin m → A) →
      (⟦ G ⟧ (μ F m) → ⟦ G ⟧ A)
  fmap U ⟨f⟩ ⟪f⟫ tt  = tt
  fmap I  ⟨f⟩ ⟪f⟫ d  = cata ⟨f⟩ ⟪f⟫ d -- apply cata to rec. pos.
  fmap (G₁ ⊕ G₂) ⟨f⟩ ⟪f⟫ (inj₁ d) = inj₁ (fmap G₁ ⟨f⟩ ⟪f⟫ d)
  fmap (G₁ ⊕ G₂) ⟨f⟩ ⟪f⟫ (inj₂ d) = inj₂ (fmap G₂ ⟨f⟩ ⟪f⟫ d)
  fmap (G₁ ⊗ G₂) ⟨f⟩ ⟪f⟫ (d₁ , d₂) =
      (fmap G₁ ⟨f⟩ ⟪f⟫ d₁ , fmap G₂ ⟨f⟩ ⟪f⟫ d₂)

  cata : {F : Code} → {m : ℕ} → {A : Set} →
      (⟨f⟩ : ⟦ F ⟧ A → A) → (⟪f⟫ : Fin m → A) → (μ F m → A)
  cata {F} ⟨f⟩ ⟪f⟫ ⟨ d ⟩ = ⟨f⟩ (fmap F ⟨f⟩ ⟪f⟫ d)
  cata {F} ⟨f⟩ ⟪f⟫ ⟪ x ⟫ = ⟪f⟫ x
```

These definitions pass Agda's termination check, because it is now evident that the third argument to cata is strictly decreasing. This kind of mutually recursive function definitions appears many times in our development. Note that the new fmap requires two pattern functors, F and G, because we want to recurse over the pattern functor (via G) to find the recursive position, but the recursive position itself has the type that depend on the original functor (μ $F\ m$).

In this paper, we will often transform metavariables without changing the overall structure of the data. We define the following function that maps metavariables by instantiating A in the definition of cata with $\mu\ F\ m'$ and ⟨f⟩ with ⟨_⟩:

```
⟪cata⟫ : {F : Code} → {m m′ : ℕ} →
    (⟪f⟫ : Fin m → μ F m′) → (μ F m → μ F m′)
⟪cata⟫ ⟪f⟫ t = cata ⟨_⟩ ⟪f⟫ t
```

Using ⟪cata⟫, we can, for example, define lift≤ that injects a term of type $\mu\ F\ m$ into the same term of type $\mu\ F\ m'$ when $m \leq m'$:

```
lift≤ : {F : Code} → {m m′ : ℕ} → m ≤ m′ → μ F m → μ F m′
lift≤ m≤m′ t = ⟪cata⟫ (λ x → ⟪ inject≤ x m≤m′ ⟫) t
```

where inject≤ is a function that injects x of type Fin m into the same x of type Fin m' given $m \leq m'$. It is important that lift≤ is defined based on inequality $m \leq m'$ and the conclusion of lift≤ does not impose any constraint on the form of the number of metavariables. One could easily define lift+ that lifts the number of metavariables by m'.

```
lift+ : {F : Code} → {m : ℕ} →(m′ : ℕ) → μ F m → μ F (m + m′)
lift+ m′ t = ⟪cata⟫ (λ x → ⟪ inject+ m′ x ⟫) t
```

The applicability of this function, however, is severely restricted, because we can apply this function only when the goal has exactly the form $\mu\ F\ (m + m')$ for some m and m'. Suppose we have a goal of the form $\mu\ F\ (f\ 0)$ for some function f and want to prove it by lifting the number of metavariables of a term t of type $\mu\ F$ m. We cannot use lift+ directly, because $\mu\ F\ (f\ 0)$ does not have the form $\mu\ F\ (m + m')$. We have to identify that the goal has to be of the form $\mu\ F\ (m + m')$, prove that $\mu\ F\ (f\ 0)$ is actually equal to $\mu\ F\ (m + m')$ for some m', and manually replace the type of t accordingly (which is painful), before being able to use lift+ at all. This is in contrast to lift≤, which does not impose such restriction and which does not require manual replacement of a type of a term. We can directly apply lift≤ in the above case and we are left with a constraint $m \leq f\ 0$ which we can prove later. The difference between these two becomes particularly evident when we implement more complex functions such as type inference.

In addition to a generic function, we will need a generic predicate to prove properties on generic terms. To define generic predicates, we follow the same path as generic functions, but with predicates instead of sets. The following interpretation function relates a pattern functor with an Agda predicate, given a predicate P that holds for the recursive position.

```
⟦_⟧′ : (F : Code) → {R : Set} → (P : R → Set) → (d : ⟦ F ⟧ R) → Set
⟦ U ⟧′ P tt = ⊤
⟦ I  ⟧′ P d = P d
⟦ F ⊕ G ⟧′ P (inj1 d) = ⟦ F ⟧′ P d
⟦ F ⊕ G ⟧′ P (inj2 d) = ⟦ G ⟧′ P d
⟦ F ⊗ G ⟧′ P (d1 , d2) = ⟦ F ⟧′ P d1 × ⟦ G ⟧′ P d2
```

We can then define the following two functions mutually recursively.

```
mutual
  everywhere : {F : Code} → (G : Code) → {m : ℕ} → (P : μ F m → Set) →
    ⟨f⟩ : (d : ⟦ F ⟧ (μ F m)) → ⟦ F ⟧′ P d → P ⟨ d ⟩) →
    ⟪f⟫ : (x : Fin m) → P ⟪ x ⟫) →
    (d : ⟦ G ⟧ (μ F m)) → ⟦ G ⟧′ P d
  everywhere U P ⟨f⟩ ⟪f⟫ tt = tt
  everywhere I P ⟨f⟩ ⟪f⟫ d = ind P ⟨f⟩ ⟪f⟫ d -- apply ind to rec. pos.
  everywhere (G1 ⊕ G2) P ⟨f⟩ ⟪f⟫ (inj1 d) = everywhere G1 P ⟨f⟩ ⟪f⟫ d
  everywhere (G1 ⊕ G2) P ⟨f⟩ ⟪f⟫ (inj2 d) = everywhere G2 P ⟨f⟩ ⟪f⟫ d
  everywhere (G1 ⊗ G2) P ⟨f⟩ ⟪f⟫ (d1 , d2) =
    (everywhere G1 P ⟨f⟩ ⟪f⟫ d1 , everywhere G2 P ⟨f⟩ ⟪f⟫ d2)

  ind : {F : Code} → {m : ℕ} → (P : μ F m → Set) →
    (⟨f⟩ : (d : ⟦ F ⟧ (μ F m)) → ⟦ F ⟧′ P d → P ⟨ d ⟩) →
    (⟪f⟫ : (x : Fin m) → P ⟪ x ⟫) → (t : μ F m) → P t
  ind {F} P ⟨f⟩ ⟪f⟫ ⟨ d ⟩ = ⟨f⟩ d (everywhere F P ⟨f⟩ ⟪f⟫ d)
  ind {F} P ⟨f⟩ ⟪f⟫ ⟪ x ⟫ = ⟪f⟫ x
```

The second one defines the induction principle that proves that a generic data t satisfies a predicate P.

3. First-order Unification, Generically

During type inference, types (possibly containing metavariables) are unified to satisfy type constraints present in the typing rules. To implement unification in type theory where all function must terminate, McBride (2003) presented a unification algorithm that is structurally recursive and hence is guaranteed to terminate.

3.1 Thick

McBride's key observation is that whenever a metavariable is instantiated, the number of (uninstantiated) metavariables reduces by one. To reduce the number of metavariables, we first define a function thick. It receives two arguments, x and y (of type Fin (suc m)), and checks whether they differ. When they do, it 'thickens' the number y at position x. Intuitively speaking, x represents the metavariable to be instantiated (and removed) and y is some other metavariable. Then, thick x y returns a new y (of type Fin m) after x is removed. Mathematically, it is defined as follows, where nothing and just are the two constructors of Agda's Maybe (option) type.

$$\mathsf{thick}\ x\ y = \begin{cases} \mathsf{just}\ y & (y < x) \\ \mathsf{nothing} & (y = x) \\ \mathsf{just}\ (y-1) & (y > x) \end{cases}$$

For example, suppose that TypeEx3 in Section 2 is obtained as a result of instantiating the metavariable ⟪ suc zero ⟫ (and hence the metavariable ⟪ suc zero ⟫ is not present in TypeEx3). Since ⟪ suc zero ⟫ is no longer used and is removed, we want to decrease the number of metavariables from three to two. To do so, we rename the metavariable bigger than ⟪ suc zero ⟫ by its predecessor. In other words, we apply thick ⟪ suc zero ⟫ to all the metavariables in TypeEx3. The result becomes as follows.

```
TypeEx5 : μ TypeF 2
TypeEx5 = (TUnit ⇒ ⟪ suc zero ⟫) ⇒ ⟪ zero ⟫
```

Observe that ⟪ zero ⟫ remains the same as in TypeEx3, but ⟪ suc (suc zero) ⟫ is changed to ⟪ suc zero ⟫, and both the metavariables have type Fin 2.

We will also use thin, the partial inverse of thick.

$$\mathsf{thin}\ x\ y' = \begin{cases} y' & (y' < x) \\ y'+1 & (y' \geq x) \end{cases}$$

For any x and y of type Fin (suc m) and y' of type Fin m, we have thin x $y' = y$ if and only if thick x y = just y'. It is useful to define a version of thick that comes with this property.

```
thick2 : {m : ℕ} → (x y : Fin (suc m)) →
  x ≡ y ⊎ Σ[ y′ ∈ Fin m ] thin x y′ ≡ y
```

In the return type, Σ[y' ∈ Fin m] thin x y' ≡ y denotes a dependent product whose first element is a number y' and second element is a proof for thin x y' ≡ y. See the accompanying code for the straightforward definitions of thick, thin, and thick2.

3.2 Occur Check

Using thick, we can define a function check that performs the occur check. Since we want to prove both the soundness and completeness of type inference, the occur check not only returns whether a variable occurs in a data, but also its proof.

```
check : {F : Code} → {m : ℕ} →
  (x : Fin (suc m)) → (t : μ F (suc m)) →
  (Σ[ C ∈ Context F (μ F (suc m)) ] plug C ⟪ x ⟫ ≡ t)
  ⊎ (Σ[ t′ ∈ μ F m ] ⟪cata⟫ (⟪_⟫ ∘ thin x) t′ ≡ t)
```

If a metavariable x occurs in a type t, check x t returns a *context* that, when filled with the variable, becomes the type t, showing that x actually occurs in t. The definition of Context and plug are standard and omitted; we extend (McBride 2003b) to cope with generic data. If a metavariable x does not occur in a type t, on the other hand, we can thicken the metavariables in t at x to produce a type t' with one less metavariables. The original term t is then expressed as thinning t' at x, ensuring that x does not actually occur in t.

```
check {F} {m} x t =
  ind (λ t → (Σ[ fs ∈ Context F (μ F (suc m)) ] plug fs ⟪ x ⟫ ≡ t)
    ⊎ (Σ[ t′ ∈ μ F m ] cata ⟨_⟩ (⟪_⟫ ∘ thin x) t′ ≡ t))
  (⟨check⟩ F x) (⟪check⟫ x) t
```

The occur check is implemented by ind, using two functions, ⟨check⟩ and ⟪check⟫, that take care of the pattern functor case and the metavariable case, respectively. The former simply traverses the generic data recursively to propagate the result of recursive positions to the call site. It is somewhat lengthy but can be straightforwardly defined, following the recursive structure of everywhere.

The real occur check is done in ⟪check⟫, using thick2.

```
⟪check⟫ : {F : Code} → {m : ℕ} → (x y : Fin (suc m)) →
  (Σ[ C ∈ Context F (μ F (suc m)) ] plug {F} C ⟪ x ⟫ ≡ ⟪ y ⟫)
  ⊎ (Σ[ t′ ∈ μ F m ] ⟪cata⟫ (⟪_⟫ ∘ thin x) t′ ≡ ⟪ y ⟫)
⟪check⟫ x y with thick2 x y
... | inj1 x≡y = inj1 ([] , (cong ⟪_⟫ x≡y)) -- x occurs in y
... | inj2 (y′ , thinxy′≡y) = inj2 (⟪ y′ ⟫ , (cong ⟪_⟫ thinxy′≡y))
```

3.3 Substitution

We next define substitution. Following McBride (2003), we represent substitution as a snoc list.

```
data AList (F : Code) : (l m : ℕ) → Set where
  anil : {m : ℕ} → AList F m m -- empty substitution
  _asnoc_/_ : {l m : ℕ} → (σ : AList F l m) → (t : μ F l) →
    (x : Fin (suc l)) → AList F (suc l) m -- maps x to t before σ
```

When a substitution σ is applied to a type t, σ replaces metavariables in t one by one, reducing the number of metavariables in t. The type AList F l m represents a substitution that transforms a type with l metavariables to a type with m metavariables. The empty substitution anil does not change the number of metavariables, and hence has type AList F m m. On the other hand, σ asnoc t / x replaces the metavariable x with t, thus reducing the number of metavariables by one, before applying the rest of the substitution σ. From the definition of AList, we can easily show that whenever σ has type AList F l m, l must be equal to or greater than m.

We can append two substitutions, ρ and σ, to obtain ρ ++ σ, which applies σ first, followed by ρ.

```
_++_ : {F : Code} → {l m n : ℕ} →
  (ρ : AList F m n) → (σ : AList F l m) → AList F l n
ρ ++ anil = ρ
ρ ++ (σ asnoc t / x) = (ρ ++ σ) asnoc t / x
```

Observe the numbers of metavariables exhibit transitivity, if we interpret AList F l m as $m \leq l$. See Figure 2 (left).

When we implement type inference, we will need to lift the number of metavariables of a substitution, too. Suppose we have a data with l metavariables and a substitution σ of type AList F l m that reduces the number of metavariables from l to m. Suppose further that we want to introduce k new metavariables, resulting in $k + l$ metavariables in total. To apply σ to the resulting data, we first need to lift σ so that it can accept $k + l$ metavariables. Since σ operates on the first l metavariables only, σ should leave the new k metavariables intact. In other words, after lifting, we need a substitution that reduces the number of metavariables from $k + l$ to $k + m$, where the new k metavariables remain the same. We can actually define a function that has such a type:

```
liftAList : {F : Code} → {l m : ℕ} → (k : ℕ) →
  (σ : AList F l m) → AList F (k + l) (k + m)
```

However, as we saw for the function lift≤ in Section 2, applicability of this function is severely restricted, because we have to not only specify the number k by ourselves but also adjust the number of metavariables manually in the return type of the function to be exactly in the forms $k + l$ and $k + m$.

It is in general not a good idea to unnecessarily constrain the return type of a function. McBride (2014, Section 4) made a similar observation. To avoid the problem, we can introduce new variables together with constraints they have to satisfy.

```
liftAList′ : {F : Code} → {l l′ m m′ : ℕ} → (k : ℕ) →
  l′ ≡ k + l → m′ ≡ k + m → (σ : AList F l m) → AList F l′ m′
```

With this definition, we do not have to adjust the return type of the function manually, but we are given separate constraints that we can satisfy later with the help of Agda type checker.

However, there still remains a problem that we have to specify the number k explicitly. Since the required number of new metavariables varies depending on which kind of terms we are working on, the need to explicitly specify the number k disturbs the structure of the proofs with unnecessary calculation of the number of metavariables. However, the exact number of new metavariables in each case is not important. We just allocate *some* metavariables as needed. Thus, what we really want to express is the inequality.

```
liftAList≤′ : {F : Code} → {l m m′ : ℕ} → (m≤m′ : m ≤ m′) →
  (σ : AList F l m) → AList F ((m′ ∸ m) + l) m′
```

Instead of specifying the difference k, we supply an inequality $m \leq m'$ and state that we lift the number of metavariables from m to m'. This definition liberates us from keeping the number of metavariables exactly. In return, however, it requires us to handle a rather complex formula $(m' \dot{-} m) + l$, where $m' \dot{-} m$ represents subtraction on natural numbers, i.e., the result becomes 0 if m is greater than m'. The formula arises because l and m are not independent. If m is raised to m', l must be raised the amount exactly the same as m is raised, namely, $m' \dot{-} m$.

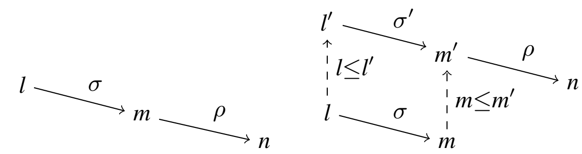

Figure 2. Concatenation of two substitutions, σ and ρ. When the numbers of metavariables match, they can be appended directly (ρ ++ σ, left). When new metavariables are allocated between the two substitutions, we need lifting of σ (ρ +⟨ *para* ⟩ σ, right, where *para* is a proof term for $l{\leq}l'$ // $m{\leq}m'$ and σ' is liftAList≤ *para* σ).

Introducing another variable with an equality constraint (as we did for liftAList′) is not a good idea, because it does not liberate us from managing the subtraction formula. What we really want to do is to introduce inequality constraints for both l and m, while maintaining the dependency between them. For this purpose, we introduce the following inductive relation between two inequalities.

```
data _//_ : {l l′ m m′ : ℕ} → l ≤ l′ → m ≤ m′ → Set where
  Refl : {m m′ : ℕ} → (m≤m′ : m ≤ m′) → m≤m′ // m≤m′
  Step : {l l′ m m′ : ℕ} → {l≤l′ : l ≤ l′} →
    {m≤m′ : m ≤ m′} → l≤l′ // m≤m′ → s≤s l≤l′ // m≤m′
```

Two inequalities, $l{\leq}l'$ and $m{\leq}m'$ where l is greater than or equal to m, are in *parallel*, written $l{\leq}l'$ // $m{\leq}m'$, when their differences are the same, i.e., $l' \dot{-} l = m' \dot{-} m$. The first constructor says that an inequality is in parallel to itself. The second constructor says that when two inequalities are in parallel, we obtain another parallel inequalities by adding 1 to both sides of the first inequality. The constructor s≤s adds 1 to both sides of the argument inequality. If $l{\leq}l'$ and $m{\leq}m'$ are in parallel, we have that l' is greater than or equal to m' by the same amount as l and m. Intuitively, l, m, l', and m' form a parallelogram as illustrated in Figure 2 (right).

Using this relation, we can finally define the lifting function for substitutions that is sufficiently easy to use.

```
liftAList≤ : {F : Code} → {l l′ m m′ : ℕ} →
  {l≤l′ : l ≤ l′} → {m≤m′ : m ≤ m′} → l≤l′ // m≤m′ →
  (σ : AList F l m) → AList F l′ m′
```

We can lift a substitution σ from AList F l m to AList F l' m', when $l{\leq}l'$ and $m{\leq}m'$ are in parallel. Notice that the conclusion of the definition does not impose any restriction on the numbers of metavariables and that it does not contain any complex formula. All the necessary constraints are embedded in the parallel relation of two inequalities.

The definition of liftAList≤ in terms of the parallel relation between two inequalities is one of the technical contributions of this paper. It not only avoids manual replacement of a type of a term, but also keeps the necessary constraints on the number of metavariables minimally. Without the definition of liftAList≤, it would have been impossible to describe type inference in the form of typing rules (as we show in Section 5) while showing all the fine details of the number of metavariables.

Using liftAList≤, we can define more flexible version of the substitution concatenation that lifts the second substitution (the one that is applied first) before concatenation, which we use in the subsequent development.

```
_+⟨_⟩_ : {F : Code} → {l m l′ m′ n : ℕ} →
  {l≤l′ : l ≤ l′} → {m≤m′ : m ≤ m′} →
  (ρ : AList F m′ n) → l≤l′ // m≤m′ → (σ : AList F l m) →
  AList F l′ n
ρ +⟨ para ⟩ σ = ρ ++ liftAList≤ para σ
```

In this definition, m' in the type of ρ does not have to be exactly the same as m in the type of σ, as long as m' is greater than or equal to m. Within the angle brackets, we specify the parallel relation that must hold to connect the two substitutions. Note that the numbers of metavariables still exhibit transitivity, intertwined with the parallel relation, as depicted in Figure 2.

3.4 Unifier

To apply a substitution to a generic data, we first turn the substitution into a unifier, an Agda function from metavariables to generic data.

```
sub : {F : Code} → {m m′ : ℕ} →
  (σ : AList F m m′) → Fin m → μ F m′
sub anil = ⟪_⟫
sub (σ asnoc t / x) = ⟪cata⟫ (sub σ) ∘ (t for x)
```

It decomposes a substitution, turns each element to an Agda function, and composes the results using the function composition operator ∘. Here, *t* for *x* is a function that maps a metavariable *y* to *t* if *y* is equal to *x*, and otherwise thickens *y*.

```
_for_ : {F : Code} → {m : ℕ} →
  (t : μ F m) → (x : Fin (suc m)) → Fin (suc m) → μ F m
(t for x) y with thick x y
... | nothing = t -- x = y
... | just y′ = ⟪ y′ ⟫ -- y is thickened to y′
```

Once a substitution is turned into a unifier, we can use ⟪cata⟫ to apply it to all the metavariables in a data. Since we often want to lift the number of metavariables before applying a unifier, it is handy to define the following three functions.

```
⟪cata⟫≤ ⟪f⟫ m≤m′′ t = ⟪cata⟫ ⟪f⟫ (lift≤ m≤m′′ t)
applySub σ t = ⟪cata⟫ (sub σ) t
applySub≤ σ m≤m′′ t = applySub σ (lift≤ m≤m′′ t)
```

3.5 Unification

We are now ready to define unification.

```
mgu : {F : Code} → {m : ℕ} → (t1 t2 : μ F m) →
  ununifiable t1 t2
  ⊎ (Σ[ m′ ∈ ℕ ] Σ[ σ ∈ AList F m m′ ] mg t1 t2 (sub σ))
```

The function mgu takes two data, t_1 and t_2, both with m metavariables, and returns either a proof that t_1 and t_2 are not unifiable or a unifying substitution σ together with the proof that sub σ is the most general unifier for them.

Two data t_1 and t_2 are ununifiable, when there exists no unifier that unifies them.

```
ununifiable : {F : Code} → {m : ℕ} → (t1 t2 : μ F m) → Set
ununifiable {F} {m} t1 t2 =
  (l′ l′′ : ℕ) → (m≤l′′ : m ≤ l′′) → (f0 : Fin l′′ → μ F l′) →
  ¬ (⟪cata⟫≤ f0 m≤l′′ t1 ≡ ⟪cata⟫≤ f0 m≤l′′ t2)
```

Note that we consider not only unifiers for m metavariables but also any unifiers that accept more than m metavariables. We have to take such unifiers into account, because they arise as the result of introducing new metavariables during type inference.

A unifier g is the most general unifier for t_1 and t_2, if (1) g unifies t_1 and t_2 and (2) any unifier f that unifies t_1 and t_2 can be represented as the composition of some f' and possibly lifted g for any x in the first m metavariables.

```
mg : {F : Code} → {m m′ : ℕ} → (t1 t2 : μ F m) →
    (g : Fin m → μ F m′) → Set
mg {F} {m} {m′} t1 t2 g =
  (⟪cata⟫ g t1 ≡ ⟪cata⟫ g t2) ×
  ((l′ l′′ : ℕ) → (f : Fin l′′ → μ F l′) → (m≤l′′ : m ≤ l′′) →
    (⟪cata⟫≤ f m≤l′′ t1 ≡ ⟪cata⟫≤ f m≤l′′ t2) →
    (Σ[ k′′ ∈ ℕ ] Σ[ m′≤k′′ ∈ m′ ≤ k′′ ] Σ[ para ∈ m≤l′′ // m′≤k′′ ]
    Σ[ f′ ∈ (Fin k′′ → μ F l′) ]
    ((x : Fin m) →
      f (inject≤ x m≤l′′) ≡ (f′ +⟨ para ⟩′ g) (inject≤ x m≤l′′))))
```

There are two subtle points in this definition. First, it includes many lifting operations to account for possible allocation of new metavariables. In particular, the number of metavariables of the input of f and f' are set in a least restrictive way. Second, and more importantly, the decomposition of f is considered only for the first m metavariables, i.e., the metavariables appearing in t_1 and t_2. By restricting the considered case to the first m metavariables only, we obtain flexibility in choosing f'. The same technique is used by Leroy (1992, page 28).

The function mgu is defined using a helper function amgu written in the accumulator passing style.

```
mgu t1 t2 with amgu t1 t2 anil
... | inj1 f rewrite ⟪-⟫-id t1 | ⟪-⟫-id t2 = inj1 f
... | inj2 (m′ , σ , mgσ) rewrite ⟪-⟫-id t1 | ⟪-⟫-id t2 = inj2 (m′ , σ , mgσ)
```

Here, ⟪-⟫-id asserts that applying an empty unifier cancels out.

```
⟪-⟫-id : {F : Code} → {m : ℕ} → (t : μ F m) → ⟪cata⟫ ⟪_⟫ t ≡ t
```

In the following, we do not explain this kind of rewrite rules that are needed to go through the proof but whose contents are unimportant. Starting from the empty substitution, amgu accumulates necessary substitution by traversing over the given data.

```
amgu : {F : Code} → {m m′ : ℕ} → (t1 t2 : μ F m) →
  (ρ : AList F m m′) →
  ununifiable (applySub ρ t1) (applySub ρ t2)
⊎ (Σ[ m′′ ∈ ℕ ] Σ[ σ ∈ AList F m′ m′′ ]
  mg (applySub ρ t1) (applySub ρ t2) (sub σ))
amgu t1 t2 anil
  rewrite ⟪-⟫-id t1 | ⟪-⟫-id t2 = amguAnil t1 t2
amgu t1 t2 (ρ asnoc t / x)
  with amgu (⟪cata⟫ (t for x) t1) (⟪cata⟫ (t for x) t2) ρ
... | inj1 f
  rewrite fuse (sub ρ) (t for x) t1 | fuse (sub ρ) (t for x) t2
  = inj1 f
... | inj2 (m′′ , σ′ , mgσ′)
  rewrite fuse (sub ρ) (t for x) t1 | fuse (sub ρ) (t for x) t2
  = inj2 (m′′ , σ′ , mgσ′)
```

The function amgu is defined by induction on m, or equivalently, the length of the substitution in the accumulator. When m is positive (i.e., when the substitution has the form σ asnoc t / x), we can reduce the number of metavariables by substituting t for x in t_1 and t_2. Note that because of the substitution, the sizes of t_1 and t_2 can become bigger than before. This is where the standard termination argument fails. We can make a recursive call here, because we keep track of the number of metavariables that is strictly decreasing.

When m is zero (when the substitution is empty), amgu delegate the task to amguAnil.

```
amguAnil : {F : Code} → {m : ℕ} → (t1 t2 : μ F m) →
  ununifiable t1 t2
⊎ (Σ[ m′′ ∈ ℕ ] Σ[ σ ∈ AList F m m′′ ] mg t1 t2 (sub σ))
amguAnil {F} ⟨ d1 ⟩ ⟨ d2 ⟩ with amgu′ F d1 d2 anil
... | inj1 f rewrite fmap-id F d1 | fmap-id F d2
  = inj1 (ununifiable-⟨d⟩ d1 d2 f)
... | inj2 (m′ , σ′ , mgσ′) rewrite fmap-id F d1 | fmap-id F d2
  = inj2 (m′ , σ′ , mg-⟨d⟩ d1 d2 (sub σ′) mgσ′)
amguAnil ⟨ d1 ⟩ ⟪ x2 ⟫ with flexRigid x2 d1
... | inj1 f = inj1 (ununifiable-sym ⟨ d1 ⟩ ⟪ x2 ⟫ f)
... | inj2 (m′ , σ′ , mgσ′) =
  inj2 (m′ , σ′ , mg-sym ⟨ d1 ⟩ ⟪ x2 ⟫ (sub σ′) mgσ′)
amguAnil ⟪ x1 ⟫ ⟨ d2 ⟩ with flexRigid x1 d2
... | inj1 f = inj1 f
```

```
... | inj₂ (m' , σ' , mgσ') = inj₂ (m' , σ' , mgσ')
amguAnil {m = m} ⟪ x₁ ⟫ ⟪ x₂ ⟫ with flexFlex m x₁ x₂
... | (m' , σ' , mgσ') = inj₂ (m' , σ' , mgσ')
```

When the accumulator is empty, amguAnil dispatches over the shapes of t_1 and t_2. We examine each case starting from the bottom. When both are metavariables (the fourth case), we return a substitution that unifies the two metavariables using flexFlex.

```
flexFlex : {F : Code} → (m : ℕ) → (x₁ x₂ : Fin m) →
      (Σ[ m' ∈ ℕ ] Σ[ σ ∈ AList F m m' ] mg ⟪ x₁ ⟫ ⟪ x₂ ⟫ (sub σ))
flexFlex zero () x₂
flexFlex (suc m) x₁ x₂ with thick₂ x₁ x₂
... | inj₁ x₁≡x₂ rewrite x₁≡x₂ = (suc m , anil , mgAnil ⟪ x₂ ⟫)
... | inj₂ (x₂' , thinx₁x₂'≡x₂) rewrite sym thinx₁x₂'≡x₂ =
    (m , anil asnoc ⟪ x₂' ⟫ / x₁ , mgFor x₁ ⟪ x₂' ⟫)
```

When the two metavariables are the same, we return the empty substitution, together with mgAnil ⟪ x_2 ⟫ stating that the empty substitution is the most general for the same data. Otherwise, we return a substitution that unifies one of the metavariables to the thickened version of the other, reducing the number of metavariables by one. The proof mgFor x_1 ⟪ x_2' ⟫ states that anil asnoc ⟪ x_2' ⟫ / x_1 is the most general substitution for the two. (See accompanying code for completeness-related functions, such as amguAnil and mgFor, that are mostly omitted in the paper due to the lack of space.) We arbitrarily chose to map x_1 to ⟪ x_2' ⟫, but we could do the other way around.

When exactly one of the two data is a metavariable (the second and the third cases), we return a substitution that unifies the metavariable to the other data using flexRigid.

```
flexRigid : {F : Code} → {m : ℕ} → (x : Fin m) → (d : ⟦ F ⟧ (μ F m)) →
  ununifiable {F} ⟪ x ⟫ ⟨ d ⟩
  ⊎ (Σ[ m' ∈ ℕ ] Σ[ σ ∈ AList F m m' ] mg ⟪ x ⟫ ⟨ d ⟩ (sub σ))
flexRigid {m = zero} () d
flexRigid {m = suc m} x d with check x ⟨ d ⟩
... | inj₁ ([] , ())
... | inj₁ (f :: fs , eq) = inj₁ (occurred f fs x d eq) -- x occurs in ⟨ d ⟩
... | inj₂ (t' , eq) with mgFor x t'
... | mg-t'forx rewrite eq = inj₂ (m , anil asnoc t' / x , mg-t'forx)
```

If the metavariable occurs in the data, there is no unifying substitution. We show there is no unifier using occurred, in a way similar to McBride (2003b). This is one of the two cases where the unification fails. Otherwise, it returns a substitution.

Finally, the first case is when both the data are not metavariables. This case is handled by calling amgu′ and lift the result on d_1 and d_2 to ⟨ d_1 ⟩ and ⟨ d_2 ⟩. We do not show the code for amgu′, which is written as a straightforward induction similar to everywhere, but only note that it requires us to propagate the results on subparts up to the whole data along the following line.

- If one of the subparts is ununifiable, the whole data is ununifiable.
- If all the subparts return the most general unifiers, we can construct the most general unifier for the whole data.

These guidelines cover the cases for U, I, and ⊕. For products, we need some more consideration.

- If the first projection is ununifiable, the whole product is ununifiable. This case can be easily shown.
- If the second projection is ununifiable, the whole product is ununifiable, provided that the first projection returns the most general unifier. (If the first projection unnecessarily instantiates metavariables, it could make the second projection ununifiable.)
- If both the projections returns the most general unifier, we can compose the most general unifier for the pair.

The second and the third cases require careful proofs, which follows McBride (2003b) but extended to handle generic data. See accompanying code for details.

3.6 Concise Notation for Unification

Once we define the unification function, we do not have to remember all its internal workings. Instead, we pay attention only to how the number of metavariables changes, in addition to the standard behavior of unification, i.e., that it produces a unifying substitution. As a preparation for expressing type inference in the form of typing rules, we introduce a concise notation for unification.

$$\sigma[\downarrow^m_{m'}](t_1{}^{(m)}) \doteq \sigma[\downarrow^m_{m'}](t_2{}^{(m)})$$

This equation expresses that two terms, t_1 and t_2 both with m metavariables, can be unified by the substitution $\sigma[\downarrow^m_{m'}]$. The notation $[\downarrow^m_{m'}]$ after σ signifies that the substitution reduces the number of metavariables from m to m'. The inputs to the equation are t_1, t_2, and m (written in blue), and the outputs are m' and σ (written in red).[2] The above notation precisely and concisely captures the type of mgu including the property the returned substitution satisfies: the $\doteq$ sign indicates that the substitution unifies the two input terms.

4. Well-scoped and Well-typed Terms

The input to the type inference is an untyped term. We assume that the input term is closed and does not contain any free variables. We define such terms using de Bruijn index.

```
data WellScoped (n : ℕ) : Set where
  Unit : WellScoped n
  Var : (x : Fin n) → WellScoped n
  Lam : (s₁ : WellScoped (1 + n)) → WellScoped n
  App : (s₁ s₂ : WellScoped n) → WellScoped n
```

The argument n of WellScoped indicates how many binders the term is under. The number increases whenever we go into the body of Lam. The closedness condition is guaranteed by representing a variable as a number from 0 to $n-1$.

The output of the type inference is a typed term. We first fix the simple types as follows using the pattern functor TypeF defined in Section 2.

```
Type : (m : ℕ) → Set
Type m = μ TypeF m
```

It is a type of simple types that have at most m metavariables.

Since we use de Bruijn index, the type environment is represented as a vector of types:

```
Cxt : {m : ℕ} → ℕ → Set
Cxt {m} n = Vec (Type m) n
```

where Vec A n is a type of vectors of length n (with the constructors [] and ::) whose elements have type A.

We then define WellTyped Γ t, the type of well-typed terms of type t under Γ, using the same constructors as the well-scoped terms. (Agda allows sharing of constructors between different types. No ambiguity arises because all the Agda terms are explicitly typed.)

```
data WellTyped {m n : ℕ} (Γ : Cxt n) : Type m → Set where
  Unit : WellTyped Γ TUnit
  Var : (x : Fin n) → WellTyped Γ (lookup x Γ)
  Lam : {t₁ : Type m} → (t₂ : Type m) →
     (w₁ : WellTyped (t₂ :: Γ) t₁) → WellTyped Γ (t₂ ⇒ t₁)
```

[2] We will write which are the inputs and which are the outputs explicitly in the main text. However, it is easier to read if the paper is printed in color.

```
App : {t₁ t₂ : Type m} → (w₁ : WellTyped Γ (t₂ ⇒ t₁)) →
  (w₂ : WellTyped Γ t₂) → WellTyped Γ t₁
```

where lookup x Γ is the x'th element of Γ. This type embodies the typing rules of simply-typed λ-calculus. It consists of only well-typed terms.

Given a well-scoped term, the task of type inference is to find a corresponding well-typed term, if it exists. To relate the input and output of type inference, we can easily define a relation between a well-typed term and the corresponding well-scoped term obtained by stripping off the type information.

```
data erase {m n : ℕ} : {t : Type m} → {Γ : Cxt n}
  (w : WellTyped Γ t) → (s : WellScoped n) → Set
```

We define erase as a relation rather than a function, because we sometimes want to guess the shape of the well-typed term corresponding to a well-scoped term.

5. Type Inference

To formalize type inference, we first specify the properties it should satisfy. We define untypable Γ s, meaning that s is not typable under Γ, as follows.

```
untypable : {m n : ℕ} → (Γ : Cxt {m} n) → (s : WellScoped n) → Set
untypable {m} Γ s =
  (l′′ l′ : ℕ) → (m≤l′′ : m ≤ l′′) → (f₀ : Fin l′′ → Type l′) →
  (t₀ : Type l′) → (w₀ : WellTyped (applyFunCxt≤ f₀ m≤l′′ Γ) t₀) →
  ¬ erase w₀ s
```

We then define mgt σ Γ s t, meaning that σ gives the most general type t for s under Γ.

```
mgt : {m m′ m′′ n : ℕ} → (σ : AList TypeF m′′ m′) →
  (Γ : Cxt {m} n) → (s : WellScoped n) → (t : Type m′) → Set
mgt {m} {m′} {m′′} σ Γ s t =
  (l′′′ l′ : ℕ) → (m≤l′′′ : m ≤ l′′′) →
  (f₀ : Fin l′′′ → Type l′) → (t₀ : Type l′) →
  (w₀ : WellTyped (applyFunCxt≤ f₀ m≤l′′′ Γ) t₀) →
  erase w₀ s →
  Σ[ k′′ ∈ ℕ ] Σ[ k′′′ ∈ ℕ ] Σ[ m′≤k′′ ∈ m′ ≤ k′′ ]
  Σ[ m′′≤k′′′ ∈ m′′ ≤ k′′′ ] Σ[ para ∈ m′′≤k′′′ // m′≤k′′ ]
  Σ[ m≤k′′′ ∈ m ≤ k′′′ ] Σ[ f′ ∈ (Fin k′′ → Type l′) ]
  (((x : Fin m) →
    f₀ (inject≤ x m≤l′′′) ≡
    (f′ +⟨ para ⟩′ (sub σ)) (inject≤ x m≤k′′′)) ×
  t₀ ≡ applyUnifier≤ f′ m′≤k′′ t)
```

The definition is long, but it reads: if there exists a unifier f_0 that gives a type t_0 for s under Γ, then f_0 and t_0 are instantiations of sub σ and t, respectively. To be more precise, we can find f' such that

- f_0 is the composition of sub σ and f', possibly with lifting, and
- t_0 is obtained by applying f' to t, possibly with lifting.

We are now ready to formalize type inference. The type inference function we are going to construct has the following type.

```
infer : (m : ℕ) → {n : ℕ} → (Γ : Cxt {m} n) → (s : WellScoped n) →
  untypable Γ s
⊎ Σ[ m′′ ∈ ℕ ] Σ[ m′ ∈ ℕ ] Σ[ m≤m′′ ∈ m ≤ m′′ ]
  Σ[ σ ∈ AList TypeF m′′ m′ ] Σ[ t ∈ Type m′ ]
  Σ[ w ∈ WellTyped (applySubCxt≤ σ m≤m′′ Γ) t ]
  (erase w s × mgt σ Γ s t)
```

Given a type environment Γ that has m metavariables and a well-scope term s, the function infer returns either a proof that s is untypable under Γ or (roughly) a well-typed term w as a proof that the input term s is well typed.

To be more precise, infer returns a tuple of eight elements in the latter case. The first element m'' represents the number of required metavariables during type inference of s. Since we already use m metavariables before the type inference of s begins, m'' must be at least as big as m. This inequality is returned as the third element. The second element m' represents the number of metavariables when the type inference of s finishes. The fourth element σ is the substitution required to infer the type of s. It reduces the number of metavariables from m'' to m'. The fifth element t is the inferred type of s, which has m' metavariables. The sixth element is the well-typed term w that we want to obtain. Since some metavariables may need to be instantiated to type check s, however, w is a well-typed term of type t not under the original type environment Γ, but Γ after applying σ. In the type of w, the function applySubCxt≤ applies a given substitution σ to all the types in Γ after lifting the number of metavariables of Γ from m to m''. The seventh element shows that the obtained w is not an arbitrary well-typed term but actually the well-typed version of s. Finally, the last element is the proof that the obtained type and substitution are the most general ones for s.

In this paper, we express the soundness parts (i.e., the parts other than untypability proofs and generality proofs) of the type inference function infer as the following typing rule.

$$\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m}\ \Gamma^{(m)}) \vdash s : t^{(m')}$$

The inputs to the type inference are Γ, s, and m (written in blue), while the outputs are m'', m', σ, and t (written in red). The other outputs of infer are represented in the judgement as follows. The inequality $m{\leq}m''$ is implicitly shown as the lifting operator $\uparrow^{m''}_{m}$ applied to Γ, which is possible because $m{\leq}m''$. It lifts the number of metavariables of all the types in Γ. The well-typed term w is represented by the whole judgement: under σ-applied Γ, the term s is well typed and has type t. Finally, the erasure property is expressed by the fact that the judgement can be regarded as a well-scoped judgement if we remove the type parts. Thus, the above notation expresses precisely and concisely all the soundness properties of infer.

The function infer is defined by induction on terms.

```
infer m Γ Unit = infer-Unit m Γ
infer m Γ (Var x) = infer-Var m Γ x
infer m Γ (Lam s₁) = infer-Lam m Γ s₁
infer m Γ (App s₁ s₂) = infer-App m Γ s₁ s₂
```

Below, we will examine each case in turn. Figure 3 precisely summarizes all the cases in the form of typing rules.

5.1 Unit

The type inference for the unit case becomes as follows:

```
infer-Unit : (m : ℕ) → {n : ℕ} → (Γ : Cxt {m} n) →
  untypable Γ Unit
⊎ Σ[ m′′ ∈ ℕ ] Σ[ m′ ∈ ℕ ] Σ[ m≤m′′ ∈ m ≤ m′′ ]
  Σ[ σ ∈ AList TypeF m′′ m′ ] Σ[ t ∈ Type m′ ]
  Σ[ w ∈ WellTyped (applySubCxt≤ σ m≤m′′ Γ) t ]
  (erase w Unit × mgt σ Γ Unit t)
infer-Unit m Γ =
  inj₂ (m , m , m≤m , anil , TUnit , Unit , Unit , mg-Unit)
```

where m≤m is a proof term for m being (less than or) equal to itself. The type of infer-Unit is obtained by instantiating s in the type of infer to Unit. Since we need neither allocation of new metavariables nor substitution for the unit case, the numbers of metavariables become both m and the substitution is empty. The returned type is TUnit. Since Unit is well typed in any type environments, it trivially has the required type WellTyped (applySubCxt≤ σ $m{\leq}m$ Γ) TUnit, whose erasure is also trivially Unit as required.

Completeness can be easily shown since the empty substitution unifies two (identical) Units and any unifiers are composition of

$$\frac{}{\phi[\downarrow^{m}_{m}](\uparrow^{m}_{m}\ \Gamma^{(m)}) \vdash \bullet : \mathsf{unit}^{(m)}}\ (\mathsf{IUnit}) \qquad \frac{\Gamma^{(m)}(x) = t^{(m)}}{\phi[\downarrow^{m}_{m}](\uparrow^{m}_{m}\ \Gamma^{(m)}) \vdash x : t^{(m)}}\ (\mathsf{IVar})$$

$$\frac{\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{1+m}\ ((\uparrow^{1+m}_{m}\ \Gamma^{(m)}), x : m^{(1+m)})^{(1+m)}) \vdash s_1 : {t_1}^{(m')} \qquad m^{(1+m)} \text{ is a new metavariable}}{\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m}\ \Gamma^{(m)}) \vdash \lambda x.\, s_1 : \sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{1+m}\ m^{(1+m)}) \to {t_1}^{(m')}}\ (\mathsf{ILam})$$

$$\frac{\begin{array}{l}
\sigma_3[\downarrow^{1+m'_2}_{m'_3}](\uparrow^{1+m'_2}_{m'_2}\ (\sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1}\ \boxed{\sigma_1[\downarrow^{m''_1}_{m'_1}](\uparrow^{m''_1}_{m}\ \Gamma^{(m)}) \qquad \vdash s_1 : {t_1}^{(m'_1)}}))) \\
\sigma_3[\downarrow^{1+m'_2}_{m'_3}](\uparrow^{1+m'_2}_{m'_2}\ \boxed{\sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1}\ (\sigma_1[\downarrow^{m''_1}_{m'_1}](\uparrow^{m''_1}_{m}\ \Gamma^{(m)}))^{(m'_1)}) \vdash s_2 : {t_2}^{(m'_2)}}) \\
\sigma_3[\downarrow^{1+m'_2}_{m'_3}](\uparrow^{1+m'_2}_{m'_2}\ (\sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1}\ {t_1}^{(m'_1)}))) \doteq \sigma_3[\downarrow^{1+m'_2}_{m'_3}](\uparrow^{1+m'_2}_{m'_2}\ {t_2}^{(m'_2)} \to {m'_2}^{(1+m'_2)}) \\
{m'_2}^{(1+m'_2)} \text{ is a new metavariable} \qquad m''_1 \leq m'''_2 \,//\, m'_1 \leq m''_2 \qquad m''_2 \leq 1 + m'''_2 \,//\, m'_2 \leq 1 + m'_2 \\
\sigma_{321}[\downarrow^{1+m'''_2}_{m'_3}](\uparrow^{1+m'''_2}_{m}\ \Gamma) = \sigma_3[\downarrow^{1+m'_2}_{m'_3}](\uparrow^{1+m'_2}_{m'_2}\ (\sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1}\ (\sigma_1[\downarrow^{m''_1}_{m'_1}](\uparrow^{m''_1}_{m}\ \Gamma)))))
\end{array}}{\sigma_{321}[\downarrow^{1+m'''_2}_{m'_3}](\uparrow^{1+m'''_2}_{m}\ \Gamma^{(m)}) \vdash s_1\ s_2 : \sigma_3[\downarrow^{1+m'_2}_{m'_3}]({m'_2}^{(1+m'_2)})}\ (\mathsf{IApp})$$

Figure 3. Type inference rules

itself and the empty substitution. It is shown in the omitted code mg-Unit.

This case is summarized as (IUnit) in Figure 3, where ϕ stands for the empty substitution.

5.2 Variable

The type inference for the variable case becomes as follows:

```
infer-Var : (m : ℕ) → {n : ℕ} → (Γ : Cxt {m} n) → (x : Fin n) →
   untypable Γ (Var x)
 ⊎ Σ[ m'' ∈ ℕ ] Σ[ m' ∈ ℕ ] Σ[ m≤m'' ∈ m ≤ m'' ]
   Σ[ σ ∈ AList TypeF m'' m' ] Σ[ t ∈ Type m' ]
   Σ[ w ∈ WellTyped (applySubCxt≤ σ m≤m'' Γ) t ]
   (erase w (Var x) × mgt σ Γ (Var x) t)
infer-Var m Γ x =
   inj₂ (m , m , m≤m , anil , t , VarX , eq , mg-Var x)
   where
     t : Type m
     t = lookup x Γ
```

Again, we need neither allocation of new metavariables nor substitution. The returned type t is found in the type environment. The well-typed term VarX (whose definition is omitted) is almost Var x of type WellTyped Γ t. However, the required type for the well-typed term is WellTyped (applySubCxt≤ anil m≤m Γ) t. Thus, we need to show that applySubCxt≤ anil m≤m Γ and Γ are the same, which can be easily proved by induction on Γ.

Completeness is again easy to show (using mg-Var, omitted), because any unifiers are composition of itself and the empty substitution.

This case is summarized as (IVar) in Figure 3. The required property can be written in mathematical form as follows:

$$\phi[\downarrow^{m}_{m}](\uparrow^{m}_{m}\ \Gamma^{(m)}) = \Gamma^{(m)}$$

With this equation, (IVar) becomes identical to (TVar) ensuring that the obtained term is well typed.

5.3 Abstraction

For the abstraction case, we examine the (ILam) in Figure 3 first, because it precisely reflects (the soundness part of) the Agda code and is easier to understand.

To infer the type of an abstraction $\lambda x.\, s_1$, we infer recursively the type of its body s_1. To do so, however, we need to assign a yet unknown type to the argument x. For this purpose, we allocate a new metavariable $m^{(1+m)}$. The current number m of metavariables indicates that we have used the metavariables from 0 to $m-1$ so far. Thus, the next available metavariable is m.[3]

Because we allocate a new variable, the number of metavariables is increased by one. To accommodate it, we lift the number of metavariables of Γ by one. Now that both $\uparrow^{1+m}_{m}\ \Gamma^{(m)}$ and $m^{(1+m)}$ have $1+m$ metavariables, we can form an extended type environment $(\uparrow^{1+m}_{m}\ \Gamma^{(m)}), x : m^{(1+m)}$. We infer the type of s_1 under this type environment.

When the type inference of s_1 finishes, we obtain the type t_1 of s_1 and a substitution σ together with a number m'' of required metavariables to infer t_1 and the final number m' of metavariables. They together satisfy the judgement in the premise of (ILam). Because application of a substitution to a type environment is element wise, we can rewrite the type environment in the premise of (ILam) as follows.

$$\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{1+m}\ (\uparrow^{1+m}_{m}\ \Gamma^{(m)})), x : \sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{1+m}\ m^{(1+m)})$$

By fusing the two lifting operations $\uparrow^{m''}_{1+m}\ (\uparrow^{1+m}_{m}\ \cdot)$ into one $\uparrow^{m''}_{m}\ \cdot$, it is further rewritten to the following.

$$\sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{m}\ \Gamma^{(m)}), x : \sigma[\downarrow^{m''}_{m'}](\uparrow^{m''}_{1+m}\ m^{(1+m)})$$

We can then obtain the conclusion of (ILam) using (TLam).

By transcripting the above scenario, we can define infer-Lam as follows.

```
infer-Lam : (m : ℕ) → {n : ℕ} →
   (Γ : Cxt {m} n) → (s₁ : WellScoped (1 + n)) →
   untypable Γ (Lam s₁)
 ⊎ Σ[ m'' ∈ ℕ ] Σ[ m' ∈ ℕ ] Σ[ m≤m'' ∈ m ≤ m'' ]
   Σ[ σ ∈ AList TypeF m'' m' ] Σ[ t ∈ Type m' ]
   Σ[ w ∈ WellTyped (applySubCxt≤ σ m≤m'' Γ) t ]
   (erase w (Lam s₁) × mgt σ Γ (Lam s₁) t)
```

[3] When the current number of metavariables is m, it is always the case that m is a new metavariable. Thus, the second premise of (ILam) is redundant. We keep it for better readability. It does not mean to use a gensym-like operator here.

The body of infer-Lam starts by allocating a new metavariable ⟪ fromℕ m ⟫, which we call t_2. Here, fromℕ is a function to turn an integer m to the same finite natural number m of type Fin $(1 + m)$.

```
infer-Lam m {n} Γ s₁
  with let
    t₂ : Type (1 + m)
    t₂ = ⟪ fromℕ m ⟫ -- new type variable
    Γ' : Cxt {1 + m} n
    Γ' = liftCxt 1 Γ
  in infer (1 + m) (t₂ :: Γ') s₁
```

We next lift the number of metavariables of Γ by one and call it Γ'. We then infer the type of the body s_1 in the type environment Γ' extended by t_2. Note that the recursive call to infer is made with m being increased by one.

```
... | inj₁ ill-s₁ = -- s₁ is ill-typed
  inj₁ (illtyped-Lam Γ s₁ ill-s₁)
```

If the body s_1 is ill typed, the whole term is also ill typed. To show completeness, the ill-typedness of s_1 is propagated to the ill-typedness of the whole term (using illtyped-Lam; one would need a clever trick to treat the newly allocated metavariable t_2 specially, as shown by Nazareth and Nipkow (1996)).

```
... | inj₂ (suc m'' , m' , s≤s m≤m'' , σ , t₁ , w₁ , eraseW₁≡S₁ , mgσ) =
  inj₂ (suc m'' , m' , m≤1+m'' , σ , σt₂ ⇒ t₁ ,
    LamW₁ , eraseLamW₁≡LamS₁ ,
    mg-Lam s₁ t₁ m≤m'' σ w₁ eraseW₁≡S₁ mgσ)
  where
    σt₂ : Type m'
    σt₂ = applySub≤ σ (s≤s m≤m'') t₂
```

If s_1 is well typed, we obtain a tuple with the eight elements. During this type inference, we used metavariables as many as the recursive call had used, namely, m''. We have to show that it is greater than (or equal to) m. It can be easily seen from the fact that (1) $1 + m$ is greater than m, (2) m'' is greater than or equal to $1 + m$ as the third element of the result of the recursive call shows, and the transitivity law. The type of the abstraction becomes $\sigma t_2 \Rightarrow t_1$ which has m' metavariables.

The returned well-typed term LamW₁ of type

$$\text{WellTyped}\ \underline{(\text{applySubCxt}{\leq}\ \sigma\ \text{m}{\leq}\text{m}''\ \Gamma)}\ (\sigma t_2 \Rightarrow t_1)$$

is almost Lam σt_2 w_1. However, like in the variable case, we have to adjust its type, because w_1 has type

$$\text{WellTyped}\ (\text{applySubCxt}{\leq}\ \sigma\ 1{+}\text{m}{\leq}\text{m}''\ (\text{liftCxt}\ 1\ \Gamma))\ t_1$$

and thus Lam σt_2 w_1 has type

$$\text{WellTyped}\ \underline{(\text{applySubCxt}{\leq}\ \sigma\ 1{+}\text{m}{\leq}\text{m}''\ (\text{liftCxt}\ 1\ \Gamma))}\ (\sigma t_2 \Rightarrow t_1)$$

which is different from the required type for LamW₁. We thus have to show that the two underlined type environments in the above two types are equal, which can be shown by induction on the structure of Γ and then by using the fusion property of liftings.

We can provide the seventh element eraseLamW₁≡LamS₁ by attaching the Lam constructor to both sides of the recursive result *eraseW₁≡S₁*, where we need to use the equality between the above two type environments.

Finally, the completeness is shown in mg-Lam that promote the generality $mg\sigma$ of σ for s_1 to the generality for Lam s_1.

We can observe that the properties we needed to interpret Figure 3 (such as $\uparrow^{m''}_{1+m} (\uparrow^{1+m}_{m} \cdot)$ can be fused to $\uparrow^{m''}_{m} \cdot$) must also be shown in Agda. Put differently, we can reconstruct a proof in Agda, once we precisely specify the type inference as in Figure 3. To our knowledge, type inference has not been formalized as concisely as Figure 3 without sacrificing the details to reconstruct mechanized soundness proofs.

5.4 Application

For the application case, we again examine the (IApp) in Figure 3 first. To infer the type of $s_1\, s_2$, we first infer the type of s_1 and obtain a type t_1 and a substitution σ_1 (the first box in the premise of (IApp)). We then infer the type of s_2 under $\sigma_1\,\Gamma$ to obtain t_2 and σ_2 (the second box). Roughly speaking, we want to check at this point that the type of the function part $(\sigma_2\, t_1)$ has the form $t_2 \to t$ for some type t. Since we do not know what t will be, we allocate a new metavariable. Because we have already used m'_2 metavariables at the end of type inference of s_2, we use m'_2 as the new metavariable. Now that we have allocated a new metavariable, t_2 (which has m'_2 metavariables) must be lifted by one to adjust the number of metavariables. Thus, $t_2 \to t$ should actually be written as follows.

$$\uparrow^{1+m'_2}_{m'_2} t_2{}^{(m'_2)} \to m'_2{}^{(1+m'_2)}$$

The type of the function part is more complicated. First of all, t_1 has m'_1 metavariables, but the substitution σ_2 obtained later expects a type with m''_2 metavariables. Thus, we need to lift t_1's number of metavariables from m'_1 to m''_2. Furthermore, after σ_2 is applied, one more metavariable (for t) is allocated. As a whole, $\sigma_2\, t_1$ should actually be written as follows.

$$\uparrow^{1+m'_2}_{m'_2} (\sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1} t_1{}^{(m'_1)}))$$

We can now unify these two types to obtain yet another substitution σ_3.

Since types of both s_1 and s_2 are obtained, we then try to use (TApp) to obtain the type of $s_1\, s_2$. However, the type environments in the two boxes in (IApp) are not the same, because new substitutions are obtained as the type inference proceeds. To restore their equality, we apply the substitution obtained after the type inference to each judgement. For the judgement for s_2 (the second box), σ_3 was obtained afterwards, so we apply σ_3 to the judgement for s_2. In the figure, it is represented by applying σ_3 to the box containing the judgement for s_2. It expresses to apply σ_3 to both the type environment and the type in the boxed judgement. As for judgement for s_1 (the first box), both σ_2 and σ_3 were obtained afterwards, which are applied in this order to the judgement for s_1. The two judgements now share the same environment

$$\sigma_3[\downarrow^{1+m'_2}_{m'_3}](\uparrow^{1+m'_2}_{m'_2} (\sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1} (\sigma_1[\downarrow^{m''_1}_{m'_1}](\uparrow^{m''_1}_{m} \Gamma^{(m)})))))$$

with s_1 having the type

$$\sigma_3[\downarrow^{1+m'_2}_{m'_3}](\uparrow^{1+m'_2}_{m'_2} (\sigma_2[\downarrow^{m''_2}_{m'_2}](\uparrow^{m''_2}_{m'_1} t_1{}^{(m'_1)})))$$

and s_2 having the type

$$\sigma_3[\downarrow^{1+m'_2}_{m'_3}](\uparrow^{1+m'_2}_{m'_2} t_2{}^{(m'_2)})$$

By the correctness property of σ_3, the type of s_1 is equal to

$$\sigma_3[\downarrow^{1+m'_2}_{m'_3}](\uparrow^{1+m'_2}_{m'_2} t_2{}^{(m'_2)} \to m'_2{}^{(1+m'_2)})$$

which can be rewritten to

$$\sigma_3[\downarrow^{1+m'_2}_{m'_3}](\uparrow^{1+m'_2}_{m'_2} t_2{}^{(m'_2)}) \to \sigma_3[\downarrow^{1+m'_2}_{m'_3}](m'_2{}^{(1+m'_2)})$$

Because the argument part of this type is equal to the type of s_2, we can now apply (TApp) to obtain the type of $s_1\, s_2$:

$$\sigma_3[\downarrow^{1+m'_2}_{m'_3}](m'_2{}^{(1+m'_2)})$$

as is also written in the conclusion of (IApp).

Now, what about the substitution? We have three substitutions, σ_1, σ_2, and σ_3. We want to return one substitution σ_{321} that combines them all. By carefully inspecting the numbers of metavariables of the three substitutions, σ_{321} is defined as the last premise of (IApp) in Figure 3. The relationship of the three substitutions is depicted in Figure 4.

We started the type inference of $s_1\, s_2$ with m metavariables. During the type inference of s_1, the number of metavariables was raised to m_1'', which was then reduced to m_1' by σ_1. During the type inference of s_2, the number of metavariables was raised to m_2'', which was then reduced to m_2' by σ_2. Finally, we introduced one more metavariable, raising the number of metavariables to $1 + m_2'$, which was then reduced to m_3' by σ_3. To combine the three substitutions, we first raise the number of metavariables from m to some number (written as $1 + m_2'''$ in (IApp)) that takes all the introduced metavariables during the type inference of $s_1\, s_2$ into account. Then, σ_{321} reduces it to m_3'.

At first sight, it is not immediately clear how to set the number $1 + m_2'''$. In fact, if we did not employ the parallel relation of substitutions, we *had* to devise the number by ourselves. With the parallel relation of substitutions, it is mechanically determined by the Agda type checker. From $\sigma_1[\downarrow^{m_1''}_{m_1'}]$ and the inequality $m_1' \le m_2''$, we can define σ_1' in such a way that the substitution commutes with lifting. (See the small parallelogram in Figure 4.)

$$\sigma_1'[\downarrow^{m_2'''}_{m_2''}](\uparrow^{m_2'''}_{m_1''} \cdot) = \uparrow^{m_2''}_{m_1'} (\sigma_1[\downarrow^{m_1''}_{m_1'}](\cdot))$$

Here, m_2''' is uniquely determined by the parallel relation $m_1'' \le m_2''' \,//\, m_1' \le m_2''$ (one of the premises of (IApp)), which we interpret as calculating m_2''' (output, red) from m_1'' and $m_1' \le m_2''$ (inputs, blue). In this case, m_2''' can be calculated in fact as $m_1''+m_2''-m_1'$. However, specifying an exact number is not a good idea, because we would then have to manipulate such a formula all over the places. Rather, we specify only the parallel relation and prove necessary relations as Agda type checker requires. Once we obtain σ_1' and m_2''', we can compose it with σ_2 to obtain σ_{21}.

We apply the same procedure to σ_{21} and $m_2' \le 1+m_2'$ to define σ_{21}'. (The big parallelogram in Figure 4.)

$$\sigma_{21}'[\downarrow^{1+m_2'''}_{1+m_2'}](\uparrow^{1+m_2'''}_{m_2'''} \cdot) = \uparrow^{1+m_2'}_{m_2'} (\sigma_{21}[\downarrow^{m_2'''}_{m_2'}](\cdot))$$

We have provided the concrete value $1 + m_2'''$ here, but we did not have to. It can be uniquely determined by the parallel relation $m_2''' \le 1 + m_2''' \,//\, m_2' \le 1 + m_2'$. In fact, in the Agda program, we *name* the number as *1+m₂'''* suggesting that it has the value $1 + m_2'''$, but it is merely an identifier whose value is deduced from the parallel relation. Finally, we can compose σ_{21}' and σ_3 to obtain σ_{321}.

The deduction of σ_{321} from σ_1, σ_2, and σ_3 is summarized in mathematical form as follows.

$$\begin{array}{cl}
 & \sigma_3[\downarrow^{1+m_2'}_{m_3'}](\uparrow^{1+m_2'}_{m_2'} (\sigma_2[\downarrow^{m_2''}_{m_2'}](\uparrow^{m_2''}_{m_1'} (\sigma_1[\downarrow^{m_1''}_{m_1'}](\uparrow^{m_1''}_{m} \Gamma^{(m)}))))) \\
= & \{\text{definition of } \sigma_1'\} \\
 & \sigma_3[\downarrow^{1+m_2'}_{m_3'}](\uparrow^{1+m_2'}_{m_2'} (\sigma_2[\downarrow^{m_2''}_{m_2'}](\sigma_1'[\downarrow^{m_2'''}_{m_2''}](\uparrow^{m_2'''}_{m_1''} (\uparrow^{m_1''}_{m} \Gamma^{(m)}))))) \\
= & \{\text{definition of } \sigma_{21}\text{, lifting fusion}\} \\
 & \sigma_3[\downarrow^{1+m_2'}_{m_3'}](\uparrow^{1+m_2'}_{m_2'} (\sigma_{21}[\downarrow^{m_2'''}_{m_2'}](\uparrow^{m_2'''}_{m} \Gamma^{(m)}))) \\
= & \{\text{definition of } \sigma_{21}'\} \\
 & \sigma_3[\downarrow^{1+m_2'}_{m_3'}](\sigma_{21}'[\downarrow^{1+m_2'''}_{1+m_2'}](\uparrow^{1+m_2'''}_{m_2'''} (\uparrow^{m_2'''}_{m} \Gamma^{(m)}))) \\
= & \{\text{definition of } \sigma_{321}\text{, lifting fusion}\} \\
 & \sigma_{321}[\downarrow^{1+m_2'''}_{m_3'}](\uparrow^{1+m_2'''}_{m} \Gamma^{(m)})
\end{array}$$

By transcripting the above scenario, we can define infer-App as follows.

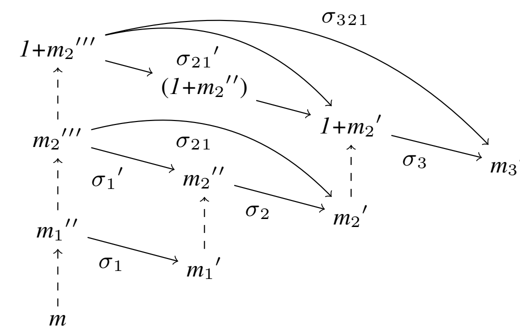

Figure 4. Relationship between substitutions in the application case

```
infer-App : (m : ℕ) → {n : ℕ} →
  (Γ : Cxt {m} n) → (s1 s2 : WellScoped n) →
  untypable Γ (App s1 s2)
⊎ Σ[ m'' ∈ ℕ ] Σ[ m' ∈ ℕ ] Σ[ m≤m'' ∈ m ≤ m'' ]
  Σ[ σ ∈ AList TypeF m'' m' ] Σ[ t ∈ Type m' ]
  Σ[ w ∈ WellTyped (applySubCxt≤ σ m≤m'' Γ) t ]
  (erase w (App s1 s2) × mgt σ Γ (App s1 s2) t)
```

We first make a recursive call to infer the type of s_1.

```
infer-App m {n} Γ s1 s2
  with infer m Γ s1
... | inj1 ill-s1 = -- s1 is ill-typed
  inj1 (illtyped-App1 Γ s1 s2 ill-s1)
... | inj2 (m1'' , m1' , m≤m1'' , σ1 , t1 , w1 , eraseW1≡S1 , mgσ1)
```

If it is untypable, we propagate the untypability of s_1 to the untypability of App s_1 s_2 (using illtyped-App₁). Otherwise, we obtain a tuple with the eight elements, in particular, the (most general) type t_1 for s_1 and the substitution σ_1.

We next apply σ_1 to Γ and infer the type of s_2 under the substituted type environment.

```
  with let
    σ1Γ : Cxt n
    σ1Γ = applySubCxt≤ σ1 m≤m1'' Γ
  in infer m1' σ1Γ s2
... | inj1 ill-s2 = -- s2 is ill-typed
  inj1 (illtyped-App2 Γ s1 s2 t1 m≤m1'' σ1 mgσ1 ill-s2)
... | inj2 (m2'' , m2' , m1'≤m2'' , σ2 , t2 , w2 , eraseW2≡S2 , mgσ2)
```

Again, if it is untypable, we propagate the untypability of s_2 to the untypability of App s_1 s_2 (using illtyped-App₂, which requires σ_1 to be most general for s_1). Otherwise, we obtain the (most general) type t_2 for s_2 and the substitution σ_2.

After the type inference of the two subterms, we allocate a new metavariable and check whether $\sigma_2 t_1$ unifies with $t_2 \Rightarrow t$.

```
  with let
    t : Type (suc m2')
    t = ⟪ fromℕ m2' ⟫ -- new type variable
    m2'≤1+m2' : m2' ≤ suc m2'
    m2'≤1+m2' = n≤m+n 1 m2'
    σ2t1 : Type m2'
    σ2t1 = applySub≤ σ2 m1'≤m2'' t1
  in mgu (lift≤ m2'≤1+m2' σ2t1) (lift≤ m2'≤1+m2' t2 ⇒ t)
... | inj1 ill-unify = inj1 (illtyped-unify Γ s1 s2 t1 m≤m1'' m1'≤m2''
  σ1 mgσ1 σ2 t2 mgσ2 ill-unify)
... | inj2 (m3' , σ3 , mgσ3)
```

If it does not, we report the untypability of App s_1 s_2 (using illtyped-unify, again requiring the two substitutions to be most general). Otherwise, we obtain the most general unifying substitution σ_3.

We next obtain the parallel relation.

```
= let
    m2''' = proj1 (σ→// σ1 m1'≤m2'')
    m1''≤m2''' = proj1 (proj2 (σ→// σ1 m1'≤m2''))
    m1''≤m2'''//m1'≤m2'' = proj2 (proj2 (σ→// σ1 m1'≤m2''))
```

Given a substitution and an inequality, the function $\sigma{\rightarrow}//$ returns (in a triple) the parallel relation that must hold together with the lifted number of metavariables and the induced inequality. We can now obtain σ_{21}:

```
in let
    σ21 : AList TypeF m2''' m2'
    σ21 = σ2 +⟨ m1''≤m2'''//m1'≤m2'' ⟩ σ1
    m2'≤1+m2' : m2' ≤ suc m2'
    m2'≤1+m2' = n≤m+n 1 m2'
    1+m2''' = proj1 (σ→// σ21 m2'≤1+m2')
    m2'''≤1+m2''' = proj1 (proj2 (σ→// σ21 m2'≤1+m2'))
    m2'''≤1+m2'''//m2'≤1+m2' = proj2 (proj2 (σ→// σ21 m2'≤1+m2'))
```

from which we obtain the second parallel relation.

Finally, we return the result of the type inference.

```
in inj2 (1+m2''' , m3' , m≤1+m2''' , σ321 , σ3t ,
    AppW1W2 , eraseAppW1W2≡AppS1S2 ,
    mg-App s1 m≤m1'' σ1 t1 w1 eraseW1≡S1 mgσ1
      s2 m1'≤m2'' σ2 t2 w2 eraseW2≡S2 mgσ2
      σ3 mgσ3)
where
    σ3t : Type m3'
    σ3t = applySub σ3 t
```

The inequality $\mathsf{m}{\leq}1{+}\mathsf{m}_2{'''}$ can be easily shown from the inequalities obtained during the type inference and the transitivity law. The returned substitution σ_{321} is defined as follows.

```
    σ321 : AList TypeF 1+m2''' m3'
    σ321 = σ3 +⟨ m2'''≤1+m2'''//m2'≤1+m2' ⟩ σ21
```

The well-typed term $\mathsf{AppW}_1\mathsf{W}_2$ is again almost $\mathsf{App}\ \sigma_3\sigma_2\mathsf{w}_1\ \sigma_3\mathsf{w}_2$, but we have to adjust their types. In particular, the type of w_1 is based on t_1 but it has to be converted to the form $t_2 \Rightarrow t$ using the correctness property of σ_3. The situation is similar for $\mathsf{eraseAppW}_1\mathsf{W}_2{\equiv}\mathsf{AppS}_1\mathsf{S}_2$. See the accompanying code for details.

Finally, to show completeness, we exploit (in mg-App) the fact that the three obtained substitutions are all most general ones.

6. Related Work

Formalization of type inference. Type inference has been formalized in various systems. Dubois and Ménissier-Morain (1999), Naraschewski and Nipkow (1999), and Urban and Nipkow (2009) formalized the algorithm W (Damas and Milner 1982) in Coq, Isabelle/HOL, and Nominal Isabelle, respectively. They proved soundness and completeness of the algorithm W. Their proofs are more general than ours in that they handle let polymorphism, while we handle only simply-typed λ-calculus. On the other hand, their formalizations do not provide the correctness proof of the unification algorithm, but are parameterized over the properties the unification must satisfy. Our formalization includes the correctness of unification, which clarifies the interaction between the allocation of new metavariables and substitution.

Garrigue (2015) proved in Coq correctness of type inference for OCaml that includes structural polymorphism and recursion. His proof is similar to ours in that the number of metavariables is used to ensure termination. He used it as a termination measure, while we exploit the structural recursion following McBride (2003). Another difference is that his proof is done using Coq tactics while our proof specifies the proof term directly, which requires struggles for clearer and simpler proofs.

Formalization of unification. The unification algorithm was first formalized by Paulson (1985) in LCF. McBride (2003) formalized unification via structural recursion with the observation that the number of metavariables reduces as the unification proceeds. The correctness of the unification algorithm is shown in (McBride 2003b). The unification algorithm in this paper is directly based on McBride's algorithm and its correctness proof. We extend McBride's work by implementing unification generically for any data, rather than simple types as used in McBride's presentation.

Ribeiro and Camarão (2015) formalized unification algorithm in Coq following the classic textbook algorithm. They use the degree of constraints, a pair of the number of metavariables and the size of types in constraints, as a termination measure.

Metavariables in generic programming. We introduced metavariables into generic programming by adopting the two-level types presented by Sheard and Pasalic (2004), extended with additional information on the number of metavariables. Similar approach was taken by Hinze et al. (2004) to implement typed-indexed data types in Haskell and van Noort et al. (2008) to implement rewriting rules that contain metavariables.

Generic programming. The generic programming we adopted is based on Regular (van Noort et al. 2008) and supports only the sum-of-product type of data and one recursive position. More flexible framework includes PolyP (Jansson and Jeuring 1997) that supports one datatype parameter, Multirec (Rodriguez et al. 2009) that supports multiple recursive positions, and Indexed functors (Löh and Magalhães 2011) that support both. Magalhães and Löh (2012) give a clear comparison between these approaches with implementation in Agda. In this paper, we have used the most basic approach. It is an interesting future work to see if we can support more flexible approaches. We expect it would not be very hard to support a datatype parameter. As for multiple recursive positions, we would have to somehow deal with multiple kinds of metavariables, one for each recursive position, and mutual recursion based on the vector of numbers of metavariables.

7. Conclusion

In this paper, we have presented the complete formalization of type inference, including unification, and proved its soundness as well as completeness in Agda. We first extended McBride's unification algorithm to work with generic data, so that the correctness of unification is established once and for all. We then formalized type inference as a function from an untyped term to a well-typed term. The parallel relation between two inequalities was the key to maintaining the number of metavariables easily. The resulting type inference function was summarized as typing rules that are intuitively clear but reflect all the details of the underlying soundness proof including the number of metavariables. Thanks to the generic programming, we can extend the input language without reimplementing unification.

As future work, we have already mentioned in the previous section the extension to more expressive generic programming. As a longer-term goal, we would like to formalize various static analyses, which are often specified as type inference problems. For example, Asai et al. (2014) formalized an offline partial evaluation, but they assumed the input language was already staged. We could augment their work by providing binding-time analysis before partial evaluation.

References

Altenkirch, T. "Integrated Verification in Type Theory," Lecture notes for a course at ESSLLI 96, Prague, 32 pages (August 1996).

Asai, K., L. Fennell, P. Thiemann, and Y. Zhang "A type theoretic specification of partial evaluation," *Proceedings of the 2014 Symposium on Principles and Practice of Declarative Programming (PPDP'14)*, pp. 57–68 (September 2014).

Damas, L. and R. Milner "Principal type-schemes for functional programs," *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, pp. 207–212 (January 1982).

Dubois, C. and V. Ménissier-Morain "Certification of a type inference tool for ML: Damas-Milner within Coq," *Journal of Automated Reasoning*, Vol. 23, No. 3, pp. 319–346 (November 1999).

Garrigue, J. "A Certified Implementation of ML with Structural Polymorphism and Recursive Types," *Mathematical Structures in Computer Science*, Vol. 25, No. 4, pp. 867–891, Cambridge University Press (May 2015).

Hinze, R., J. Jeuring, and A. Löh. "Type-indexed data types," *Science of Computer Programming*, Vol. 51, pp. 117–151, Elsevier (2004).

Jansson, P., and J. Jeuring "PolyP—a polytypic programming language extension," *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 470–482 (January 1997).

Leroy, X. "Polymorphic typing of an algorithmic language," Research Report RR-1778, INRIA, Rocquencourt (October 1992).

Löh, A., and J. P. Magalhães "Generic programming with indexed functors," *Proceedings of the ACM SIGPLAN Workshop on Generic Programming (WGP'11)*, pp. 1–12 (September 2011).

Magalhães, J. P. and A. Löh "A formal comparison of approaches to datatype-generic programming," *Proceedings of the Fourth Workshop on Mathematically Structured Functional Programming (MSFP 2012), Electronic Proceedings in Theoretical Computer Science 76*, pp. 50–67 (March 2012).

Martin-Löf, P. *Intuitionistic Type Theory,* Bibliopolis, Napoli (1984).

McBride, C. "First-order unification by structural recursion," *Journal of Functional Programming*, Vol. 13, No. 6, pp. 1061–1075, Cambridge University Press (November 2003).

McBride, C. "First-order unification by structural recursion, correctness proof" available from `http://www.strictlypositive.org/foubsr-website/`, 8 pages (October 2003b).

McBride, C. "How to keep your neighbours in order," *Proceedings of the 2014 ACM SIGPLAN International Conference on Functional Programming (ICFP'14)*, pp. 297–309 (September 2014).

Naraschewski, W. and T. Nipkow "Type inference verified: algorithm W in Isabelle/HOL," *Journal of Automated Reasoning*, Vol. 23, No. 3, pp. 299–318 (November 1999).

Nazareth, D., and T. Nipkow "Formal Verification of Algorithm W: The Monomorphic Case," In G. Goos, J. Hartmanis, J. van Leeuwen, J. von Wright, J. Grundy, and J. Harrison, editors. *Theorem Proving in Higher Order Logics (LNCS 1125)*, pp. 331–345 (August 1996).

van Noort, T., A. Rodriguez Yakushev, S. Holdermans, J. Jeuring, and B. Heeren "A lightweight approach to datatype-generic rewriting," *Proceedings of the ACM SIGPLAN Workshop on Generic Programming (WGP'08)*, pp. 13–24 (September 2008).

Norell, U. "Dependently typed programming in Agda," In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming (LNCS 5832)*, pp. 230–266 (2009).

Paulson, L. C. "Verifying the unification algorithm in LCF," *Science of Computer Programming*, Vol. 5, No. 2, pp. 143–169, Elsevier North-Holland (June 1985).

Ribeiro, R., and C. Camarão "A mechanized textbook proof of a type unification algorithm," In M. Cornélio, and B. Roscoe, editors, *Formal Methods: Foundations and Applications (LNCS 9526)*, pp. 127–141 (September 2015).

Rodriguez Yakushev, A., S. Holdermans, A. Löh, and J. Jeuring "Generic programming with fixed points for mutually recursive datatypes," *Proceedings of the 2009 ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, pp. 233–244 (August 2009).

Sheard, T., and E. Pasalic "Two-level types and parameterized modules," *Journal of Functional Programming*, Vol. 14, No. 5, pp. 547–587, Cambridge University Press (September 2004).

Stump, A. *Verified Functional Programming in Agda*, New York: Association for Computing Machinery and Morgan & Claypool (2016).

Urban, C. and T. Nipkow "Nominal verification of algorithm W," In G. Huet, J.-J. Lévy, and G. Plotkin, editors, *From Semantics to Computer Science, Essays in Honour of Gilles Kahn*, pp. 363–382, Cambridge University Press (2009).