Abbreviations, unification, and type inference

Didier Rémy

Draft version of May 6, 2020

Abstract

Modern ML-style type inference relies on an extension of first-order unification to efficiently deal with type generalization. However, ML-like type systems, and especially OCaml also allows type abbreviations, including recursive and deconstructive type abbreviations. We show how first-order unification can be extended to efficiently deal with type abbreviations and remain compatible with type inference.

Both deconstructive type abbreviations, which constrain their arguments to be of a certain shape, and projective type abbreviations, which just returns one of their argument, must is expanded eargerly, while purely constructive type abbreviations, which do not examine their parameters and construct a new type structure from their parameters, can be expanded lazily. General type abbreviations that mix both may be decomposed into a destructive part destructive part, which must be expanded eagerly and a constructive part which can still be expanded lazily.

Recursive type abbreviations are solved by encoding them into non-recursive type abbreviations that expand to recursive types.

Contents

1 Introduction **3**

2 Type abbreviations in ML **3**

3 ML-style type inference **5**
3.1 Constraints generation 5
3.2 Constraints resolution 5
3.3 Unification in plain ML 5

4 Type abbreviations **5**
4.1 Notations 5
4.2 Types 5
4.3 Ground types 6
4.4 Type abbreviations 6
4.5 Recursive definitions of type abbreviations 8
4.6 Deconstructive type abbreviations 10

5 Multi-equations **10**
5.1 Constraints and multi-equations 10
5.2 Canonical forms for constraints 10
5.3 Multiequations and views 11
5.4 Unification without deconstructive abbreviations 11
5.5 Simplification rules 12
5.6 Invariants 13

6 Unification strategy **14**
6.1 Invariants of composed rules 15
6.2 Algorithm 16
6.3 Termination 16
6.4 Efficiency of the strategy 18
6.5 Solved forms 18

7 Deconstructive type abbreviations **19**
7.1 A purely deconstructive type abbreviation 19
7.2 Mixed type abbreviations 21
7.3 The general case 21
7.4 Further Encodings 24

8 Converting between types and canonical forms **24**
8.1 Input types 24
8.2 Output types 25
8.3 Removing auxiliary type abbreviations 26
8.4 Example 27
8.5 Computing free variables 27

1 Introduction

Modern ML-style type inference relies on an extension of first-order unification to efficiently deal with type generalization. However, ML-like type systems, and especially OCaml also allows type abbreviations. In particular, OCaml allows deconstructive type abbreviations, which uses only subterms of their parameters in their expansion.

A semantically correct and trivial treatment of type abbreviations is to first expand while as a preliminary step. However, this defeats the original purpose of abbreviations, which is to improve not only writing of input types but also and especially reading of inferred types. That is, we wish to keep user-provided abbreviations as long as possible. Besides, type abbreviations may also improve efficiency of type inference by concisely representing some common patterns. This, however, requires to work with type abbreviations and expand them lazily, on demand while preserving sharing of expanded forms.

This is particularly crucial in OCaml for object types whose expanded forms can sometimes be quite large (more than an order of magnitude). Currently, OCaml already performs lazy expansion, but does not in an optimal way, and its approach has not been formalized.

We show how to efficiently extend first-order unification to deal with type abbreviations. Type abbreviations are kept in inferred types, but choice of which form—abbreviated or expanded—to show, or rather which strategy to use when printing types is left to the user.

Both deconstructive type abbreviations, which constrain their arguments to be of a certain shape, and projective type abbreviations, which just returns one of their argument, must is expanded eargerly, while purely constructive type abbreviations, which do not examine their parameters and construct a new type structure from their parameters, can be expanded lazily. General type abbreviations that mix both may be decomposed into a destructive part destructive part, which must be expanded eagerly and a constructive part which can still be expanded lazily.

Recursive type abbreviations are solved by encoding them into non-recursive type abbreviations that expand to recursive types.

2 Type abbreviations in ML

A basic type abbreviation could be of the form:

```
type var = string * int
```

This allows to write `var` instead of `string * int` just for conciseness, to avoid mistakes, or to be more readable. While abbreviations are introduced by the user in his programs,

```
let string_of_var (x : var) = (fst x) ^ string_of_int (snd x)
```

the user also expect the inferred type be printed with abbreviations:

```
    unfreeze : var -> string
```

This requires that type inference keeps track of abbreviations, even when they must be expanded during type checking to check the compatibility with expanded forms. That is, the user expects to see types as abbreviations—where they where originally abbreviations and not otherwise. In particular, every pair `string * int` should be abbreviated as `var`. For example, when writing

```
    let adult (name : string, age) = age > 18
```

the user would be surprised (and unhappy) if the inferred type were `var -> bool` instead of `string * int -> bool`.

That is, the user does not expect type inference to use abbreviation in the reserve way for contracting all types that could be printed as abbreviations (even if this could sometimes be useful in some very specific cases).

Odd usage of regular type abbreviations Type abbreviations may also be used in less usual ways to carry extra information along *phantom parameters*, i.e., parameters that do not appear in the expanded form, as in:

```
    type 'a str = string
    let one : int str = "1"
```

As these parameters are ignore, they are not decomposable and cannot be source of a clash:

```
    let f : bool str = "false"
```

Indeed, `one` of type `int str` and `f` of type `bool str` are compatible types since they both expand to `string`.

Somewhat similarly, they may also be used a marker to make a type look different while it is not—the type abbreviation marker will vanish when in conflict.

```
    type 'a special = 'a
    let one : int special = 1
    let r = one + 2
```

Although, the type `int special` of `one` looks as a special kind of an `int`, it is nothing else but `int`.

Destructive type abbreviations More interestingly, we may define abbreviations whose type parameters are constrained¿:

```
    type 'a elem = 'b constraint 'a = 'b list
```

That is, `'a elem` requires `'a` to be of the from `b list` and is then equal to `'b`. The above definition is intuitively equivalent to the follow definition (illegal in OCaml):

```
    type ('a list) elem = 'a
```

It allows to write:

```
let cons (h : 'a elem) (t : 'a) : 'a = h :: t
```

Notice than while type abbreviations have been available in OCaml for about two decades, they have never be formalized and typechecking them still raises issue in corner cases—typechecking the above expression erroneously loops in OCaml (version 4.09). Arguments of deconstructive type abbreviations may themselves be deconstructive, indeed:

```
let cons_elem (x : 'a elem elem) (t : 'a) : 'a = [x] :: t
```

Deconstructive abbreviations may still be constructive at the same time:

```
type 'a swap = 'c * 'b where 'a = 'b * 'c
let swap (x, y : 'a) : 'a swap = y, x
```

Examples with objects

Type abbreviations and modules

3 ML-style type inference

3.1 Constraints generation

3.2 Constraints resolution

3.3 Unification in plain ML

4 Type abbreviations

4.1 Notations

Given a relation $\mathcal{R}$, we write $\mathcal{R}^*$ for the reflexive transitive closure of $\mathcal{R}$ and $\mathcal{R}^\infty$ for the saturation of $\mathcal{R}$, i.e., the relation defined by

$$x \mathrel{\mathcal{R}^\infty} y \iff x \mathrel{\mathcal{R}^*} y \land \forall z, \neg(y \mathrel{\mathcal{R}} z)$$

We write $\mathcal{R}_1 \mathbin{;} \mathcal{R}_2$ for the composition of relations $\mathcal{R}_2 \circ \mathcal{R}_1$ in inverse order.

4.2 Types

We write α or β for type variables and F for type constructors, which may be primitive type constructors or type abbreviations. Primitive type constructors are such as int, bool, etc., or parameterized such as list, $*$, $\rightarrow$, etc. Types may also be recursive. Recursive types are defined with the construction $\mu(\alpha)\,\tau$ and require τ to be productive, i.e., equal to a term of the form $\mathsf{F}\bar{\sigma}$ where F is a primitive type constructor.

$$\tau, \sigma ::= \alpha \mid \mathsf{F}\bar{\tau} \mid \mu(\alpha)\,\tau$$

We call primitive types those without type abbreviations. The equality on primitive types is that on iso-recursive types.

$$\text{UNFOLDING}\quad \mu(\alpha)\,\tau = \tau[\alpha \leftarrow \mu(\alpha)\,\tau] \qquad\qquad \text{COINDUCTION}\quad \frac{\tau \text{ productive} \qquad \tau_1 = \tau[\alpha \leftarrow \tau_1] \qquad \tau_2 = \tau[\alpha \leftarrow \tau_2]}{\tau_1 = \tau_2}$$

Type equality also coincide with equality of their representation as infinite regular trees. It can be check by equality of automaton, or more efficiently by the unification algorithm.

Small types are types of depth at most 1, i.e., with type variables or a type constructor (which may be a type abbreviation) applied to a type variable.

We write $\mathcal{V}$ for be the set of type variables We write $\mathsf{ftv}(\tau)$ for the set of type variables occurring in τ. Type constructors come with a fix arity, so that all occurrences of F have the same number of arguments.

4.3 Ground types

Ground types are regular trees built out of type constructors and respecting their arities. That is, they do not contain type variables.

4.4 Type abbreviations

A type abbreviation is a constructor F that comes with a pair written $\mathsf{F}\bar{\alpha} \triangleright \sigma_{\mathsf{F}}$ where $\bar{\alpha}$ is a tuple of disjoint type variables such that $\mathsf{ftv}(\sigma_{\mathsf{F}}) \subseteq \bar{\alpha}$.

We write $\mathsf{F}_1 \prec \mathsf{F}_2$ when F_1 is a base type constructor or F_1 is a type abbreviation $\mathsf{F}_1\bar{\alpha} \triangleright \sigma_{\mathsf{F}_1}$ and F_2 does not appear in σ_{F_1}.

We assume that $\prec$ is a strict partial ordering, i.e., irreflexive and transitive. We call definitional-ordering any total ordering that extends $\prec$.

Type abbreviations define rewriting rules $\mathsf{F}\bar{\alpha} \triangleright \sigma_{\mathsf{F}}$, which can be applied in any context. This defines a rewriting system $\multimap\!\!\triangleright$, which is terminating on finite terms, as well as on syntactic terms. That is, if a regular tree is defined syntactically with the $\mu(\alpha)\,\tau$ notation, then rewriting τ with $\multimap\!\!\triangleright$ eliminates all type abbreviations in τ. Hence, the fact that syntactic terms represent regular trees does not raise any problem regarding the expansion of type abbreviations, if we perform then on the finite syntactic representation and preserve sharing, i.e., do not perform unfolding.

We write $\nabla\tau$ the full expansion of τ, i.e., the type τ' such that $\tau \multimap\!\!\triangleright^{\infty} \tau'$.

This also defines an equality $=_E$ on ground types obtained by expanding all type abbreviations and testing usual equality on the resulting canonical forms, that is:

$$\tau_1 =_E \tau_2 \iff \nabla\tau_1 = \nabla\tau_2$$

where simple equality is equality without equations but in the presence of recursive types, as usual.

We define the set of free type variables $\mathsf{ftv}(\tau)$ that appear syntactically in τ, as usual, but also the set of type variables remaining after expansion $\mathsf{ftv}^{\nabla}(\tau)$ equal to $\mathsf{ftv}(\nabla\tau)$.

Phantom and decomposable parameters In particular, type abbreviations allow to define phantom type parameters: A position i in an abbreviation $\mathsf{F}\bar{\alpha} \triangleright \sigma_F$ is phantom if $\alpha_i \notin \mathsf{ftv}^{\nabla}(\sigma)$.

A phantom occurrence is an occurrence in a term whose path contains a phantom position. We can compute $\mathsf{ftv}^{\nabla}(\tau)$ without computing $\nabla\tau$ just by not following phantom positions of type constructors. Therefore, we can also compute phantom positions of type constructors, in definitional ordering.

A position i in F is decomposable if $\mathsf{F}\bar{\tau} = \mathsf{F}\bar{\tau}'$ implies $\tau_i = \tau_i{}'$. A phantom occurrence is actually decomposable when it is not phantom. We write $\downarrow\!\mathsf{F}$ the set of decomposable positions of F.

Head expansion Consider a type abbreviation $\mathsf{F}\bar{\alpha} \triangleright \sigma_{\mathsf{F}}$. The expansion $\nabla\sigma_{\mathsf{F}}$ may be a variable or a type constructor.

If $\nabla\sigma_{\mathsf{F}}$ is a type variable β, it must be some α_i of $\bar{\alpha}$: we write $\dot{\vee}\,\mathsf{F} = i$ and say that the type abbreviation F (or $\mathsf{F}\bar{\alpha}$) is degenerate. Otherwise, $\nabla\sigma_{\mathsf{F}}$ must be of the form $\mathsf{G}\bar{\tau}$ where G a primitive constructor: we write $\dot{\vee}\,\mathsf{F} = \mathsf{G}$ and say that F is productive. Notice that when $\mathsf{F}\bar{\alpha}$ is degenerate, all its arguments but one are phantom. We extend $\dot{\vee}$ to be the identity function on primitive type constructors.

We can compute $\dot{\vee}\,\mathsf{F}$ without expanding σ_{F} by searching in σ_{F} for the topmost occurrence of a symbol G that is productive and not in at a phantom path. Then, $\dot{\vee}\,\mathsf{F}$ is equal to G if G is primitive or $\dot{\vee}\,\mathsf{G}$ otherwise.

When F is productive, the topmost constructor of σ_{F} is a type constructor F_1 and we write $\mathsf{F} \gtrdot \mathsf{F}_1$.

If $\dot{\vee}\,\mathsf{F} = \mathsf{G}$, then $\mathsf{F} \gtrdot^{\infty} \mathsf{G}$. We actually have a chain

$$\mathsf{F} = \mathsf{F}_0 \gtrdot \mathsf{F}_1 \gtrdot \ldots \mathsf{F}_n = \mathsf{G}$$

whose length n is called the rank of F and written $\mathsf{rank}\,\mathsf{F}$.

The rank can be inductively defined, without computing the chain: taking 0 for the rank of a primitive type constructor, we then have $\mathsf{rank}\,\mathsf{F} = 1 + \mathsf{rank}\,\mathsf{G}$ when $\mathsf{F} \gtrdot \mathsf{G}$. The rank is only defined for productive type abbreviations.

Notice that the relation $\gtrdot$ can be represented by a forest where roots are exactly primitive type constructors, and all constructors in a given tree have the same head-constructor—the root of the tree. When F_1 and F_2 are in the same forest, we write $\mathsf{F}_1 \vee \mathsf{F}_2$ the smallest G such that $\mathsf{F}_1 \gtrdot^{*} \mathsf{G}$ and $\mathsf{F}_2 \gtrdot^{*} \mathsf{G}$. We say that a set of type constructors S is *connected* if for any pair of type constructors F and G in S, there is a path subset $\mathsf{F}_i{}^{i\in 1..n}$ in S such that

$$\mathsf{F} = \mathsf{F}_0 \gtrdot \mathsf{F}_1 \gtrdot \ldots \mathsf{F}_n = \mathsf{G}$$

Compatibility Head expansion may also be used to check whether two type constructors F and G are compatible. Ideally, we wish $\mathsf{F} \bowtie \mathsf{G}$ to tell that two terms σ and σ' with top constructors F and G cannot be unified. Hence, the definition:

$$\mathsf{F} \bowtie_{\infty} \mathsf{G} \iff \exists\bar{\alpha}\bar{\beta} \cdot \mathsf{F}\bar{\alpha} \doteq \mathsf{G}\bar{\beta} \equiv \mathsf{false}$$

This cannot in general be computed from information coming from F and G independently, and instead need to be computed for every pair (F, G). In particular, tabulating the results so that $\bowtie$ can be checked in constant time would require space proportional to the square of the number of symbols.

In fact, we need not an exact computation of $\bowtie$, just an approximation that contains the minimal relation $\bowtie_0$ defined as the restriction of $\neq$ to primitive type constructors. This allows earlier detection of clashes—without further expansions—and might speed up when unification is more expected to fail than succeed.

That is, we need a relation $\bowtie$ such that $\bowtie_0 \subseteq \bowtie \subseteq \bowtie_\infty$. A (good) possible choice is

$$\mathsf{F} \bowtie_1 \mathsf{G} \iff \dot{\vee}\, \mathsf{F} = \mathsf{F}_0 \wedge \dot{\vee}\, \mathsf{G} = \mathsf{G}_0 \wedge \mathsf{F}_0 \bowtie_0 \mathsf{G}_0$$

which can be computed efficiently, as it only requires the precomputation of $\dot{\vee}$ for every type constructor.

4.5 Recursive definitions of type abbreviations

We wish to also allow recursive type abbreviations $(\mathsf{F}_i(\bar\alpha) = \sigma_i)^{i \in I}$ where F_j may appear in σ_i. We request however that all occurrences of F_j be applied to the same tuple of variables $\bar\alpha$. This ensures that recursion is regular. We may further assume that there is no j such that $\mathsf{F}_j \prec \mathsf{F}_i$ for all $i \neq j$. Otherwise, F_i could be defined prior to others.

Let $(\beta_i)^{i \in I}$ be a collection of fresh type variables. Let $\hat{\sigma_i}^{i \in I}$ be σ_i where each occurrence of $\mathsf{F}\bar{\alpha_i}$ has been replaced by β_i.

We defined the collection of intermediate non recursive type abbreviations $(\hat{\mathsf{F}}_i(\bar\alpha\bar\beta) = \hat\sigma_i)^{i \in I}$.

We then redefined $\mathsf{F}_i(\bar\alpha) \triangleright \mu((\beta_j = \hat{\mathsf{F}}_j(\bar\alpha, \bar\beta))^{j \in J})\, \beta_i$ for each i in I, which are all independent, as they only depend on previously defined collection $(\hat{\mathsf{F}_i})^{i \in I}$.

We have used here a general form of recursive types $\mu(\alpha_i = \sigma_i)\, \sigma$ that generalizes $\mu(\alpha)\, \sigma$—but does not add any difficulty: it stands for the type $\phi\sigma$ where ϕ is the principal solution of the equations $\alpha_i = \sigma_i$, that is if I is $1..n$ and ϕ_0 is the identity, then $\phi = \phi_n$ where $\phi_{i+i} = [\alpha_i \mapsto \mu(\alpha_i)\, \phi\sigma_i] \circ \phi$. However, the syntactic form $\mu((\beta_i = \sigma_i)^{i \in I})\, \sigma$ is more compact and we may keep as a possible syntactic form.

If we only introduce $\hat{\mathsf{F}}_i$'s symbols by expansion of F_i's, then we establish the invariant that *for any occurrence of* $\hat{\mathsf{F}}_i(\bar\sigma, \bar\tau)$*, the term* τ_j *is always* E*-equal to* $\hat{\mathsf{F}}_j(\bar\sigma, \bar\tau)$. This invariant is preserved by all transformations that preserve E-equality. Besides, when this invariant holds, we also have $\hat{\mathsf{F}}_i(\bar\sigma, \bar\tau) = \mathsf{F}_i\bar\sigma$. That is, we can see $\hat{\mathsf{F}}_i(\bar\sigma, \bar\tau)$ as a version of $\mathsf{F}_i\bar\sigma$—instrumented to enable lazy expansion of recursive occurrences. In particular, we may always read back $\mathsf{F}_i\bar\sigma$ form $\hat{\mathsf{F}}_i(\bar\sigma, \bar\tau)$, which will just loose expanded forms, and thus re-expand them again.

Example The most common case is a single recursively defined type abbreviation $\mathsf{F}\alpha \triangleright \mathsf{H}(\alpha, \mathsf{F}\alpha)$. One may think of H as (or being replaced by) a very large term, which you wish to allocate lazily.

Following, the instructions above, we define an auxiliary abbreviation $\hat{\mathsf{F}}(\alpha, \beta) \triangleright \mathsf{H}(\alpha, \beta)$ and redefine $\mathsf{F} \triangleright \mu(\beta)\, \hat{\mathsf{F}}(\alpha, \beta)$. Therefore, when $\mathsf{F}\tau$ must be expanded in two steps, this will proceed in two steps: it first expands to the recursive terms $\mu(\beta)\, \hat{\mathsf{F}}(\tau, \beta)$, which in turn can be expanded to $\mu(\beta)\, \mathsf{H}(\tau, \beta)$, as expected. Notice in particular that this enforces sharing between this expansion of $\mathsf{F}\tau$ of of its recursive instances withing τ. Compare this with the naive expansion if we had not used the auxiliary form:

$$\mathsf{F}\tau \twoheadrightarrow \mathsf{H}(\alpha, \mathsf{F}\tau) \twoheadrightarrow \mathsf{H}(\alpha, \mathsf{H}(\alpha, \mathsf{F}\tau) \twoheadrightarrow \dots$$

The intermediate abbreviation still allows to do this, but this would require "explicit" unfolding of $\mu(\beta)\, \hat{\mathsf{F}}(\alpha, \beta)$, as this is not needed during unification.

Notice that the abbreviation $\mathsf{F}\alpha \triangleright \mu(\beta)\, \hat{\mathsf{F}}(\alpha, \beta)$ is a non recursive type abbreviation, as F does not occur in $\mu(\beta)\, \hat{\mathsf{F}}(\alpha, \beta)$, but an abbreviation to a recursive type, which is perfectly fine.

For a single recursive type abbreviation, we could have directly defined $\mathsf{F}\alpha \twoheadrightarrow \mu(\beta)\, \mathsf{H}(\alpha, \beta)$, without the helper type abbreviation $\hat{\mathsf{F}}$, as there is not recursively defined abbreviation to delay the expansion of. Hence, the benefit of helper type abbreviations really appears with multiple recursively defined type abbreviations.

Consider $\mathsf{F}_1\alpha \twoheadrightarrow \mathsf{H}_1(\alpha, \mathsf{F}_2\alpha)$ and $\mathsf{F}_2\alpha \twoheadrightarrow \mathsf{H}_2(\alpha, \mathsf{F}_1\alpha)$. The encoding becomes

$$\begin{array}{ll}
\hat{\mathsf{F}}_1(\alpha, \beta_1, \beta_2) \twoheadrightarrow \mathsf{H}_1(\alpha, \beta_2) & \mathsf{F}_1\alpha \twoheadrightarrow \mu(\beta_1 = \hat{\mathsf{F}}_1(\alpha, \beta_1, \beta_2), \beta_2 = \hat{\mathsf{F}}_2(\alpha, \beta_1, \beta_2))\, \beta_1 \\
\hat{\mathsf{F}}_2(\alpha, \beta_1, \beta_2) \twoheadrightarrow \mathsf{H}_2(\alpha, \beta_1) & \mathsf{F}_2\alpha \twoheadrightarrow \mu(\beta_1 = \hat{\mathsf{F}}_1(\alpha, \beta_1, \beta_2), \beta_2 = \hat{\mathsf{F}}_2(\alpha, \beta_1, \beta_2))\, \beta_2
\end{array}$$

where

$$\begin{array}{lcl}
\mu(\beta_1 = \hat{\mathsf{F}}_1(\alpha, \beta_1, \beta_2), \beta_2 = \hat{\mathsf{F}}_2(\alpha, \beta_1, \beta_2))\, \beta_1 & = & \mu(\beta_1)\, \hat{\mathsf{F}}_1(\alpha, \beta_1, \mu(\beta_2)\, \hat{\mathsf{F}}_2(\alpha, \beta_1, \beta_2)) \\
\mu(\beta_1 = \hat{\mathsf{F}}_1(\alpha, \beta_1, \beta_2), \beta_2 = \hat{\mathsf{F}}_2(\alpha, \beta_1, \beta_2))\, \beta_2 & = & \mu(\beta_2)\, \hat{\mathsf{F}}_2(\alpha, \mu(\beta_1)\, \hat{\mathsf{F}}_1(\alpha, \beta_1, \beta_2), \beta_2)
\end{array}$$

For example, head-expanding $\mathsf{F}_1(\tau)$ leads to $\mu(\beta_1)\, \hat{\mathsf{F}}_1(\alpha, \beta_1, \mu(\beta_2)\, \hat{\mathsf{F}}_2(\alpha, \beta_1, \beta_2))$ which again head-expands into $\mu(\beta_1)\, \mathsf{H}_1(\alpha, \beta_1, \mu(\beta_2)\, \hat{\mathsf{F}}_2(\alpha, \beta_1, \beta_2))$ while leaving $\mu(\beta_2)\, \hat{\mathsf{F}}_2(\alpha, \beta_1, \beta_2)$ unexpanded.

Relaxed, regular recursive definitions We requested all occurrences of F_j to be applied to the same tuple of variables, which ensure that recursions are regular. This condition could can be relaxed, by allowing some of the parameters to be instantiated by closed types.

For instance $F\alpha = \mathsf{G}_2(\alpha, \mathsf{F}\alpha, \mathsf{F}\mathsf{G}_0)$ is regular, as it can be encoded as:

$$\hat{\mathsf{F}}(\alpha, \beta) = \mathsf{G}_2(\alpha, \mathsf{F}\alpha, \beta) \qquad \mathsf{F}\alpha = \hat{\mathsf{F}}(\alpha, \mu(\beta)\, \hat{\mathsf{F}}(\mathsf{G}_0, \beta, \beta))$$

By contrast $\mathsf{F}\alpha = \mathsf{G}_2(\alpha, \mathsf{F}(\mathsf{G}_1\alpha))$ is not regular.

Since those recursive schemas are rare and can still be encoded (manually), while the general case is involved and may be computationally costly, we do not include them in the formalization.

4.6 Deconstructive type abbreviations

Type abbreviation definitions may also deconstruct their argument, such as in $\mathsf{F}(\mathsf{G}(\alpha)) \triangleright \alpha$, or more generally, type abbreviations of the form $\mathsf{F}(\bar{\sigma}) = \sigma_{\mathsf{F}}$ where to be well-formed free type variable of σ_0 should also appear in a decomposable occurrence of $\bar{\sigma}$.

However, this generalization will be better explained once we understood the simpler cases of constructive type abbreviations, which can be categorized into degenerate or productive type abbreviations, while by contrast deconstructive type abbreviations mixes both concepts.

Hence, we will only consider them later in §7.

5 Multi-equations

5.1 Constraints and multi-equations

The grammar of constraints is

$$C ::= \mathsf{true} \mid \mathsf{false} \mid m \mid \exists \alpha \cdot C \mid C \wedge C$$

The existential quantification takes priority over conjunction of multiequations, i.e., $\exists \alpha \cdot C \wedge C$ stands for $\exists \alpha \cdot (C \wedge C)$. Conjunction of multiequations if commutative and associative.

Letter m stands for multiequations m. Multiequations are non multisets of terms We write $\doteq$ for the union of multiequations. That is, we write $\tau_1 \doteq \tau_2 \doteq \ldots \tau_n$ for multiequations.

The semantics of a constraint C is given by the set $[\![C]\!]$ of is solutions, i.e., the ground substitutions, mapping variables to ground types, satisfy all the constraints simultaneously, or equivalently by the judgment $\phi \vdash C$.

$$\phi \vdash \mathsf{true} \qquad \frac{\phi \vdash C_1 \qquad \phi \vdash C_2}{\phi \vdash C_1 \wedge C_2} \qquad \frac{\phi, \alpha \mapsto \tau \vdash C_0}{\phi \vdash \exists \alpha \cdot C} \qquad \frac{\forall \sigma_1, \sigma_2 \in m, \phi\sigma_1 =_E \phi\sigma_2}{\phi \vdash m}$$

where type equality $=_E$ is that of the equational theory.

A constraint C_1 implies a constraint C_2 and we write $C_1 \vdash C_2$ if any solution of C_1 is also a solution of C_2, i.e., $[\![C_1]\!] \subseteq [\![C_2]\!]$. Two constraints C_1 and C_2 are equivalent and we write $C_1 \equiv C_2$ they imply one another. We also write $C_1 \Rrightarrow C_2$ when we wish to orientate the equivalence for rewriting purposes.

5.2 Canonical forms for constraints

We may freely reorganize constraint by treating the conjunction of constraints $\wedge$ as associative commutative with true as neutral element and false as absorbing element, which indeed preserves the semantics. Besides, we implicitly use the extrusion of existential bindings:

$$\begin{array}{l}\textsc{ExAnd}\\ \dfrac{\alpha \# C}{(\exists \alpha \cdot C_0) \wedge C \equiv \exists \alpha \cdot (C_0 \wedge C)}\end{array}$$

The side condition $\alpha \mathrel{\#} C_2$ means that $\alpha \notin \mathsf{ftv}(C)$ where $\mathsf{ftv}(C)$ is defined in the obvious way. Equivalences $\equiv$ (or rewriting rules $\Rrightarrow$) can be applied in any context:

$$\textsc{Context-Ex}\quad \frac{C_1 \Rrightarrow C_2}{\exists\alpha \cdot C_1 \Rrightarrow \exists\alpha \cdot C_2} \qquad\qquad \textsc{Context-And}\quad \frac{C_1 \Rrightarrow C_2}{C_1 \wedge C \Rrightarrow C_2 \wedge C}$$

This allows to put constraints in canonical form $\exists\bar{\alpha} \cdot \bigwedge^{i \in I} m_i$, i.e., a conjunction of multiequations preceded by existential quantification, or one of the trivial constraint true or false.

In the following, we implicitly maintain such canonical forms after every rewriting step. In particular, existential quantifiers are automatically extruded and rewriting may be aborted to return false as soon as a subconstraint is equivalent to false.

5.3 Multiequations and views

A multiequation is a multiset of terms. A substitution is a solution of a multiequation if it equates all terms of the multiequation, as indicated above in §5.1.

We introduce additional structure to multiequations by grouping terms into views. Hence, views are themselves multisets of terms. We write ν for views. We distinguish two kinds of views: a productive view $\langle\nu\rangle$ contains productive terms that whose head-constructors form a connected set; a degenerate view $[\nu]$ contains terms that all head expands to some variable. We write $\tau \doteq \nu$ for adding term τ to the view ν.

A multiequation becomes a multiset of both isolated terms (as before) and views. Views, i.e., the partitioning of the multiequation into views, can always be dropped, leaving the multiequation as just a plain multiset of terms. In particular, a solution of a multiequation with views is just a solution of the multiequation ignoring views.

We still write $\doteq$ for union both within view and within multiequations.

5.4 Unification without deconstructive abbreviations

We describe how to solve multisets of multiequations. We will initially establish and maintain the following invariants:

- A multiequation has a unique (possibly empty) degenerate view $[\nu]$ such that $[\nu] \doteq m \equiv m$. That is, the variable head expansions of terms in productive are already in m.
- A multiequation may have zero or several productive views. We maintain a *global* invariant all terms in a productive view are equivalent, i.e., for a whole system of constraint:

$$\langle \tau \doteq \tau_0 \doteq \nu \rangle \doteq m \wedge C \quad\equiv\quad \langle \tau \doteq \nu \rangle \doteq m \wedge C$$

This equivalence is not however used to simplify constraints (from left to right), as τ_0 is kept in the view both for efficiency reasons and for better presentation of the solutions of the multiequation.

Notice that variable terms are never put in views. Non variable terms are introduced as such outside of a view, awaiting for being dispatched in either a productive or degenerate view.

5.5 Simplification rules

Stutter

$$\alpha \doteq \alpha \doteq \nu \implies \alpha \doteq \nu$$

Merge

$$\alpha \doteq [\nu_1] \doteq m_1 \wedge \alpha \doteq [\nu_2] \doteq m_2 \implies \alpha \doteq [\nu_1 \doteq \nu_2] \doteq m_1 \doteq m_2$$

Dispatch-D

$$\frac{\dot\vee\, \mathsf{F} = i}{\mathsf{F}\bar\alpha \doteq [\nu] \doteq m \implies [\mathsf{F}\bar\alpha \doteq \nu] \doteq \alpha_i \doteq m}$$

Dispatch-P

$$\frac{\dot\vee\, \mathsf{F} = \mathsf{G}}{\mathsf{F}\bar\tau \doteq m \implies \langle \mathsf{F}\bar\tau \rangle \doteq m}$$

Trim

$$\frac{\tau \notin \mathcal{V} \qquad \alpha \# \tau_1, \tau_2, m}{\mathsf{F}(\bar\tau_1, \tau, \bar\tau_2) \doteq m \implies \exists\alpha \cdot \mathsf{F}(\bar\tau_1, \alpha, \bar\tau_2) \doteq m \wedge \alpha \doteq \tau}$$

Trim-P

$$\frac{\tau \notin \mathcal{V} \qquad \alpha \# \tau_1, \tau_2, \nu, m}{\langle \mathsf{F}(\bar\tau_1, \tau, \bar\tau_2) \doteq \nu \rangle \doteq m \implies \exists\alpha \cdot \langle \mathsf{F}(\bar\tau_1, \alpha, \bar\tau_2) \doteq \nu \rangle \doteq m \wedge \alpha \doteq \tau}$$

Decompose

$$\langle \mathsf{F}\bar\alpha \doteq \nu \rangle \doteq \langle \mathsf{F}\bar\beta \doteq \nu' \rangle \doteq m \implies \langle \mathsf{F}\bar\alpha \doteq \mathsf{F}\bar\beta \doteq \nu \doteq \nu' \rangle \doteq m \wedge (\alpha_i \doteq \beta_i)^{i \in \downarrow \mathsf{F}}$$

Clean

$$\{\mathsf{F}\bar\alpha \doteq \mathsf{F}\bar\beta \doteq \nu\} \doteq m \implies \{\mathsf{F}\bar\alpha \doteq \nu\} \doteq m$$

Clash

$$\frac{\mathsf{F} \bowtie \mathsf{G}}{\langle \mathsf{F}\bar\alpha \doteq \nu \rangle \doteq \langle \mathsf{G}\bar\beta \doteq \nu' \rangle \doteq m \implies \mathsf{false}}$$

Expand

$$\frac{\mathsf{F}\bar\beta \triangleright \mathsf{G}\bar\tau}{\langle \mathsf{F}\bar\alpha \doteq \nu \rangle \doteq m \implies \exists\bar\beta \cdot \langle \mathsf{G}\bar\tau \doteq \mathsf{F}\bar\alpha \doteq \nu \rangle \doteq m \wedge_{i \in \downarrow \mathsf{F}} \alpha_i \doteq \beta_i}$$

Rule Stutter removes duplicated variables. Rule Merge merges two multi-equations that share a variable term.

Rules Dispatch-P and Dispatch-D place non-variable terms in views, according to the degenerate *vs.* productive status of the head-symbol: for degenerate terms, we also add α_i in m, which amounts to fully expand F: this is needed to enforce further merging that could otherwise remain unnoticed; this is also efficient as this kind of expansion does not allocate. By contrast, for productive abbreviations (or primitive type constructors) we just move them unexpanded in a productive view, letting expansion to be performed on demand.

Rule Trim allows to turn large terms into small ones, i.e., of high at most one. This is done position by position, but could be performed more aggressively. Trim-P is the same as Trim but operates within a productive view. This is needed, as an expansion allocates a new (large) term directly inside a productive view.

Rule Decompose applies when two productive views have a common head constructor F: these views are merged and corresponding subterms are equated along decomposable positions $\downarrow\!\mathsf{F}$. To avoid duplication of terms, decomposed terms should be small so that all subterms are variables.[1]

Rule Clean eliminates duplicates within a view. Braces mean that it applies to both degenerate and productive views.

Rule Clash detects obvious clashes based on $\bowtie$. A better (larger) relation $\bowtie$ will detect clashes earlier, avoiding further expansions. However, the least relation $\bowtie_0$ is sufficient to detect all clashes, since in the absence of clashes expansion will be performed, as long as views do not share a common head-symbol, eventually ending with primitive symbols, on which $\bowtie$ is complete.

Rule Expand applies when a term $\langle \mathsf{F}\bar\alpha \rangle$ appears in a productive view of a multiequation and F is a type abbreviation $\mathsf{F}\bar\beta \triangleright \mathsf{G}\tau$. We then add (a fresh copy of) $\mathsf{G}\tau$ to the view containing $\mathsf{F}\bar\alpha$ and add multiequations $\alpha_i \doteq \beta_i$ for all decomposable positions i in F.

5.6 Invariants

We formalize the invariants that are maintained by the unification rules. These subsume invariants informally described at the beginning of §5.4.

The invariants applies to a constraint $\exists\alpha \cdot C$ in canonical form, i.e., C is a conjunction of multiequations.

Invariant 1 *If C contains $[\tau \doteq \nu] = m$, then τ is of the form $\mathsf{F}\bar\sigma$ and $\dot\vee\, \mathsf{F} = i$.*

Invariant 2 *If C is $[\nu] \doteq m \wedge C_0$, then $C \equiv m \wedge C_0$,*

Rule Dispatch-D, which is the only one to add a term τ of the form $C\bar\sigma$ to the degenerate view, preserve both invariants: it only applies when $\dot\vee\, C = i$ and it inserts the expansion α_i in m. Rule Merge, which merges two degenerate views ν_1 and ν_2 preserves the invariant. Rule Clean relies on the invariant 2 to preserve the equivalence.

Invariant 3 *If C contains $\langle\nu\rangle \doteq m$, then m is nonempty and for all τ in m, τ is of the form $\mathsf{F}\bar\sigma$ and $\dot\vee\, \tau = \mathsf{G}$.*

[1]Notice, that only productive constructors are decomposed. Degenerate type constructors are decomposable along their non phantom position, but this will never be used. One could think that we miss an opportunity for efficient simplification, but this is not the case because degenerate type constructors must be eagerly expanded anyway and their expansion—a variable, is a quite efficient to compute and represent.

Invariant 4 *If C is $\langle \nu \doteq m \rangle \wedge C_0$ and ν' is a non-empty subset of ν then $C \equiv \langle \nu' \doteq m \rangle \wedge C_0$.*

Rule Dispatch-P is the only one that creates a new productive (singleton) view $\langle \tau \rangle$ when τ is of the form $\mathsf{F}\bar{\sigma}$ and $\curlyvee \mathsf{F}$ is a constructor G, so F is productive and the new view $\langle \tau \rangle$ satisfies the invariant 4. That view also satisfies 3 since it has no strict non-empty subview. Rule Expand increases a productive view. It introduces a new term $\mathsf{G}\bar{\tau}$ in the productive view, which is productive, since it is the expansion of a productive term $\mathsf{F}\bar{\beta}$. It therefore preserves the invariant 4; it also preserves the invariant 3. Rule Decompose merges two productive views of the same multiequation and preserves 3.

Invariant 5 *The set of toplevel symbols of any productive view ν in C forms a continuous subtree of the forest $\gtrdot$. The root of this subtree, the type constructor of smallest rank.*

This invariant obviously holds for a singleton view $\langle \tau \rangle$ as introduced by rule Dispatch-P. It is maintained by Decompose, since we join two continuous subtrees that share a node. Rule Expand extends the tree (towards the trunk) with an immediate successor, which preserves continuity.

Rules Clean and Clash obviously preserve all five invariants.

Lemma 1 *The unification rules preserve the semantics of constraints.*

Proof: Rule Merge and Trim, Decompose, Class are standard.

Rule Dispatch leaves the raw multiequation unchanged. Rule Clean relies on the invariant 2.

Finally, Expand is a combination of the addition of the immediate expansion of a term along $\triangleright$, which preserves equality followed by Trim and Decompose, but it can also be seen directly:

- Assume that ϕ is a solution of $\langle \mathsf{F}\bar{\alpha} \doteq \nu \rangle \doteq m$ and $\mathsf{F}\beta \triangleright \mathsf{G}\bar{\tau}$.
- Let ϕ' be $\phi[\beta_i \leftarrow \alpha_i]$ (notice that β_i's are all distinct). Then $\phi'(\mathsf{F}\bar{\beta}) = \phi'(\mathsf{G}\bar{\tau})$, we also have $\phi'(\mathsf{F}\bar{\alpha}) = \phi(\mathsf{F}\bar{\beta})$ and therefore $\phi'(\mathsf{F}\bar{\alpha}) = \phi'(\mathsf{G}\bar{\tau})$.
- Therefore ϕ' is a solution of $\langle \mathsf{G}\bar{\tau} \doteq \mathsf{F}\bar{\alpha} \doteq \nu \rangle \doteq m \wedge_{i \in \downarrow \mathsf{F}} \alpha_i \doteq \beta_i$, that is, ϕ is solution of $\exists \bar{\beta} \cdot \langle \mathsf{G}\bar{\tau} \doteq \mathsf{F}\bar{\alpha} \doteq \nu \rangle \doteq m \wedge_{i \in \downarrow \mathsf{F}} \alpha_i \doteq \beta_i$.

The converse is obvious.

---

6 Unification strategy

The unification rules are sound and can be applied in any order. We describe a strategy to restrict the application of the rules that is both efficient and sufficient to obtain normal forms. We do so in two steps: we first defined a collection

$$
\begin{aligned}
(\textsc{Dispatch-All}) &\triangleq (\textsc{Dispatch-P}) \cup (\textsc{Dispatch-N}) \\
(\textsc{Trim-Dispatch}) &\triangleq (\textsc{Trim-P})^{\infty} \; ; (\textsc{Trim})^{\infty} \; ; (\textsc{Dispatch-All})^{\infty} \\
(\textsc{Decompose-Clean}) &\triangleq (\textsc{Clash}) \cup \big((\textsc{Decompose}) \; ; (\textsc{Clean})^{\infty}\big) \\
(\textsc{Merge-Decompose}) &\triangleq (\textsc{Merge}) \; ; (\textsc{Stutter})^{\infty} \; ; (\textsc{Clean})^{\infty} \; ; \\
&\qquad\qquad (\textsc{Decompose-Clean})^{\infty} \\
(\textsc{Expand-Decompose}) &\triangleq (\textsc{Expand})^{\dagger} \; ; (\textsc{Trim-Dispatch})^{\infty} \; ; \\
&\qquad\qquad (\textsc{Decompose-Clean})^{\infty}
\end{aligned}
$$

Figure 1: Derived rules

of derived rules (Figure 1) that will be used instead of basic transformation rules. Derived rules ensure further invariants, following a transformation rule by further rules that restore some invariants broken by the first application. We then define an algorithm that applies derived rule in a deterministic order.

Derived rules are defined in Figure 1.

Rule Dispatch-All is a helper rule to factor all forms of dispatch. Rule Trim-Dispatch implicitly uses ExAnd to extrude existential quantifiers that Trim introduced.

Rule Merge-Decompose applies to a conjunction of multiequations, but each of the other rules applies to a single multiequation.

Rule Expand-Decompose uses Rule $(\textsc{Expand})^{\dagger}$ is a restriction of Rule Expand that requires:

- the expanded term to be the representative of the view of highest rank
- the rest of the multiequation m to contain at least another view ν'

6.1 Invariants of composed rules

Invariant 6 (Strategy) *The strategic rules maintain the additional invariants:*

- *All terms are small.*
- *All constructed terms are in a productive or degenerate view.*
- *The top constructors of two productive views are disjoint.*
- *All terms in a productive view have different toplevel symbols.*

The term of a productive view with the toplevel symbol of smallest rank is called the representative.

These invariants are preserved by each of the composed rules—often in an obvious way. These invariants can be ensured on the input constraint as follows:

- Input multiequations have only single terms views (one could also drop views that do not satisfy the invariants).
- Apply (Trim-Dispatch) to make all terms small and place then in the appropriate views.

6.2 Algorithm

The algorithm Solve is defined as the following composition of rules:

$$(\textsc{Solve}) \triangleq \\ \big((\textsc{Merge-Decompose})^{\infty} \,;\, (\textsc{Expand-Decompose}) \,;\, (\textsc{Merge-Decompose})^{\infty}\big)^{\infty}$$

In words, the algorithm makes the system merged and decomposed as far as possible; when no more merge, decomposition, nor clash can be performed, it does a single expansion; it iterates until no more (restricted) expansion is possible, i.e., all multiequation have at most one productive view. It is important to perform a single expansion because these may allocate new terms and may be expansive. It is also important to select the terms of lowest rank in the view of highest rank for expansion, as this is the one most likely to allow decomposition after merging. Otherwise, expansion could uselessly expand all type abbreviations away, which kills the purpose of this algorithm.

The algorithm Solve always terminate, with solved forms.

6.3 Termination

We show the termination of (Solve).

The weight $|\mathsf{F}|$ of a type constructor F of rank n is the polynomial X^n. The weight $|\tau|$ of a term τ is defined as

$$|\alpha| = 0 \qquad\qquad |\mathsf{F}\bar{\tau}| = |\mathsf{F}| + 2 * (\Sigma_{i\in I}|\tau_i|)$$

In particular, the weight of a variable is null and the weight of a term is higher that the weight of its subtrees plus the weight of the toplevel type constructor.

The weight of a productive view is the weight of its representative, i.e., the term of the lowest rank.

The weight of a multiequation is the sum of the weights of its productive views plus twice the size of terms that are not in a view.

The weight of the constraint is the sum of the weight of its multiequations. The size of the constraint is $sY + mZ$ where s is its weight and m is the number of multiequations.

Rewriting rules reduce the size of C. More precisely, assuming that the initial constrain satisfies the invariant 6, i.e., in particular, where all terms are small:

- Rules Stutter and Clean are auxiliary rules that cannot be applied forever, and do not increase the size of C, but may not decrease it either.

- Rule Dispatch-All does not increase the weight of the multiequation as it moves a term whose size count into a degenerate view where its size does not count or a productive view where it counts less. (Currently, it does not increase the number of multiequations, but it could.) Hence, this rule can only be used after another rule that strictly decreases the weight of C.

- Rule Trim-P and Trim both reduce the weight of the multiequation they apply to, but also introduces new multiequations, which counts less.

- Rule Trim-Dispatch does not increase the size of the constraint. (It does not necessarily decreases it, as all terms may already be small.) The rule alone always terminates, as it does not create new type constructors while it decreases the weight of terms and the number of terms not in a view. Hence, this rule may be safely used after a rule that strictly decreases the weight of multiequations.

- Rule Decompose-Clean decreases the size of the constraint. Rule Clash does, indeed! Rule Decompose followed by Clean decreases the weight of the system. Indeed, it replaces two views, with representatives $\mathsf{F}_1\bar{\alpha_1}$ and $\mathsf{F}_2\bar{\alpha_2}$, by a single view, with representative either $\mathsf{F}_1\bar{\alpha_1}$ or $\mathsf{F}_2\bar{\alpha_2}$. This is regardless of whether F is one of F_1 or F_2. Simultaneously, the rule may introduce new multiequations composed of just variables, i.e., of null weight, hence the whole weight of the constraint decreases and so does its size.

- Rule Merge-Decompose decreases the size of the constraint: mainly, Rule Merge reduces the number of multiequations without increasing anything else. Then, after cleaning steps, which leaves the size unchanged, it may decrease the size of the constraint (Rule Decompose-Clean) or leave it unchanged if this rule does not apply.

- Rule Expand-Decompose decreases the size of the constraint. Mainly, Rule Expand decreases the weight of the system by replacing in a view a representative of by another representative of lower rank, hence of smaller size, while increasing the number of multiequations. Notice that the original term is left in the view whose length increases, but it it will never be used as the representative anymore. Rule (Trim-Dispatch)$^{\infty}$, does not increase the size of the constraint. Rule Decompose-Clean if applicable decreases the size of the constraint.

- Rule Solve uses three rules, Merge-Decompose, Expand-Decompose, and Merge-Decompose, each of which decreases the size of the constraint.

6.4 Efficiency of the strategy

By construction, the algorithm never performs an expansion (of a constructive type abbreviation) when a decomposition is possible.

Decompositions are in general preferable, because they do not allocate new terms, but on the opposite merges existing terms. Furthermore, when a multiequation is expanded, we also choose the view with highest rank hoping to be able to merge it (and decompose it) with a view of a lower rank. This choice ensures that the rank of the multiequation in its canonical form will be maximal, i.e., as high as the rank in any other canonical form.

However, this does not ensure a minimal number of expansions. The choice of the multiequation to expand is arbitrary among those that are candidate for expansion.

Expanding a view ν_1 in a multiequation m_1 may perhaps be avoided by another expansion that will induce merging of m_1 with another multiequation m_2 that will contain a view ν_2 that will be decomposable with ν_1. For example, consider

$$\alpha_1 \doteq \langle \nu_1 \rangle \doteq \langle \nu_2 \doteq \tau \rangle \wedge \alpha_2 \doteq \langle \nu_2 \rangle \doteq \nu_1 \doteq \tau \wedge C$$

when the multiequation α_1 cannot be decomposed, hence is a candidate for expansion. Instead, an expansion in C might lead to a decomposition which in turn will require merging α_1 and α_2, leading to a constraint of the form

$$\alpha_1 \doteq \alpha_2 \doteq \langle \nu_1 \doteq \tau \rangle \doteq \langle {\nu_1}' \rangle \langle \nu_1 \rangle \doteq {\nu_1}' \doteq \tau \wedge C'$$

Now, we first decompose the two views containing τ leading to

$$\alpha_1 \doteq \alpha_2 \doteq \langle \nu_1 \doteq \tau \doteq \nu_2 \rangle \doteq \langle \nu_1 \rangle \doteq \nu_2 \wedge C''$$

which in turn allows to decompose the two remaining views ending up with a multiequation with a single view—without doing any further expansion.

In summary, our strategy delays expansions until at least one is necessary, and chooses an expansion that is promising, but does not ensure optimality in the number of expansions performed.

6.5 Solved forms

The algorithm terminates with constrains in solved form, i.e., constraints C of the form $\exists \beta \cdot \wedge_{i \in I} \nu_i$ where

- Any variable occurring in C (free or bound) belongs to a unique multiequation.

- Every multiequation ν_i has at most one productive view, hence is of the form $\bar{\alpha_i} \doteq \langle \nu_i \rangle \doteq [{\nu_i}']$ where ν_i and ${\nu_i}'$ may be empty and otherwise satisfy the invariants 6 (in addition to invariants 1–4 of §5.6).

Indeed,

- This follows from the fact that Merge does not apply.
- Since Expand does not apply, two views of the same multiequations cannot be such that one contains a type abbreviation.

  Since Decompose does not apply, two views of the same multiequations cannot have the same head symbols.

  Since Clash does not apply, then there cannot be two such views in a multi-equations, i.e., there is be at most one productive view per multi-equation.

7 Deconstructive type abbreviations

We now extend the algorithm to allow deconstructive type abbreviations. We first consider a simple case in §7.1, relax it in §7.2, before we tackle the general case in §**??**

7.1 A purely deconstructive type abbreviation

The simplest for of deconstructive type abbreviation is $\mathsf{F}(\mathsf{G}\bar{\beta}) = \alpha_i$.

Notice that G may itself be a type abbreviation. Hence, we must assume that G is at least decomposable in position i, otherwise α_i is undetermined and the type definition does not make sense.

Such an abbreviation is non-productive: since $\mathsf{F}(\mathsf{G}\bar{\beta})$ expands to the type variable β_i, it should behave as β_i; in particular $\mathsf{F}(G\bar{\beta}) \doteq m$ should require the merge of this multiequation with with the one β_i belongs to. A simple way to ensure this is to expand $\mathsf{F}(\mathsf{G}\bar{\beta})$ eagerly: when a term $\mathsf{F}\alpha$ appears in a multiequation m, we add β_i to m with an additional constraint $\alpha \doteq \mathsf{G}\bar{\beta}$. We perform this when we dispatch $\mathsf{F}\alpha$ into the degenerate view:

Dispatch-Exp-D

$$\frac{\mathsf{F}(\mathsf{G}\bar{\beta}) \triangleright \beta_i}{\mathsf{F}\alpha \doteq [\nu] \doteq m \implies \exists\bar{\beta} \cdot [\mathsf{F}\alpha \doteq \nu] \doteq \beta_i \doteq m \wedge \alpha = \mathsf{G}\bar{\beta}}$$

Rule Dispatch-Exp-D is a mix of Dispatch-N and Expand in the sense that, when we meet $\mathsf{F}\alpha$, we need to allocate some structure $\alpha \doteq \mathsf{G}\bar{\beta}$ to force α to have the requested shape and extract α_i from it.

Lemma 2 *Rule* Dispatch-C *is an equivalence.*

Proof: Assume that ϕ is a solution of $\mathsf{F}\alpha \doteq [\nu] \doteq m$. The definition of the type abbreviation of $\mathsf{F}(\mathsf{G}\bar{\beta}) \triangleright \beta_i$ implies that $\phi\alpha$ is of the form $\mathsf{F}(\mathsf{G}\tau)$ for some $\bar{\tau}$. Let ϕ' be $\phi[\beta_i \leftarrow \tau_i]$. By definition $\mathsf{F}(\mathsf{G}\tau)$ is equal to τ_i. Therefore, ϕ' is a solution of $[\mathsf{F}\alpha \doteq \nu] \doteq \alpha_i \doteq m \wedge \alpha = \mathsf{G}\bar{\beta}$. Hence, ϕ is a solution of $\exists\bar{\beta} \cdot [\mathsf{F}\alpha \doteq \nu] \doteq \beta_i \doteq m \wedge \beta = \mathsf{G}\bar{\beta}$. The inverse direction is obvious.

Rule Dispatch-Exp-D preserves invariant 2. To preserve invariant 1, we need to define $\dot{\forall}$ for type abbreviations such as F, replacing i by a sequence of projections to access the variable β_i from $\mathsf{F}(\mathsf{G}\bar{\beta})$ in the definition of the type abbreviation[2]. In our example, $\dot{\forall}\,\mathsf{F}$ is $1 \cdot i$. In fact, we may also allow two parameters β_i and β_j to be actually the same variable. Thus, $\dot{\forall}\,\mathsf{F}$ is rather a set of paths than a single path. Hence, we should write $1 \cdot i \in \dot{\forall}\,\mathsf{F}$ or $\pi \in \dot{\forall}\,F$ when π stands for a path.

Adding rule Dispatch-Exp-D to the definition of Dispatch-All, we preserve properties of the unification algorithm. In particular, rule Dispatch-All does not increase the weight of the constraint, although it may now increase the number of multiequations.

Therefore, the (Solve) still terminates, with the same solved forms (but more general degenerate views).

Solved forms Notice that a constraint such as $\alpha \doteq \mathsf{F}\beta \wedge \beta \doteq \mathsf{F}_0$ looks in solved form—but it is not! Indeed, $\mathsf{F}\beta$ carries the implicit constraint that β must be of the form $\mathsf{G}\alpha_0$ which implies that the constraint $\mathsf{G}\alpha_0 \doteq \mathsf{F}_0$ should also hold—but it is unsolvable (equivalent to false).

Fortunately, solved forms satisfy the invariant 6 which requires non variable terms to be either in degenerate views or productive views. Here, F must therefore have been moved to the degenerate view, which enforces its expansion and will produce a clash between F_0 and G.

Decomposable paths If G is unary, then F is decomposable and $\downarrow\!\mathsf{F} = \{1\}$. Otherwise, F is not decomposable—unless all arguments of G but one are phantom. For instance, assume F is unary and G is binary and decomposable in both directions. Then $\mathsf{F}(\mathsf{G}(\tau_0, \tau_1)) = \mathsf{F}(\mathsf{G}(\tau_0, \tau_2))$ but $\mathsf{G}(\tau_0, \tau_1) \neq \mathsf{G}(\tau_0, \tau_2)$.

Thus, there are now positions that are neither decomposable nor phantom. This is not a problem however, since degenerate type systems will never appear in a productive view and need not be decomposed.

It may seem annoying that a position may be neither phantom nor decomposable. However, we should not consider positions but paths, which are a generalization of positions.

We redefine $\downarrow$ to return a list of paths (starting from the root) instead of a list of positions. That is, it is not immediate parameters that are decomposable or phantom but paths rooted at F. Continuing with our example, we would have $\downarrow\!\mathsf{F}$ is $\{1 \cdot 1\}$, i.e., the path $1 \cdot 1$ is decomposable, while the occurrence $1 \cdot 2$ is phantom.

Generalization We can easily generalize the previous example with type abbreviations of the form $\mathsf{F}(\mathsf{G}_i \bar{\alpha}_i)^{i \in I} \triangleright \alpha_{ij}$.

[2]In the general case, $\mathsf{G}\bar{\beta}$ could even be replaced by a term σ where β_i also occurs deeper inside σ.

7.2 Mixed type abbreviations

Deconstructive type abbreviation pattern match on their argument to extract some subterm. They need not return just one subterm, but instead may also construct a new term by combining several subterms.

Consider for example $\mathsf{F}(\mathsf{G}(\alpha, \beta)) = \mathsf{G}(\beta, \alpha)$ where G is a primitive type constructor (or more generally a productive one, decomposable on both directions). This abbreviation is both deconstructive and constructive. How should we treat it? Here are a few hints:

- F is decomposable: if $\mathsf{F}\tau = \mathsf{F}\tau'$ then τ and τ' must be of the form $\mathsf{G}(\tau_1, \tau_2)$ and $\mathsf{G}(\tau_1', \tau_2')$ and $\mathsf{F}\tau = \mathsf{G}(\tau_2, \tau_1)$ and $\mathsf{F}\tau' = \mathsf{G}(\tau_2', \tau_1')$.
- Hence, it looks like we could treat F as productive.
- Still, we must be aware that $\alpha \doteq \mathsf{F}\beta \wedge \beta \doteq \mathsf{F}_0$, which looks in canonical form is actually not. Indeed, there is an implicit constraint that β must be of the form $\mathsf{G}(\tau_1, \tau_2)$ and $\alpha \doteq \tau_1$
- Hence, as with the previous example, we must eagerly expand F—but still put it in a productive views.

This leads to the following dispatching rule:

Dispatch-Exp-P
$$\frac{\mathsf{F}(\mathsf{G}\bar{\beta}) \triangleright \sigma}{\mathsf{F}\alpha \doteq m \implies \exists \bar{\beta} \cdot \langle \sigma \doteq \mathsf{F}\alpha \rangle \doteq m \wedge \alpha \doteq \mathsf{G}(\bar{\beta})}$$

This rule decreases the size of the constraint (it decreases its weight while increasing the number of multiequations, which counts less).

There (solve) still terminates, with the same normal forms. By construction, we know that whenever $\mathsf{F}\alpha$ appears in a productive view of a constraint C, then $C \vdash \exists \bar{\beta} \cdot \alpha \doteq \mathsf{G}\beta$ holds, since such constrains are injected into the system when F is initially placed into the productive view.

7.3 The general case

Let us now tackle the general case of an abbreviation $\mathsf{F}(\bar{\tau}) \triangleright \sigma_{\mathsf{F}}$.

We need a few restrictions for this definition to be well formed:

- F should be new, i.e., not occur in $\bar{\tau}$ nor in σ_{F}.
- All variables should be bound, i.e., $\mathsf{ftv}(\sigma_{\mathsf{F}}) \subseteq \mathsf{ftv}^{\nabla}(\bar{\tau})$
- $\nabla\sigma_{\mathsf{F}}$ should exist, i.e., deconstructive type constructors that occur in σ_{F} should be applied to subterms of expected shapes.

There are several cases to consider:

- σ_{F} is a variable β; then, F is degenerate, as in §7.1

$$\textsc{Dispatch-Exp-D}\quad \frac{\mathsf{F}(\bar\tau) \triangleright \sigma_{\mathsf{F}} \qquad i \cdot \pi \in \dot\forall\, \mathsf{F} \qquad \tau_i/\pi = \beta \qquad \bar\tau \not\subseteq \mathcal{V}}{\mathsf{F}\bar\alpha \doteq [\nu] \doteq m \implies \exists\, \mathsf{ftv}(\bar\tau) \cdot [\mathsf{F}\bar\alpha \doteq \nu] \doteq \beta \doteq m \wedge^{i \in I} \alpha_i \doteq \tau_i}$$

$$\textsc{Dispatch-Exp-P}\quad \frac{\mathsf{F}(\bar\tau) \triangleright \sigma_{\mathsf{F}} \qquad \dot\forall\, \mathsf{F} = \mathsf{H} \qquad \bar\tau \not\subseteq \mathcal{V}}{\mathsf{F}\bar\alpha \doteq m \implies \exists\, \mathsf{ftv}(\bar\tau) \cdot \langle \sigma_{\mathsf{F}} \doteq \mathsf{F}\bar\alpha \rangle \doteq m \wedge^{i \in I} \alpha_i \doteq \tau_i}$$

Figure 2: Derived rules for deconstructive abbreviations

- σ_{F} is of the form $\mathsf{G}\bar\tau$ and $\mathsf{G}\bar\tau \multimap \beta$. This is quite similar to the previous case.
- σ_{F} is of the form $\mathsf{G}\bar\tau$ and $\dot\forall\, \mathsf{G} = \mathsf{H}$. Then F is productive.
- σ_{F} is of the form $\mathsf{G}\bar\tau$ and G is itself a deconstructive type abbreviation. This is in fact, a particular case of previous ones: we just look at $\dot\forall\, \mathsf{G}$ which is the same as $\dot\forall\, \mathsf{F}$ and tell us whether the type abbreviation is degenerate or productive.

In all cases, we compute $\dot\forall\, \mathsf{F}$, directly by computing $\nabla\sigma_{\mathsf{F}}$ directly, or indirectly by looking at σ_{F}. Then

- if $\nabla\sigma_{\mathsf{F}}$ is a variable, then $\dot\forall\, \mathsf{F}$ is a path π (of the form $i \cdot \pi$) and F is degenerate;
- otherwise, $\dot\forall\, \mathsf{F}$ is a (primitive) type constructor H and F is productive

This leads to two new dispatching rules Dispatch-Exp-D and Dispatch-Exp-P defined on Figure 2. These rules are complementary and should not replace the rules Dispatch-P and Dispatch-D. Even though the Dispatch-Exp versions of the rules are semantically correct, the (grayed) side condition $\bar\tau \not\subseteq \mathcal{V}$ requires that at least one of the τ_i's is not a variable, so that the original rules do not apply.

XXX [Fix $\downarrow\mathsf{F}$ for rule Expand Dispatch-Exp-P]

Degenerate type abbreviations For degenerate terms, we can actually see Dispatch-D as Dispatch-Exp-D followed by some cleanup of useless variables. Indeed, when $\bar\tau$ are all distinct type variables $\bar\beta$, the difference is just the introduction of auxiliary existential variables $\bar\beta$ with multiequations $\wedge^{i \in I} \alpha_i \doteq \beta_i$, which can then be removed, using the equivalence

$$\exists \beta \cdot C \wedge \alpha \doteq \beta \Rrightarrow C \qquad \text{(when } \beta \,\#\, C)$$

Productive type abbreviations: eager *vs.* lazy expansion The main difference between deconstructive rules and non-deconstructive ones if for productive terms: Rule Dispatch-Exp-P performs an eager expansion while Dispatch-P does not perform expansion, which will them be done on demand, lazily. Hence, there is a discontinuity between the two rules.

We can in fact reduce this difference, by automatically splitting deconstructive type annotations as follows:

- Let $\bar{\beta}$ be $\mathsf{ftv}(\sigma_{\mathsf{F}})$;
- Define $\hat{\mathsf{F}}\bar{\beta} \triangleright \sigma_{\mathsf{F}}$ where $\bar{\beta}$;
- Redefine $\mathsf{F}\bar{\tau} \triangleright \hat{\mathsf{F}}\beta$.

With this encoding, the dispatch rule for F becomes:

$$\textsc{Dispatch-Exp-P'}\quad \frac{\mathsf{F}(\bar{\tau}) \triangleright \hat{\mathsf{F}}\bar{\beta} \triangleright \sigma_{\mathsf{F}} \qquad \dot{\forall}\,\mathsf{F} = \dot{\forall}\,\hat{\mathsf{F}} = \mathsf{H}}{\mathsf{F}\bar{\alpha} \doteq m \implies \exists\, \mathsf{ftv}(\bar{\tau}) \cdot \langle \hat{\mathsf{F}}\bar{\beta} \doteq \mathsf{F}\bar{\alpha} \rangle \doteq m \wedge^{i \in I} \alpha_i \doteq \beta_i}$$

The construction of σ_{F} is now delayed until $\hat{\mathsf{F}}\bar{\beta}$ need to be expanded, as for non destructive type abbreviations.

This could be automated or used manually to delay the constructive part of deconstructive type abbreviations.

It can also be used to regain the continuity between the Dispatch-Exp-P and Dispatch-P, so that the latter can be seen as a particular case of the former (up to some details) when $\bar{\tau}$ is a tuple of disjoint variables: F becomes a pure alias for $\hat{\mathsf{F}}$ which could (intuitively) be removed. This would lead to the rule:

$$\textsc{Dispatch-P'}\quad \frac{\mathsf{F}\bar{\beta} \triangleright \sigma_{\mathsf{F}} \qquad \dot{\forall}\,\mathsf{F} = \mathsf{H}}{\mathsf{F}\bar{\alpha} \doteq m \implies \exists \bar{\beta} \cdot \langle \mathsf{F}\bar{\alpha} \rangle \doteq m \wedge^{i \in I} \alpha_i \doteq \beta_i}$$

where the conclusion is equivalent and could be simplified to $\langle \mathsf{F}\bar{\alpha} \rangle \doteq m$, which then coincides with the conclusion of Rule Dispatch-P.

Soundness Rules Dispatch-Exp-D and Dispatch-Exp-D preserve the semantics of constraints.

Termination Both rules decrease the size of the constraint. Notice that this is different from the dispatching of non-deconstructive type abbreviations, which did not decrease the size of constraint. The deconstructive version allocate new terms, but this is coming and compensated by the expansion of the type abbreviation constructor, which as for regular expansions counts much more than the terms it allocates. Therefore, after including these rules in Dispatch-All, Rule Trim-Dispatch will still terminate when used alone and will never increase the size of the constraint.

Therefore Rule Solve still terminates with the same solved forms.

7.4 Further Encodings

We have seen that the constructive part of deconstructive type abbreviations can be recovered by an helper type abbreviation.

Similarly, we could use helper type abbreviations for the deconstructive parts. Consider $\mathsf{F}(\bar{\tau}) \triangleright \sigma_{\mathsf{F}}$. Let $\bar{\alpha}$ be $\mathsf{ftv}(V\tau)$ and $(\mathsf{F}_i(\bar{\alpha}) \triangleright \tau)^{i\in I}$ a sequence of type abbreviations. We could then redefine $\mathsf{F}(\mathsf{F}_i\bar{\tau}) \triangleright \sigma_{\mathsf{F}}$ with just one level of deconstruction.

This allows to limit the deconstruction of the arguments to their topmost constructor during the dispatch and treat the rest of decomposition lazily—or avoid it completely if the F_i's are never mixed with some other type constructor.

Thanks to this encoding, we could also have limited the general case to type abbreviations of the form $\mathsf{F}(\mathsf{G}_i\bar{\alpha}_i)^{i\in I} \triangleright \sigma_{\mathsf{F}}$ and solve others by encoding.

Alternative encodings According to the above schema, the deconstructive type abbreviation $\mathsf{F}(\mathsf{G}(\mathsf{G}(\alpha_0, \alpha_2), \alpha_0)) \triangleright \alpha_0$ would be encoded as

$$\hat{\mathsf{F}}(\alpha_0, \alpha_1, \alpha_2) \triangleright \mathsf{G}(\mathsf{G}(\alpha_0, \alpha_1), \alpha_2) \qquad\qquad \mathsf{F}(\hat{\mathsf{F}}(\alpha_0, \alpha_1, \alpha_2)) \triangleright \alpha_0$$

where the deconstruction is done minimally delaying the allocation of the intermediate structure to $\hat{\mathsf{F}}$.

Another possible encoding would be:

$$\hat{\mathsf{F}}(\mathsf{G}(\alpha_0, \alpha_1)) \triangleright \alpha_0 \qquad\qquad \mathsf{F}(\mathsf{G}(\beta, \alpha_2)) \triangleright \hat{\mathsf{F}}\beta$$

to deconstruct the argument step by step.

The difference is interesting:

- The former uses an auxiliary constructive type abbreviation $\hat{\mathsf{F}}$ as an argument to the deconstructive one F to destruct "more at once". However, the expansion of $\hat{F}$ will be performed lazily, only when $\hat{\mathsf{F}}(\alpha_0, \alpha_1, \alpha_2)$ will be mixed with some other non variable type.

  So this may behave better when the expansion of $\hat{\mathsf{F}}$ is a large term.

  Still, when the encoding is automated, the abbreviations $\hat{\mathsf{F}}$ in the output result will have to be expanded away.

- The latter uses an auxiliary type abbreviation $\hat{\mathsf{F}}$ which as themselves Deconstructive as the result of the abbreviation F to deconstruct. The deconstruction is done "incrementally", but not lazily, i.e., since $\hat{\mathsf{F}}$ must itself be destructed when dispatched, which happens at the very beginning before the constraints is really solved, so there is no real gain.

8 Converting between types and canonical forms

8.1 Input types

Types constraints are usually introduced as a collection of simple equations of the form

$$(\sigma_1^j \doteq \sigma_2^j)^{j\in J}$$

Types may be large types, which can always be turned into small types with Trim, which can be applied before hand or on demand. Types, may also contain $\mu((\alpha_i = \tau_i)^{i \in I})\,\tau$ types, which can can be eliminated using the following rule:

Trim-Rec
$$\mu(\alpha_i = \tau_i)\,\tau = m \implies \exists {\alpha_i}^{i \in I} \cdot \tau = m \wedge^{i \in I} (\alpha_i = \tau_i)$$

(If we count the $\mu(\alpha)\,\tau$ form as a constructor in the weight of term, then Trim-Rec reduces the size of the system.)

8.2 Output types

A principal solution of a constraint C is a substitution ϕ of domain $\mathsf{ftv}(C)$ that maps type variables to syntactic types such that solutions of C are exactly those of the form $\phi' \circ \phi$ when ϕ' range over ground substitutions—up to expansion of type abbreviations.

When C in canonical form $\exists \bar{\beta} \cdot \wedge^{i \in 1..n} m_i$, a principal solution for C can be computed inductively as follows:

- Take ϕ_0 be the identity.
- Then, for i in $1..n$:
  - let $\bar{\alpha_i}$ be $\nu_i \cap \mathcal{V}$;
  - if m_i contains a productive view ν_i (which is then unique as m is canonical), let $\hat{\tau_i}$ be any element of ν_i and let $\hat{\phi_i}$ be $(\bar{\alpha_i} \mapsto \mu(\hat{\alpha_i})\,\phi_{i-1}\hat{\tau_i})$, where $\mu(\bar{\beta})\,\sigma$ stands for $\mu(\hat{\beta})\,(\sigma[\bar{\beta} \leftarrow \hat{\beta}])$ where β is an arbitrary element of $\bar{\beta}$;
  - otherwise, let $\hat{\alpha_i}$ be any element of $\bar{\alpha_i}$ and ϕ_i be $(\bar{\alpha_i} \mapsto \hat{\alpha_i})$;
  - let ϕ_i be $\hat{\phi_i} \circ \phi_{i-1}$.
- Return ϕ_n restricted outside of $\bar{\beta}$.

Lemma 3 *The solution ϕ is a principal solution.*

Notice that types $\hat{\tau_i}$ may be chosen arbitrarily in the productive view for each productive multiequation m_i: all choices are semantically equivalent.

Still, some choices may be less arbitrary than others:

(repr) Pick the representative of ν_i for $\hat{\tau_i}$. This is the most canonical choice: the representative is unique; its is the least unfolded presentation that sufficed to make all terms of the view agreed during constraint resolutions.

(all) Return all terms of the view: there head symbol form a tree for $\gtrdot$: starting from the leave, the next term may be obtain by a single expansion at the root.

(leaves) Return all the leaves of the view, when ordered by $\gtrdot$, e.g., terms with toplevel symbol of minimal ranks. (This would mean extending the syntax of terms to allow printing terms with synonyms, which the current syntax does not allow.

The interest of the (repr) strategy is to be canonical. However, it tends to loose type abbreviations while the (leaves) strategy will always preserve them. The (leaves) strategy may return multiple views, which then forces a choice again, unless there is a single leave.

Degenerate type abbreviations Notice that all of these strategies ignore degenerate type abbreviations, which will them never be used in output types. One reason is that the expansion of a degenerate term is smaller that the term itself. Another reason is that showing a degenerate term does not in general prevent from also showing a productive term when the multi-equation is productive.

8.3 Removing auxiliary type abbreviations

We have shown some encodings that use auxiliary type abbreviations. These should be eliminated in output types.

Recursive type abbreviations Recursive type abbreviations uses non-recursive helper type abbreviations $\mathsf{F}_i(\bar{\alpha}) \triangleright \mu((\beta_j = \hat{\mathsf{F}}_j(\bar{\alpha}, \bar{\beta}))^{j \in J})\, \beta_i)$, but introduced in such a way that we can always read back $\mathsf{F}_i(\bar{\tau})$ from $\hat{\mathsf{F}}_i(\bar{\tau}, \bar{\sigma})$ at anytime (an invariant that we preserve during unification). This amounts to performing a contraction. We can indeed, perform all such contractions on canonical forms and eliminate all such helper type abbreviations before outputting terms.

Alternatively, helper type abbreviations can be retain in principal solutions and only eliminated on the fly when show printing or converting terms.

Other auxiliary type abbreviations We have also proposed type abbreviations for the productive part of deconstructive type abbreviations. To eliminated those, we need to expand them. This seems to be removing the purpose of introducing them in the first place, but this is not quite the case: some of these might have been introduced to solved intermediate constraints on existential type variables and will not appear in the output solution, while we only need to expand those appearing in the output types.

Scoped type annotations Scoped type annotations are similar, they must be removed in the output, an[] d we can do so by forcing their expansion in the constraint at anytime. In particular, we can deal with scoped type annotations during resolution of the constraint if needed.

8.4 Example

Here is an example that loops in OCaml (version 4.09)

```
type 'a f = 'b constraint 'a = 'b option
let _ = fun (x : ('a f) option) (y : 'a) -> (x = y)
```

That is, we have an abbreviation:

$$\mathsf{F}(\mathsf{G}\alpha) \triangleright \alpha$$

where G, which stands for the option type, is a primitive type. Typechecking the program produces the constraint C equal to $\alpha \doteq \mathsf{H}(\mathsf{F}\alpha)$.

This will be reduced as follows:

$$\begin{array}{rcl}
 & & C \\
\textsc{Split} & \Rrightarrow & \exists\beta \cdot \alpha \doteq \mathsf{H}\beta \wedge \beta \doteq \mathsf{F}\alpha \\
\textsc{Dispatch-P} & \Rrightarrow & \exists\beta \cdot \alpha \doteq \langle \mathsf{H}\beta \rangle \wedge \beta \doteq \mathsf{F}\alpha \\
\textsc{Dispatch-Exp-D} & \Rrightarrow & \exists\beta \cdot \alpha \doteq \langle \mathsf{H}\beta \rangle \wedge (\exists\beta_1 \cdot \beta \doteq [\mathsf{F}\alpha] \doteq \beta_1 \wedge \alpha \doteq \mathsf{H}\beta_1) \\
\textsc{Merge} & \Rrightarrow & \exists\beta\beta_1 \cdot \alpha \doteq \langle \mathsf{H}\beta \rangle \doteq \mathsf{H}\beta_1 \wedge \beta \doteq \beta_1 \doteq [\mathsf{F}\alpha] \\
\textsc{Dispatch-P} & \Rrightarrow & \exists\beta\beta_1 \cdot \alpha \doteq \langle \mathsf{H}\beta \rangle \doteq \langle \mathsf{H}\beta_1 \rangle \wedge \beta \doteq \beta_1 \doteq [\mathsf{F}\alpha] \\
\textsc{Decompose} & \Rrightarrow & \exists\beta\beta_1 \cdot \alpha \doteq \langle \mathsf{H}\beta \doteq \mathsf{H}\beta_1 \rangle \wedge \beta \doteq \beta_1 \doteq [\mathsf{F}\alpha] \wedge \beta \doteq \beta_1 \\
\textsc{Merge} & \Rrightarrow & \exists\beta\beta_1 \cdot \alpha \doteq \langle \mathsf{H}\beta \doteq \mathsf{H}\beta_1 \rangle \wedge \beta \doteq \beta_1 \doteq \beta_1 \doteq [\mathsf{F}\alpha] \\
\textsc{Clean-P} & \Rrightarrow & \exists\beta\beta_1 \cdot \alpha \doteq \langle \mathsf{H}\beta \rangle \wedge \beta \doteq \beta_1 \doteq \beta_1 \doteq [\mathsf{F}\alpha] \\
\textsc{Stutter} & \Rrightarrow & \exists\beta\beta_1 \cdot \alpha \doteq \langle \mathsf{H}\beta \rangle \wedge \beta \doteq \beta_1 \doteq [\mathsf{F}\alpha]
\end{array}$$

The solution ϕ, is defined as follows:

$$\begin{array}{l}
\phi_0 = id \\
\phi_1 = (\alpha \mapsto \mu(\alpha)\,\mathsf{H}\beta) \circ \phi_0 = (\alpha \mapsto \mathsf{H}\beta) \\
\phi_2 = (\beta, \beta_1 \mapsto \beta) \circ \phi_1 = (\beta_1 \mapsto \beta, \alpha \mapsto \mathsf{H}\beta) \\
\phi = \phi_2 \setminus \{\beta, \beta_1\} = (\alpha \mapsto \mathsf{H}\beta)
\end{array}$$

That is ϕ is equal to $\alpha \mapsto \mathsf{H}\beta$. As noticed earlier, F does not appear in the output type since it is degenerate. We may still verify that ϕ solves the input constraint, indeed:

$$\phi\alpha = \mathsf{H}\beta =_E \mathsf{H}(\mathsf{F}(\mathsf{H}\beta)) = \phi(\mathsf{H}(\mathsf{F}\alpha))$$

8.5 Computing free variables

In section 4.2, we defined the set $\mathsf{ftv}(\tau)$ of syntactic free variables, computed as usual. We also defined the set $\mathsf{ftv}^{\nabla}(\tau)$ of semantic free variables as $\mathsf{ftv}(\triangledown\tau)$, which can still be computed on τ without fully expanding τ.

For this purpose, we only need to expand (the) deconstructive (part of) type abbreviations—if only to check that the constraints are satisfied. Then, we can

Acknowledgments

This core of this work, including the treatment constructive type abbreviations, is entirely based on the master internship of Morel [2019].

References

Carine Morel. Type inference and modular elaboration with constraints for ML extended with type abbreviations. Master's thesis, Université Paris Diderot-Paris 7, September 2019. URL `https://hal.inria.fr/hal-02361707`.