An insider's look at LF type reconstruction:

Everything you (n)ever wanted to know

Brigitte Pientka
McGill University, Montreal, Canada
bpientka@cs.mcgill.ca

Abstract

Although type reconstruction for dependently typed languages is common in practical systems, it is still ill-understood. Detailed descriptions of the issues around it are hard to find and formal descriptions together with correctness proofs are non-existing. In this paper, we discuss a one-pass type reconstruction for objects in the logical framework LF, describe formally the type reconstruction process using the framework of contextual modal types, and prove correctness of type reconstruction. Since type reconstruction will find the most general types and may leave free variables, we in addition describe abstraction which will return a closed object where all free variables are bound at the outside. We also implemented our algorithms as part of the Beluga language, and the performance of our type reconstruction algorithm is comparable to type reconstruction in existing systems such as the logical framework Twelf.

1 Introduction

The logical framework LF (Harper *et al.*, 1993) provides an elegant meta-language for specifying formal systems together together with the proofs about them. It combines a powerful type system based on dependent types with a simple, yet sophisticated technique, called higher-order abstract syntax, to encode local variables and hypothesis.

One of the most well known proof assistants based on the logical framework LF is the Twelf system (Pfenning & Schürmann, 1999). It is a widely used and highly successful system which is particularly suited for formalizing the meta-theory of programming languages (see for example (Crary, 2003; Lee *et al.*, 2007; Pientka, 2007)) and certified programming (Necula, 1997; Necula & Lee, 1998). Its theoretical foundation, the dependently typed lambda-calculus, is small and easily understood. Type checking for LF objects can be implemented in a straightforward way in a few hundred lines of code, and such an implementation is easily trusted.

Yet, Twelf is "the only industrial-strength proof assistant for developing meta-theory based on HOAS representations" (Aydemir *et al.*, 2008). We believe one major reason for this is that the technology which makes systems like Twelf practical and usable are ill-understood, and remains mysterious to the user and possible future implementors. The situation is not very different for other related dependently typed systems which also support recursion such as Coq (Bertot & Castéran, 2004) , Agda (Norell, 2007), Epigram (McBride & McKinna, 2004). All of these systems are supporting some form of type reconstruction

to infer omitted arguments. Yet there are hardly any concise formal description on how this is accomplished and what requirements the user-level syntax should satisfy. Formal foundations and correctness guarantees are even harder to find.

This is especially unfortunate, because we see recently a push towards incorporating logical framework technology into mainstream programming languages to support the tight integration of specifying program properties with proofs that these properties hold (see (Pientka, 2008; Poswolsky & Schürmann, 2008; Licata & Harper, 2007; Licata *et al.*, 2008)). However, the lack of foundations for how to build a practical dependently typed system remains a major obstacle in achieving this goal.

This paper tries to rectify the situation by providing an insider's guide and formal foundation to one of the most important practical issues: type reconstruction. Its goal is twofold: First, we provide a theoretical foundation based on contextual modal types (Nanevski *et al.*, 2008) for LF type reconstruction together with soundness and completeness proof. Based on the recipe for LF reconstruction described in (Pfenning, 1991), we explain concisely the conditions for type reconstruction of LF objects, present a bi-directional one-pass type reconstruction algorithm and prove its correctness. Our formal development mirrors our implementation of type reconstruction in OCaml as part of the Beluga language (Pientka, 2008; Pientka & Dunfield, 2008; Pientka & Dunfield, 2010). We have tested it on all the examples from the Twelf repository[1], and the performance of our implementation is competitive. Second, we hope our description will make LF technology and issues around type reconstruction in the presence of dependent type in general more accessible to future implementers and language designers. For example, it should be directly applicable to systems such as Dedukti (Boespflug, 2010), a proof checking environment based on the dependently typed lambda-calculus together with theories specified as rewrite rules.

The long-term goal is a precise understanding of programming language constructs involving dependent types and a sound mathematical basis for reasoning formally about these languages. We believe our work provides general insights into how to develop type reconstruction algorithms for dependently typed programming. More generally we believe it is a valuable step towards developing a clean foundation for dependently typed programming which supports pattern matching and recursion. Already we have used the described methodology also for type reconstruction of functional programs over dependently-typed higher-order data in Beluga. Finally, we believe it may help us to understand the necessary design choices and trade-offs for dependently-typed languages and will help to spread dependent types to main stream languages.

2 LF Type reconstruction 101

We first review the central ideas behind encodings in the logical framework LF and describe the general principle of type reconstruction.

The logical framework LF provides two key ingredients: 1) dependent types, which allow us to track statically powerful invariants and are necessary to adequately represent

[1] All examples which do not use definitions or constraint solvers. For some examples, we expanded the definitions by hand.

Source-level program

```
% Types for formulas, individuals
o: type .
i: type .

% Propositions
and: o → o  → o.
all:(i → o) → o.

% Natural deduction rules
nd: o → type .
andI:
     nd A → nd B
  → nd (and A B).

allI:
     (Π a:i. nd (A a))
   → nd (all A).

allE:
      nd (all A)
   → Π T:i. nd (A T)
```

Signature storing reconstructed program and number of reconstructed arguments

```
o    (0): type .
i    (0): type .

and  (0): o → o  → o.
all  (0):(i → o) → o.

nd   (0): o → type
andI (2): Π A:o. Π B:o.
          nd A → nd B
       → nd (and A B).

allI (1): Π A:i → o.
          (Π a:i. nd (A a))
       → nd (all (λx. A x).

allE (1): Π A:i → o.
          nd (all (λx. A x)
       → Π T:i. nd (A T).
```

Fig. 1. Encoding of Natural Deduction in the logical framework LF

proofs, and 2) support for higher-order abstract syntax, where binders in the object language are represented by binders in the meta-language. Both features present challenges. Support for higher-order abstract syntax for example means type reconstruction must rely on higher-order unification which is in general undecidable. We also must handle issues regarding η-contraction and expansion. The combination of both makes inferring the type of free variables and determining omitted arguments non-trivial. Our goal is not only to engineer a front-end for type reconstruction, but also to develop a theoretical foundation together with correctness guarantees.

We begin, by illustrating a typical example which showcases LF technology and highlights some of the issues which arise during reconstruction. As an example, we formalize a subset of the natural deduction calculus in LF and present it in Figure 1. We show the source code of the signature the user writes on the left. We briefly explain the idea behind this implementation of the natural deduction calculus. We first define a type `o` for formulae and a type `i` for individuals. Next, we define two propositions, conjunction and universal quantification. Conjunction is defined by the constructor `and` which takes in two propositions of type `o` and returns a proposition of type `o`. The universal quantifier is interesting since we need to model the scope of the variable it quantifies over. We achieve this by defining a constructor `all` which takes an LF abstraction of type `i → o` as an argument and returns an object of type `o`. Modelling binders in the object language (in our case formulae) by binders in the meta-language (in our case LF) is the essence of higher-order abstract syntax. One key advantage of this technique is that α-renaming is inherited from the meta-language and substitution in the object language can be modelled via β-reduction in LF. Finally, we consider the implementation of the natural deduction rules for conjunction introduction and universal quantifier introduction and elimination. We first define a type family `nd` which is indexed by propositions. Each inference rules is then represented as a constant of a specific type. For example, `andI` is the constant defining

the conjunction introduction rule. Its type says "Given a derivation of type `nd A` and a derivation of type `nd B`, we can create an object of type `nd (and A B)`." In the definition of the constant `allI`, we again exploit the power of the underlying logical framework to model parametric derivations. To establish a derivation for `nd (all A)`, we need to show that "for all parameters `a`, we have a derivation of type `nd (A a)`." This is modelled by the type `(Π x:i. nd (A a)) → nd (all A)`. Finally, the rule for `allE`. Given a derivation of type `nd (all A)` and any term `T` we can build a derivation of type `nd (A T)`. For an excellent introduction to LF encodings, we refer the reader to (Pfenning, 1997; Harper & Licata, 2007).

Our source language supports writing LF objects which are not necessarily in η-expanded (see for example declaration of `allI`). The user also may omit declaring free variables together with their types. The goal of type reconstruction is to take a source-level object and reconstruct a fully explicit LF object in β-normal and η-long form.

Type reconstruction - Basic: Inferring types of free variables

The right column in Figure 1 shows the reconstructed signature. Type reconstruction proceeds by processing one declaration at a time in the order they are specified. The main purpose of reconstruction for the given constant declarations is to infer the type of free variables occurring in its type or kind. This is a straightforward task for the constants `andI`, since all the types of free variables are simple types and are uniquely determined by the constructor `and`. In the declaration `allI` we infer the type of the free variable `A` as `i → o` since it is the argument to the constructor `all` and we η-expand A.

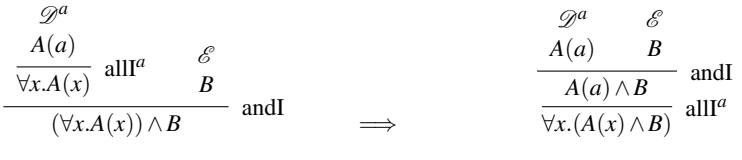
In general, we can infer the type of a free variable, if there is at least one occurrence which falls within the pattern fragment (Miller, 1991). This is the case when a free variable is applied to distinct bound variables. For example the free variable `A` in the term `λx.A x` is a pattern, but its second occurrence in `A T` is not a pattern, since `A` is applied to the free variable `T`. We will come back to this idea when we describe the inference algorithm more formally in Section 6.2.2.

In the reconstructed signature shown on the right in Figure 1, we store together with each constant the number of inferred arguments. We call inferred arguments also implicit arguments, since these are the arguments one may omit when we use the declared constant. For example, there are two inferred arguments in the type of the constant `andI`. Hence, when we build a derivation using the constant `andI` we may omit these two arguments. In all the examples listed here, the number of implicit arguments is equal to the number of free variables occurring in each declaration. This recipe was first described in (Pfenning, 1991) and works well in practice. We illustrate its basic principle and challenges next.

Type reconstruction - Intermediate: Inferring omitted arguments

To illustrate the idea behind reconstruction, consider proof transformations on natural deduction derivations such as the following:

```
trans: nd A → nd B → type .
andI-allI: trans   (andI E (allI D))           (allI (λa. (andI E (D a)))).
allE-allI: trans   (andI (allE T (allI D)) E)  (andI (D T) E).
```




```
trans        (andI (allI D) E)                (allI (a. (andI (D a) E)))
```

Fig. 2. Proof Transformations

The constant `andI-allI` specifies the transformation of a formula $(\forall x.A(x)) \wedge B$ into $\forall x.(A(x) \wedge B)$. `andI-allI` states that a proof `(andI (allI D) E)` for the formula `(and ( all λa. (A a)) B)` can be translated into the proof `(allI (λa. (andI (D a) E)))` for the formula `(all λa. (and (A a) B))` (see Figure 2). When the user declares this relation, s/he may omit passing those arguments to a constant which have been inferred when the constant was originally declared. For example, the constant `andI` allows us to omit the first two arguments, and we only need to supply the proof `allI D` and the proof `E` but not the concrete instantiations for `A` and `B` which will be inferred.

Reconstruction will translate the source-level signature into a well-typed LF signature by inserting meta-variables for omitted arguments, inferring the types of free variables and η-expanding bound and free variables, if necessary. The general idea is easily explained looking at the constant `andI-allI`.

We first traverse the term (or type) and insert meta-variables for omitted arguments. By looking up the type of a given constant, we know how many arguments must have been omitted. For example, we know that from the kind of `trans` stored in the signature that `trans` takes in two additional arguments, one for `A` and one for `B`.When we encounter the constant `andI`, its type in the reconstructed signature tells us that two arguments have been omitted. We show the type of the constant `andI-allI` after this step where we mark omitted arguments with underscores.

```
andI-allI:
    trans _____________________ _____________________
      (andI _________________ __ (allI _________ (λa. D a)) E)
      (allI ___________________ (λa. andI ______a __a (D a) E)).
```

Since omitted arguments may occur within the scope of `a`, the holes in the object `andI ______a __a (D a) E)` may depend on the bound variable `a`. To describe meta-variables with their bound variable dependencies more formally, we use contextual meta-variables (Nanevski *et al.*, 2008). For example, the holes which may depend on the bound variable `a` are described by the meta-variables `X1` and `X2` of contextual type `o[a':i]`. The meta-variable `X1` defines a hole of type `o` which can refer to the bound variable `a':i`. Meta-variables are closures consisting of the actual meta-variable together with a suspended substitution. We associate `X1` and `X2` with a substitution `[a]` which will rename the variable `a'` to `a`. In general we can omit writing the domain of the substitution, which simplifies the development. The meta-variable `X[.]` on the other hand denotes a closed object and is not allowed to refer to any bound variable. We write `[.]` for the empty substitution. Contextual meta-variables will be useful when we describe reconstruction and prove properties about

it. To summarize, the first step in type reconstruction is to insert meta-variables wherever an argument is omitted.

For now, let us look at the result of type reconstruction. Unification will try to find the most general instantiation for the holes and the final result as shown below

```
andI-allI:
    trans (and (all (λx. A1[x])) A2[.]) (all (λx. and (A1[x]) A2[.]))
      (andI (all (λx. A1[x])) A2[.] (allI (λx. A1[x]) (λa. D a)) E)
      (allI (λx. and (A1[x])  A2[.]) (λa. andI A1[a] A2[.] (D a) E)).
```

contains the meta-variables `A1`, `A2` describing the most general instantiations for the holes. We observe that the holes in the object `(andI _____a __a (D a) E))` are filled with `A1[a]` and `A2[.]` respectively. Although the second hole allowed its instantiation to depend on the variable `a`, unification eliminated this bound variable dependency. Since the variable `a` does not occur in the derivation described by `E`, its type cannot depend on it. Higher-order unification will properly weed out spurious bound variable dependencies.

Finally, we abstract over the meta-variables and free variables and explicitly bind them by creating a Π-prefix. Meta-variables of contextual type `o[x:i]` are lifted into ordinary variables of functional type `i → o`. The substitution associated with the meta-variable is turned into a series of applications. So for example, the meta-variable `A1[x]` is translated into a Π-bound variable `A1` of type `i→o` which is applied to `x`. While cyclic dependencies between meta-variables and free variables are allowed during reconstruction, an issue we will address in the next section, abstraction only succeeds if there is a linear order for meta-variables and free variables. The fully reconstructed type for the constants `andI-allI` is shown next:

```
andI-allI: Π A1:i → o. Π A2:o. Π D: Π x:i. nd (A1 x). Π E:nd A2.
           trans (and (all (λx. A1 x)) A2) (all (λx. and (A1 x) A2))
               (andI (all (λx. A1 x)) A2 (allI (λx. A1 x) (λa. D a)) E)
               (allI (λx. and (A1 x) A2) (λa. andI (A1 a) A2 (D a) E)).
```

Type reconstruction - Advanced: Circular dependencies

While the general idea behind type reconstruction is easily accessible, there are several subtleties in practice. We draw attention to one such issue in this section.

Let us consider the type reconstruction for `allE-allI`. We show first the type of `allE-allI` where we insert underscores for all the omitted arguments following the same principle explained in the previous section.

```
allE-allI: trans _____________ _____________
               (andI _____ _ (allE _____________ T (allI _____________ D)) E
               (andI _____ _ (D T) E).
```

Using the typing rules and higher-order unification, we will infer instantiations for these omitted arguments. However, these instantiations may need to refer to the free variable `T`. This is also obvious when we inspect the expected, final result for type reconstruction below.

```
allE-allI: Π A:i → o. Π B:o. Π T:i. Π D:Πa:i.nd (A a) Π E:nd B.
           trans (and (A T) B) (and (A T) B)
               (andI (A T) B (allE (λx. A x) T (allI (λx. A x) D)) E)
               (andI (A T) B (D T) E).
```

For example, the first reconstructed argument passed to `trans` in the definition of `allE-allI` is `(and (A T) B` where `T` was a free variable occurring in the user-specified object.

This however means that meta-variables characterizing holes may be instantiated with objects containing free variables; however, the type of free variables themselves is unknown and may contain meta-variables. Hence, there may be a circular dependency between meta-variables and free variables. There seems no easy way to avoid these circularities.

Fortunately, J. Reed (2009) pointed out that allowing circularities within meta-variables and free variables during unification are not problematic, i.e. the correctness of unification does not depend on it. However, for type reconstruction to succeed, abstraction must find a non-circular ordering of all the meta-variables and free variables. The exact order of free variables and the inferred variables (i.e. checking whether there in fact exists one) can only be determined once the object has been fully reconstructed. This will be done by abstraction.

In summary, type reconstruction for LF will reconstruct the type of free variables, synthesize omitted arguments, and ensure the final result is in β-normal and η-long form.

3 Implicit LF

In this section, we characterize the implicit syntax for LF which is closely related to the surface language. We may think of implicit LF as the target of a parser which translates source-level programs into implicit LF objects.

3.1 Grammar

We begin by characterizing the implicit syntax which features free variables. As a convention, we will use upper-case letters for free variables, and lower-case letters for bound variables. We will write **a** and **c** for type and term constants respectively in bold to distinguish them from types, terms, kinds and spines. We also support unknowns written as _ in the term. These unknowns or holes may occur anywhere in the term, but of course type reconstruction may not be able to instantiate all holes. We chose to have holes only as normal objects. This is not strictly necessary, and one could easily allow holes as heads. But we did not find this choice to be crucial in practice. For simplicity we also do not support holes on the level of types. This increases the burden on the user slightly since s/he needs to specify at least a type skeleton when using a Π-declaration. However, it is worth mentioning that one can infer a type skeleton by adding a pre-processing layer which explicitly verifies that the source-level expression is "approximately well-typed"[2].

Finally, unlike the implementation of LF in the Twelf system, we enforce that the objects written in the source-language are in β-normal form while the reconstructed objects are in β-normal and η-long form. Type annotations for lambda-abstractions are unnecessary in our setting since we will employ a bi-directional type system where we will always check a lambda-abstraction against a given type. We keep the implicit syntax small and concise (see Figure 3).

[2] We introduce approximate typing on page 11.

Implicit Kinds	k	::=	$\mathsf{type} \mid \Pi x{:}a.k$
Implicit Atomic types	p	::=	$\mathbf{a} \cdot s$
Implicit Types	a,b	::=	$p \mid \Pi x{:}a.b$
Implicit Normal Terms	m,n	::=	$\lambda x.m \mid h \cdot s \mid \_$
Implicit Spines	s	::=	$\mathsf{nil} \mid m;s$
Head	h	::=	$x \mid \mathbf{c} \mid X$

Fig. 3. Implicit LF - Source level syntax

The implicit syntax enforces that terms do not contain any β-redices. However, the implicit syntax does not require that terms are also η-expanded. For example, we can write `nd (all A)` instead of `nd (all λa. A a)` in Fig. 1 for example. For convenience, we choose a spine representation (Cervesato & Pfenning, 2003). For example, the implicit object `(all λa. A x)` is turned by a parser into

$$\mathbf{all} \cdot (\lambda a.A \cdot ((a \cdot \mathsf{nil})\ ;\ \mathsf{nil})\ ;\ \mathsf{nil})$$

The spine representation is convenient, since it allows us to directly access the head of a normal term.

3.2 η-contraction for implicit terms

Support for η-expansion and η-contraction is convenient for the user and it is often done silently. In our setting, the user can choose whether s/he writes a term in its η-expanded form or not. However, sometimes it is important to consider a special case: a given term M is simply the η-expansion of a bound variable. This is for example crucial, if we want to check that a free variable X is indeed applied to distinct bound variables and hence we can infer its type. Hence, we define here the operation $\eta\mathsf{con}(m) = x$ which will verify that m is the η-expanded form of the term $x \cdot \mathsf{nil}$.

$$\eta\mathsf{con}(\lambda y_1 \ldots y_n.x \cdot (m_1;\ldots;m_n;\mathsf{nil})) \quad = \quad x \qquad \text{if for all } i\ \eta\mathsf{con}(m_i) = y_i$$

If $n = 0$, we have $\eta\mathsf{con}(x \cdot \mathsf{nil}) = x$. We note that η-contraction of implicit terms is not type-directed, because we typically will not know the type of m when we want to use η-contraction.

4 Explicit LF

In this section, we present explicit LF which is the target of type reconstruction. The goal of type reconstruction will be to transform an implicit LF object into an equivalent explicit LF object.

4.1 Grammar

Explicit LF (see Figure 4) will feature meta-variables $u[\sigma]$, which are used to describe omitted implicit arguments, and are essentially based on the contextual modal type theory described in (Nanevski *et al.*, 2008). In addition, we add free variables to explicit LF. Abstraction then eliminates meta-variables and free variables by explicitly quantifying over

Kinds	K	$::=$	$\mathsf{type} \mid \Pi x{:}A.K$
Atomic types	P	$::=$	$\mathbf{a} \cdot S$
Types	A,B	$::=$	$P \mid \Pi x{:}A.B$
Normal Terms	M,N	$::=$	$\lambda x.M \mid R$
Neutral Terms	R	$::=$	$H \cdot S \mid u[\sigma]$
Head	H	$::=$	$x \mid \mathbf{c} \mid X$
Spines	S	$::=$	$\mathsf{nil} \mid M;S$
Substitutions	σ	$::=$	$\cdot \mid \sigma, M \mid \sigma; x$
Contexts	Ψ	$::=$	$\cdot \mid \Psi, x{:}A$
Free variable contexts	Φ	$::=$	$\cdot \mid \Phi, X : A$
Meta-contexts	Υ	$::=$	$\cdot \mid \Upsilon, u{::}P[\Psi]$
Signature	Σ	$::=$	$\cdot \mid \Sigma, a : (K,i) \mid \Sigma c : (A,i)$

Fig. 4. Explicit LF with meta-variables - Target of type reconstruction

them. The result of abstraction is a pure LF object which does not contain free variables nor meta-variables. In this development we do not introduce a different grammar for explicit LF with meta-variables and free variables on the one hand and pure LF on the other.

Our grammar for explicit LF will enforce that terms are in β-normal form, i.e. terms do not contain any β-redices. Our typing rules will in addition guarantee that well-typed terms must be in η-long form.

We assume that there is a signature Σ where term and type constants are declared, and their corresponding types and kinds are given in pure LF, i.e. the types of constants are fully known when we use them. In addition, we store together with constants the number i of inferred arguments. This is possible since we process a given program one declaration at a time, and it is moved to the signature Σ once the type (or kind) of a declared constant has been reconstructed. We suppress the signature in the actual reconstruction and typing judgments since it is the same throughout. However, we keep in mind that all judgments have access to a well-formed signature.

In the implementation of type reconstruction of LF we treat meta-variables as references and Υ which describes the set of meta-variables essentially characterizes the state of memory. We assume all generated meta-variables are of atomic type. This can always be achieved by lowering (Dowek *et al.*, 1996). Υ describes an unordered set of meta-variables and the context Φ describes the unordered set of free variables. The final order of Φ and Υ and whether such an order in fact exists, can only be determined after reconstruction is complete, since only then the types of the free variables are known. Our typing rules will not impose an order, and we allow circularities following similar ideas as by Reed (2009).

Substitutions σ are built either from normal objects M or heads H. This is necessary since when we apply σ to a term $\lambda x.M$ we must extend σ. However, the bound variable x may not denote a normal object unless it is of atomic type. Hence, σ, x is ill-typed, since x is not in η-long form. Consequently, there are two different substitutions which denote the same substitution: one where we encounter $\sigma; x$ and the other where we encounter σ, M where M can be η-contracted to x. Consequently, comparing two substitutions for equality must take into account η-contraction. We postpone the definition of η-contraction of explicit terms here to later, but remark it is essentially identical to the one we gave for

implicit terms. However, we will come back to the issue when we deal with eta-expansion and -contractions. We write $\mathsf{id}(\Psi)$ to describe the identity substitution for the context Ψ.

4.2 Typing rules for explicit LF

Figure 5 summarizes the typing rules for dependently-typed LF objects. Our typing rules ensure that LF objects are in β-normal and η-long form. This is convenient because concentrating on normal forms together with a bi-directional type system allows us to eliminate typing annotations for λ-abstractions. This simplifies the overall development since we do not need to ensure that such typing annotations are valid. We employ the following typing judgments:

$\Upsilon;\Phi;\Psi$	$\vdash$	M	$\Leftarrow A$	Term M checks against type A
$\Upsilon;\Phi;\Psi$	$\vdash$	$S:A$	$\Leftarrow P$	Spine S checks against type A and has target type P
$\Upsilon;\Phi;\Psi$	$\vdash$	σ	$\Leftarrow \Psi'$	Substitution σ has domain Ψ' and range Ψ

All judgments are in checking mode. For example, in the judgment for spines, we check that the spine S has type A and target type P, i.e., S, A, and P are given.

We concentrate here on the typing rules for LF objects and substitutions. We assume that type constants **a** together with constants **c** have been declared in a signature. We will tacitly rename bound variables, and maintain that contexts declare no variable more than once. Note that substitutions σ are defined only on ordinary variables x, not on modal variables u. We also require the usual conditions on bound variables. For example, in the rule for λ-abstraction, the bound variable x must be new and cannot already occur in the context Ψ. This can always be achieved via α-renaming.

Deciding whether $[\sigma]^a_\Psi P'$ is equal to P is essentially syntactic equality but must take into account η-contraction when comparing two substitutions.

It is also worth stating when meta-variable contexts and free variable contexts are well-formed. The circularity between these two contexts becomes obvious:

$$\frac{\text{for all } u::P[\Psi] \in \Upsilon \quad \Upsilon;\Phi;\Psi \vdash P \Leftarrow \mathsf{type} \quad \Upsilon;\Phi \vdash \Psi\ \mathsf{ctx}}{\vdash_\Phi \Upsilon\ \mathsf{mctx}}$$

$$\frac{\text{for all } X{:}A \in \Phi \quad \Upsilon;\Phi;\cdot \vdash A \Leftarrow \mathsf{type}}{\Upsilon \vdash \Phi\ \mathsf{fctx}}$$

In the definition of meta-variable contexts we rely on a free variable context Φ. Free variables declared in Φ are essentially treated as constants in a signature.

4.3 Substitutions

Hereditary substitutions The typing rules for neutral terms rely on *hereditary substitutions* that preserve canonical forms (Watkins *et al.*, 2002; Nanevski *et al.*, 2008). Substitution is written as $[M/x]^a_A$. The idea is to define a primitive recursive functional that always returns a canonical object. In places where the ordinary substitution would construct a redex $(\lambda y.M)N$ we must continue, substituting N for y in M. Since this could again create a redex, we must continue and hereditarily substitute and eliminate potential redices. Hereditary substitution can be defined recursively, considering both the structure of the

Normal Terms

$$\frac{\Upsilon;\Phi;\Psi,x{:}A \vdash M \Leftarrow B}{\Upsilon;\Phi;\Psi \vdash \lambda x.M \Leftarrow \Pi x{:}A.B} \qquad \frac{\Upsilon(u) = P'[\Psi'] \quad \Upsilon;\Phi;\Psi \vdash \sigma \Leftarrow \Psi' \quad [\sigma]^a_{\Psi'}P' = P}{\Upsilon;\Phi;\Psi \vdash u[\sigma] \Leftarrow P}$$

$$\frac{\Sigma(\mathbf{c}) = (A, \_) \quad \Phi;\Psi \vdash S : A \Leftarrow P}{\Upsilon;\Phi;\Psi \vdash \mathbf{c} \cdot S \Leftarrow P} \qquad \frac{x{:}A \in \Psi \quad \Upsilon;\Phi;\Psi \vdash S : A \Leftarrow P}{\Upsilon;\Phi;\Psi \vdash x \cdot S \Leftarrow P}$$

$$\frac{X{:}A \in \Phi \quad \Upsilon;\Phi;\Psi \vdash S : A \Leftarrow P}{\Upsilon;\Phi;\Psi \vdash X \cdot S \Leftarrow P}$$

Spines

$$\frac{}{\Upsilon;\Phi;\Psi \vdash \mathsf{nil} : P \Leftarrow P} \qquad \frac{\Upsilon;\Phi;\Psi \vdash M \Leftarrow A \quad \Upsilon;\Phi;\Psi \vdash S : [M/x]^a_A B \Leftarrow P}{\Upsilon;\Phi;\Psi \vdash M;S : \Pi x{:}A.B \Leftarrow P}$$

Substitutions

$$\frac{}{\Upsilon;\Phi;\Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Upsilon;\Phi;\Psi \vdash \sigma \Leftarrow \Psi' \quad \Upsilon;\Phi;\Psi \vdash M \Leftarrow [\sigma]^a_{\Psi'}A}{\Upsilon;\Phi;\Psi \vdash \sigma, M \Leftarrow \Psi', x{:}A}$$

$$\frac{\Upsilon;\Phi;\Psi \vdash \sigma \Leftarrow \Phi \quad \Psi(x) = A' \quad A' = [\sigma]^a_{\Psi'}A}{\Upsilon;\Phi;\Psi \vdash \sigma; x \Leftarrow \Psi', x{:}A}$$

Fig. 5. Typing rules for LF objects

term to which the substitution is applied and the type of the object being substituted. We also indicate with the superscript a that the substitution is applied to a type. Similarly, the superscript n indicates the substitution is applied to a term, the superscript l indicates we apply the substitution to a spine S, the superscript s indicates it is applied to a substitution σ and the superscript c indicates it is applied to a context. To guarantee that applying a substitution terminates, it is simpler to stick to non-dependent type of the object being substituted.

We therefore first define type approximations α and an erasure operation $()$ that removes dependencies. Before applying any hereditary substitution $[M/x]^a_A(B)$ we first erase dependencies to obtain $\alpha = A$ and then carry out the hereditary substitution formally as $[M/x]^a_\alpha(B)$. A similar convention applies to the other forms of hereditary substitutions. Types relate to type approximations via an erasure operation $()^-$ which we overload to work on types. Type approximations are however not only important to ensure termination of substitution, but we will also rely on the approximate types when defining and reasoning about η-expansion and define the relationship between explicit terms and implicit terms.

$$\begin{array}{rrcl} \text{Type approximations} & \alpha, \beta & ::= & \mathbf{a} \mid \alpha \to \beta \\ & (\mathbf{a} \cdot S)^- & = & \mathbf{a} \\ & (\Pi x{:}A.B)^- & = & A^- \to B^- \end{array}$$

Let us now consider the definition of hereditary substitution for normal terms. The full definition (including the definition for spines) can be found in the appendix.

Normal terms

$$
\begin{array}{lll}
[M/x]^n_\alpha(\lambda y.N) = \lambda y.N' & & \text{where } N' = [M/x]^n_\alpha(N) \\
 & & \text{choosing } y \notin \mathsf{FV}(M) \text{ and } y \neq x \\
[M/x]^n_\alpha(u[\sigma]) & = u[\sigma'] & \text{where } \sigma' = [M/x]^s_\alpha(\sigma) \\
[M/x]^n_\alpha(\mathbf{c}\cdot S) & = \mathbf{c}\cdot S' & \text{where } S' = [M/x]^l_\alpha S \\
[M/x]^n_\alpha(X\cdot S) & = X\cdot S' & \text{where } S' = [M/x]^l_\alpha S \\
[M/x]^n_\alpha(x\cdot S) & = \mathsf{reduce}(M:\alpha, S') & \text{where } S' = [M/x]^l_\alpha S \\
[M/x]^n_\alpha(y\cdot S) & = y\cdot S' & \text{where } y \neq x \text{ and } S' = [M/x]^l_\alpha S
\end{array}
$$

In general, we simply apply the substitution to sub-terms observing capture-avoiding conditions. The important case is when we substitute into a neutral term $x\cdot S$, since we may create a redex and simply replacing x by the term M is not meaningful. We hence define a function $\mathsf{reduce}(M:\alpha, S)$ which eliminates possible redices.

$$
\begin{array}{lcll}
\mathsf{reduce}(\lambda y.M:\alpha_1\to\alpha_2,(N;S)) & = & M'' & \text{where} [N/y]^n_{\alpha_1} M = M' \\
 & & & \text{and } \mathsf{reduce}(M':\alpha_2, S) = M'' \\
\mathsf{reduce}(R:\mathbf{a},\mathsf{nil}) & = & R & \\
\mathsf{reduce}(M:\alpha, S) & & \text{fails} & \text{otherwise}
\end{array}
$$

When we substitute M for x in the neutral term $x\cdot S$, we first compute the result of applying the substitution $[M/x]$ to the spine S which yields the spine S'. Second, we reduce any possible redices which are created using the given definition of reduce.

Substitution may fail to be defined only if substitutions into the subterms are undefined. The side conditions $y \notin \mathsf{FV}(M)$ and $y \neq x$ do not cause failure, because they can always be satisfied by appropriately renaming y. However, substitution may be undefined if we try for example to substitute a neutral term R for x in the term $x\cdot S$ where the spine S is non-empty. The substitution operation is well-founded since recursive appeals to the substitution operation take place on smaller terms with equal approximate type α, or the substitution operates on smaller types (see the case for $\mathsf{reduce}(\lambda y.M:\alpha_1\to\alpha_2,(N;S))$).

Lemma 4.1 (*Hereditary substitution lemma for LF objects*)
If $\Upsilon;\Phi;\Psi \vdash N \Leftarrow A$ and $\Upsilon;\Phi;\Psi, x:A, \Psi' \vdash M \Leftarrow B$
then $\Upsilon;\Phi;\Psi, [N/x]^c_A(\Psi') \vdash [N/x]^n_A(M) \Leftarrow [N/x]^a_A(B)$

Similar lemmas hold for all other judgments. For a full discussion on hereditary substitutions we refer the reader to (Nanevski *et al.*, 2008).

Contextual substitutions Meta-variables $u[\sigma]$ give rise to contextual substitutions, which are only slightly more difficult than ordinary substitutions. To understand contextual substitutions, we take a closer look at the closure $u[\sigma]$ which describes the meta-variable u together with a delayed substitution σ. We apply σ as soon as we know which term u should stand for. Moreover, we require that meta-variables have base type P and hence, we will only substitute neutral objects for meta-variables. This is not a restriction, since we can always lower the type of a meta-variable. An important consequence is that contextual substitution does not need to be hereditarily defined, since no redices can be created.

Finally because of α-conversion, the variables that are substituted at different occurrences of u may be different. As a result, substitution for a meta-variable must carry a context, written as $[\![\hat{\Psi}.R/u]\!]N$ and $[\![\hat{\Psi}.R/u]\!]\sigma$ where $\hat{\Psi}$ binds all free variables in R. This complication can be eliminated in an implementation of our calculus based on de Bruijn indexes.

Applying a single meta-substitutions $\hat{\Psi}.R/u$ to an object N, type A, substitution σ or context Φ is defined inductively in the usual manner. In each case, we apply it to its subexpressions. The only interesting case is when we encounter $N = u[\sigma]$. Here, we first compute some $\sigma' = [\![\hat{\Psi}.R/u]\!]\sigma$ and we replace u by $[\sigma']R$. Note that because all meta-variables are lowered, we can only replace meta-variables by neutral terms R. Hence, no redex is created by replacing u with $[\sigma']R$ and the termination of meta-substitution application only relies on the fact that applying σ' to R terminates (see previous section). Therefore, the termination for meta-substitutions is straightforward. Technically, we need to annotate contextual substitutions with the type of the meat-variable u and write $[\![\hat{\Psi}.R/u]\!]_{P[\Psi]}(N)$, since when we encounter $N = u[\sigma]$ and we compute $[\sigma']^n_\Psi R$ where we annotate the substitution $[\sigma']^n_\Psi$ with its domain. We usually omit this annotation on contextual substitutions subsequently to improve readability.

Theorem 4.2 (*Contextual Substitution Principles*)
If $\Delta_1; \Phi \vdash R \Leftarrow P$ and $\Delta_1, u::P[\Phi], \Delta_2; \Psi \vdash M \Leftarrow A$ then
$\Delta_1, [\![\hat{\Psi}.R/u]\!]\Delta_2; [\![\hat{\Psi}.R/u]\!]\Psi \vdash [\![\hat{\Psi}.R/u]\!]M \Leftarrow [\![\hat{\Psi}.R/u]\!]A$.

The simultaneous contextual substitution ρ maps meta-variables from Δ to a meta-variable context Δ'. As mentioned earlier, the meta-variable context Δ is not necessarily ordered which is reflected in its typing rule (see also Reed (2009)). This is different from the definition of contextual substitutions previously given for example in (Nanevski *et al.*, 2008) where we require that meta-variables are in a linear order. Finally, we must take into account the dependency between free variables and meta-variables.

$$\frac{}{\Delta' \vdash_\Phi \cdot \Leftarrow \cdot} \qquad \frac{\text{for all}(\hat{\Psi}.R/u) \in \rho \quad \Delta(u) = P[\Psi] \quad \Delta'; \Phi; [\![\rho]\!]\Psi \vdash R \Leftarrow [\![\rho]\!]P}{\Delta' \vdash_\Phi \rho \Leftarrow \Delta}$$

Applying circular contextual substitution will still terminate and it will produce a well-typed object. Intuitively a simultaneous meta-substitution can be viewed as a series of individual meta-substitutions. Hence it is still meaningful to reason about its application.

4.4 η-expansion

Since in our framework terms must be in η-long form, it is sometimes necessary to be able to η-expand a bound variable x. Type approximations suffice for the definition of η-expansion and simplify the theoretical properties about η-expanded terms. We define a function $\eta\mathsf{exp}_\alpha(x) = M$ which when given a bound variable x with type approximation α will produce its η-expanded term M.

We note that if $n = 0$ then we have $\eta\mathsf{exp}_{\mathbf{a}}(x) = x \cdot \mathsf{nil}$. For sake of completeness, we also define η-contraction for explicit terms in an identical manner to η-contraction for implicit terms, since it is necessary for comparing two substitutions for equality. It also highlights the duality of η-contraction and η-expansion.

$$\eta\mathsf{exp}_{\beta_1\to\ldots\to\beta_n\to\mathbf{a}}(x) \quad = \quad \lambda y_1\ldots\lambda y_n.x\cdot(\eta\mathsf{exp}_{\beta_1}(y_1);\ldots;\eta\mathsf{exp}_{\beta_n}(y_n);\mathsf{nil})$$

$$\eta\mathsf{con}(\lambda y_1\ldots\lambda y_n.x\cdot(M_1;\ldots;M_n;\mathsf{nil})) \quad = \quad x \qquad \text{if for all } i\ \eta\mathsf{con}(M_i)=y_i$$

Subsequently, we usually write $\eta\mathsf{exp}_A(x)$ instead of $\eta\mathsf{exp}_\alpha(x)$ where $\alpha = A^-$ but we keep in mind that we erase type dependencies before applying the definition of η-expansion. We state some simple properties about η-expansion next.

Lemma 4.3

1. If $\eta\mathsf{exp}_A(x) = M$ then $\mathsf{FV}(M) = \{x\}$.
2. $\eta\mathsf{con}(\eta\mathsf{exp}_A(x)) = x$.

Theorem 4.4

1. If $\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash N \Leftarrow B$ then $[\eta\mathsf{exp}_A(x)/x]^n_A N = N$.
2. If $\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash S : B \Leftarrow P$ then $[\eta\mathsf{exp}_A(x)/x]^l_A S = S$.
3. If $\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash \sigma \Leftarrow \Psi$ then $[\eta\mathsf{exp}_A(x)/x]^s_\alpha \sigma = \sigma$.
4. If $\Upsilon;\Phi;\Psi \vdash M \Leftarrow A$ then $[M/y]^n_A(\eta\mathsf{exp}_A(y)) = M$.

Proof
Mutual induction on N, S, and A (see appendix). □

Lemma 4.5
If $\Upsilon;\Phi;\Psi \vdash A \Leftarrow \mathsf{type}$, $\Psi(x) = A$ then $\Upsilon;\Phi;\Psi \vdash \eta\mathsf{exp}_A(x) \Leftarrow A$.

Proof
Structural nduction A using the previous theorem. □

5 Equivalence relation between implicit and explicit terms

The goal of reconstruction will be to translate an implicit term m into an equivalent explicit term M. Hence we characterize in this section when implicit terms are considered equivalent to explicit terms.

The equivalence relation between implicit terms and explicit terms will only compare terms which have the same approximate type. Defining equivalence in a type-directed manner is necessary, since the equivalence relation must take into account η-expansion.

The erasure operation $()^-$ on types A to obtain its type approximations α has been defined previously on page 11. We can extend the erasure operation to bound variable contexts in a straightforward way and will write ψ for the context approximating a bound variable context Ψ.

We define the relationship between the source-level object m and the reconstructed object M using the following judgment:

ψ	$\vdash$	m	$\approx$	M	$:\alpha$	term m is equivalent to term M at type approximation α
ψ	$\vdash$	s	$\approx$	S	$:\alpha \Leftarrow \mathbf{a}$	spine s is equivalent to spine S at type approximation α and target type approximation $\mathbf{a}$
ψ	$\vdash^i$	s	$\approx$	S	$:\alpha \Leftarrow \mathbf{a}$	spine s is equivalent to spine S at type approximation α and target type approximation $\mathbf{a}$ but the first i-th element of S are irrelevant

Equivalence on normal objects

$$\frac{\psi, x{:}\alpha \vdash m \approx M : \beta}{\psi \vdash \lambda x.m \approx \lambda x.M : \alpha \to \beta} \qquad \frac{\psi, x{:}\alpha \vdash h \cdot s@((x \cdot \mathsf{nil}); \mathsf{nil}) \approx M : \beta}{\psi \vdash h \cdot s \approx \lambda x.M : \alpha \to \beta}$$

$$\frac{\Sigma(\mathbf{c}) = (A, i) \quad \psi \vdash^{i} s \approx S : A^{-} \Leftarrow \mathbf{a}}{\psi \vdash \mathbf{c} \cdot s \approx \mathbf{c} \cdot S : \mathbf{a}} \qquad \frac{\Phi(X) = A \quad \psi \vdash s \approx S : A^{-} : \mathbf{a}}{\psi \vdash X \cdot s \approx X \cdot S : \mathbf{a}}$$

$$\frac{\psi(x) = \alpha \quad \psi \vdash s \approx S : \alpha \Leftarrow \mathbf{a}}{\psi \vdash x \cdot s \approx x \cdot S : \mathbf{a}} \qquad \frac{}{\psi \vdash \_ \approx R : \mathbf{a}}$$

Equivalence on spines with omitted arguments

$$\frac{\psi \vdash s \approx S : \alpha \Leftarrow \mathbf{a}}{\psi \vdash^{0} s \approx S : \alpha \Leftarrow \mathbf{a}} \qquad \frac{\psi \vdash^{i-1} s \approx S : \beta \Leftarrow \mathbf{a}}{\psi \vdash^{i} s \approx (M;S) : \alpha \to \beta \Leftarrow \mathbf{a}}$$

Equivalence on spines

$$\frac{\psi \vdash m \approx M : \alpha \quad \psi \vdash s \approx S : \beta \Leftarrow \mathbf{a}}{\psi \vdash (m;s) \approx (M;S) : \alpha \to \beta \Leftarrow \mathbf{a}} \qquad \frac{}{\psi \vdash \mathsf{nil} \approx \mathsf{nil} : \mathbf{a} \Leftarrow \mathbf{a}}$$

Fig. 6. Equivalence between implicit and explicit terms

We will omit here the context for meta-variables Υ and the context for free variables Φ in the definition of the rules, since they remain constant. Our equivalence relation will only compare well-typed terms. In particular, we assume that M has type A where $\alpha = A^{-}$.

The rules for defining the equivalence between implicit and explicit terms are given in Figure 6. The equivalence of terms at function type $\alpha \to \beta$ falls into two cases: a) both terms are lambda-abstractions. In this case, we check that the bodies are equivalent at type β in the extended context $\psi, x{:}\alpha$. b) the implicit term is not in η-expanded form, while the explicit term is. In this case, we incrementally η-expand m.

When comparing neutral terms, we only need to be careful, when we encounter a neutral term with a constant at the head. In this case, we look up the type of the constant together with the number i describing how many arguments can be omitted. We then skip over the first i arguments in the explicit spine and continue to compare the remaining explicit spine with the implicit spine.

From the equivalence between implicit and explicit terms we can directly derive rules which guarantee that an implicit term is approximately well-typed by keeping the implicit term M and the approximate types but dropping the explicit term M. We will use the following judgments to describe that an implicit term is approximately well-typed.

ψ	$\vdash$	m	$: \alpha$	term m has type approximation α
ψ	$\vdash$	s	$: \alpha \Leftarrow \mathbf{a}$	spine s has type approximation α and target type approximation $\mathbf{a}$
ψ	$\vdash^{i}$	s	$: \alpha \Leftarrow \mathbf{a}$	spine s has type approximation α and target type approximation $\mathbf{a}$ but the first i-th elements defined by the type α are irrelevant

We note that the approximate typing rules have access to a signature Σ which contains fully explicit types and kinds. In general, approximate typing rules are similar to the typing rules we find in the simply typed lambda-calculus with two exception: 1) we typically

do not η-expand terms which is handled in our typing rules. 2) we have the judgment $\psi \vdash^i s : \alpha \Leftarrow \mathbf{a}$ which allows us to skip over the first i arguments in the type approximation α.

Finally, we prove that if an implicit term m has approximate type A^-, and m can be η-contracted to a variable x where x has type A, then m is equivalent to the η-expanded form of x. This lemma is used in the soundness proof of reconstruction.

Lemma 5.1
If $\eta\mathsf{con}(m) = x$ and $\Psi^- \vdash m \Leftarrow A^-$ and $\Psi(x) = A$ then $\Psi^- \vdash m \approx \eta\mathsf{exp}_A(x) : A^-$.

Proof
Induction on A (see appendix). □

6 LF type reconstruction de-constructed

The reconstruction phase takes as input an implicit normal object m (resp. spine s, type a, kind k) and produces a dependent objects M (resp. S, A, K). The resulting object M is dependently typed. The main judgements are as follows:

$\Upsilon_1;\Phi_1;\Psi$	$\vdash m$	$\Leftarrow A$	$/_\rho\ (\Upsilon_2\ ;\ \Phi_2)M$	Reconstruct normal object
$\Upsilon_1;\Phi_1;\Psi$	$\vdash s : A$	$\Leftarrow P$	$/_\rho\ (\Upsilon_2\ ;\ \Phi_2)S$	Reconstruct spine

Reconstruction is type-directed. The given judgements define an algorithm where we separate the inputs from the outputs by $/$. The inputs in a given judgment are written on the left of $/$ and the outputs occur on the right. Given a source-level expression m which is stipulated to have type A in a context Υ_1 of meta-variables, a context Φ_1 of free variables, and a context Ψ of bound variables, we generate a well-typed object M together with the new context Υ_2 of meta-variables, the context Φ_2 of free variables, and a contextual substitution ρ. The contextual substitution ρ maps meta-variables from Υ_1 to the new meta-variable context Υ_2. We assume that all inputs Υ_1, Φ_1, Ψ and A are well-typed and m is well-formed. Hence, the following invariants must hold:

Assumptions:	$\cdot$	$\vdash_{\Phi_1}$	Υ_1	mctx
	Υ_1	$\vdash$	Φ_1	fctx
	$\Upsilon_1;\Phi_1$	$\vdash$	Ψ	ctx
	$\Upsilon_1;\Phi_1;\Psi$	$\vdash$	$A \Leftarrow$	type

Throughout we have that Φ_1 characterizes some of the free variables occurring in the object m, but not all of them yet, i.e. $\Phi_1 \subseteq \mathsf{FV}(m)$. In the beginning, Φ_1 will be empty and the idea is that we add to Φ_1 the free variable once we encounter it and are able to infer its type.

In this presentation, we do not assume that m has approximate type A^-, but choose to identify in the inference rules precisely where we rely on this information. This will highlight why we need the assumption that implicit objects are approximately well-typed.

All generated objects, types, meta-variable context Υ_2 and free variable context Φ_2 are well-typed, and we will maintain the following invariant:

Postconditions:	$\cdot$	$\vdash_{\Phi_2}$	Υ_2	mctx
	Υ_2	$\vdash$	Φ_2	fctx
	Υ_2	$\vdash_{\Phi_2}$	ρ	$\Leftarrow \Upsilon_1$
	$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi$	$\vdash$	M	$\Leftarrow [\![\rho]\!]A$
	$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi$	$\vdash$	$S : [\![\rho]\!]A$	$\Leftarrow [\![\rho]\!]P$

In our implementation, meta-variables are implemented via references and have a global status. Similarly, we are threading through the context of free variables and collect the free variable together with its dependent type as we traverse a given LF object. All free variables in Φ_1 will also occur in Φ_2. However, Φ_2 may contain more free variables and the type of free variables which were already present in Φ_1 may have been refined, i.e. $\Phi_2 \supseteq [\![\rho]\!]\Phi_1$. We give an overview of all the rules in Figure 7, but we will discuss them individually below.

6.1 Reconstruction of normal objects

6.1.1 Reconstruction of lambda-abstraction

Reconstruction of an abstraction is straightforward. We reconstruct recursively the body of the abstraction. We add the assumption $x : A$ to the context Ψ and continue to reconstruct m. This yields the reconstructed term M together with a new meta-variable context Υ_2 and free variable context Φ_2. By the stated invariants both contexts make sense independently of Ψ and we can simply preserve them in the conclusion and return $\lambda x.M$ as the final reconstructed object.

$$\frac{\Upsilon_1;\Phi_1;\Psi,x{:}A \vdash m \Leftarrow B \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)M}{\Upsilon_1;\Phi_1;\Psi \vdash \lambda x.m \Leftarrow \Pi x{:}A.B \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)\lambda x.M}$$

6.1.2 Reconstruction of atomic objects

Atomic objects are those which are not lambda-abstractions, i.e. they are of the form $h \cdot s$. As mentioned earlier, reconstruction is type-directed. If we encounter an atomic object which is not of atomic type, we need to first eta-expand it.

η-expandIng atomic objects type, we will η-expand it. η-expansion is done by incrementally. We write $s@(x \cdot \mathsf{nil});\mathsf{nil}$ for appending to the spine s the spine $(x \cdot \mathsf{nil});\mathsf{nil}$. This means we add the object $x \cdot \mathsf{nil}$ to the end of the spine s.

$$\frac{\Upsilon_1;\Phi_1;\Psi,x{:}A \vdash h \cdot (s@((x \cdot \mathsf{nil});\mathsf{nil})) \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)M}{\Upsilon_1;\Phi_1;\Psi \vdash h \cdot s \Leftarrow \Pi x{:}A.B \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)\lambda x.M}$$

Note that the term $x \cdot \mathsf{nil}$ is not necessarily in η-expanded form yet. This is fine, since we do not require that objects in the source-level syntax are η-expanded. The variables $x_1,\ldots,x_n$ will be expanded at a later point when it is necessary.

There are five possible atomic objects $h \cdot s$ depending on the head h. Since different actions are required depending on the head h, our spine representation is particularly useful.

Normal Terms

$$\frac{\Upsilon_1;\Phi_1;\Psi,x{:}A \vdash m \Leftarrow B \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)M}{\Upsilon_1;\Phi_1;\Psi \vdash \lambda x.m \Leftarrow \Pi x{:}A.B \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)\lambda x.M}$$

$$\frac{\Upsilon_1;\Phi_1;\Psi,x{:}A \vdash h\cdot(s@((x\cdot \mathsf{nil});\mathsf{nil})) \Leftarrow B \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)M}{\Upsilon_1;\Phi_1;\Psi \vdash h\cdot s \Leftarrow \Pi x{:}A.B \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)\lambda x.M}$$

$$\frac{\Sigma(\mathbf{c}) = (A,i) \quad \Upsilon_1;\Phi_1;\Psi \vdash^i s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S}{\Upsilon_1;\Phi_1;\Psi \vdash \mathbf{c}\cdot s \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)\mathbf{c}\cdot S}$$

$$\frac{x{:}A \in \Psi \quad \Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S}{\Upsilon_1;\Phi_1;\Psi \vdash x\cdot s \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)x\cdot S}$$

s is a pattern spine

$$\frac{X \notin \Phi_1 \qquad \Upsilon_1;\Phi_1;\Psi \vdash s \Leftarrow P \,/\, S : A \quad \Upsilon_1;\Phi_1;\Psi \vdash \mathsf{prune}\, A \Rightarrow (\Upsilon_2\,;\,\rho)}{\Upsilon_1;\Phi_1;\Psi \vdash X\cdot s \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,[\![\rho]\!]\Phi_1, X : [\![\rho]\!]A)X\cdot S}$$

$$\frac{\Phi_1(X) = A \quad \Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S}{\Upsilon_1;\Phi_1;\Psi \vdash X\cdot s \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)X\cdot S}$$

$$\frac{}{\Upsilon_1;\Phi_1;\Psi \vdash \_ \Leftarrow P \,/_{\mathsf{id}(\Upsilon_1)}\, (\Upsilon_1, u{::}P[\Psi]\,;\,\Phi_1)u[\mathsf{id}(\Psi)]}$$

Synthesize type A from pattern spine

$$\frac{}{\Upsilon_1;\Phi_1;\Psi \vdash \mathsf{nil} \Leftarrow P \,/\, \mathsf{nil} : P}$$

$$\frac{\begin{array}{lcll} \eta\mathsf{con}(m) & = & x & \Psi(x) = A \\ \eta\mathsf{exp}_A(x) & = & M & \Psi^- \vdash m \Leftarrow A^- \qquad \Upsilon_1;\Phi_1;\Psi \vdash s \Leftarrow P \,/\, S : B \quad [y/x]^a_A B = B' \end{array}}{\Upsilon_1;\Phi_1;\Psi \vdash m;s \Leftarrow P \,/\, M;S : \Pi y : A.B'}$$

Synthesize spine

$$\frac{\Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S}{\Upsilon_1;\Phi_1;\Psi \vdash^0 s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S}$$

$$\frac{\begin{array}{l} \Upsilon_1;\Phi;\Psi \vdash \mathsf{lower}\, A \Rightarrow (M, u{::}Q[\Psi']) \\ \Upsilon_1, u{::}Q[\Psi'];\Phi_1;\Psi \vdash^{i-1} s : [M/x]^a_A(B) \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S \qquad \rho = \rho', \hat{\Psi}'.R/u \end{array}}{\Upsilon_1;\Phi_1;\Psi \vdash^i s : \Pi x{:}A.B \Leftarrow P \,/_{\rho'}\, (\Upsilon_2\,;\,\Phi_2)[\![\rho]\!]M;S}$$

Check spine

$$\frac{\Upsilon_1;\Phi_1;\Psi \vdash \mathbf{a}\cdot S' \doteq \mathbf{a}\cdot S/(\rho\,,\,\Upsilon_2)}{\Upsilon_1;\Phi_1;\Psi \vdash \mathsf{nil} : \mathbf{a}\cdot S' \Leftarrow \mathbf{a}\cdot S \,/_\rho\, (\Upsilon_2\,;\,[\![\rho]\!]\Phi_1)\mathsf{nil}}$$

$$\frac{\begin{array}{l} \Upsilon_1;\Phi_1;\Psi \vdash m \Leftarrow A \,/_{\rho_1}\, (\Upsilon_2\,;\,\Phi_2)M \\ \Upsilon_2;\Phi_2;[\![\rho_1]\!]\Psi \vdash s : ([M/x]^a_A([\![\rho_1]\!]B)) \Leftarrow [\![\rho_1]\!]P \,/_{\rho_2}\, (\Upsilon_3\,;\,\Phi_3)S \qquad \rho = [\![\rho_2]\!]\rho_1 \end{array}}{\Upsilon_1;\Phi_1;\Psi \vdash m;s : \Pi x{:}A.B \Leftarrow P \,/_\rho\, (\Upsilon_3\,;\,\Phi_3)[\![\rho_2]\!]M;S}$$

Fig. 7. Reconstruction rules for LF

Reconstructing atomic objects with a bound variable as head To reconstruct the object $x \cdot s$, we will need to reconstruct the spine s by checking it against the type of x.

$$\frac{x{:}A \in \Psi \quad \Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S}{\Upsilon_1;\Phi_1;\Psi \vdash x \cdot s \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)x \cdot S}$$

Reconstructing atomic objects with constant as head When we reconstruct $\mathbf{c} \cdot s$, we first look up the constant $\mathbf{c}$ in the signature and obtain the type of $\mathbf{c}$ as well as the number i of implicit arguments (i.e. the number of arguments which may be omitted). We will now reconstruct the spine s by first synthesizing i new arguments and then continue to reconstruct the spine s (see "Synthesize Spine").

$$\frac{\Sigma(\mathbf{c}) = (A,i) \quad \Upsilon_1;\Phi_1;\Psi \vdash^i s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S}{\Upsilon_1;\Phi_1;\Psi \vdash \mathbf{c} \cdot s \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)\mathbf{c} \cdot S}$$

Reconstructing atomic objects with free variable as head – Case 1: The type of free variable is known When we encounter a neutral object $X \cdot s$ where the head is a free variable, there are two possible cases. If we already inferred the type A of the free variable X then we simply look it up in the free variable context Φ_1 and reconstruct the spine s by checking it against A.

$$\frac{\Phi_1(X) = A \quad \Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S}{\Upsilon_1;\Phi_1;\Psi \vdash X \cdot s \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)X \cdot S}$$

Reconstructing atomic objects with free variable as head – Case 2: The type of free variable is unknown If we do not yet have a type for X, we will try to infer it from the target type P. This is however only possible, if s is a pattern spine, i.e. a list of distinct bound variables. This is important, because pattern spines can be directly mapped to pattern substitutions, i.e. a substitution where we map distinct bound variables to distinct bound variables. Such substitutions have the key property that they are invertible and we will exploit this fact when inferring the type for X. More generally, given the target type P and a spine of distinct bound variables $x_1,\ldots,x_n$ which must have been declared in Ψ, we can infer a type A for X where $A = \Pi\Psi'.P'$ and there exists some renaming substitution $\sigma = x_1;\ldots;x_n$ with domain Ψ' and range Ψ s.t. $P' = [\sigma]^{-1}P$.

Because bound variables may occur in its η-expanded form, checking for a pattern spine must involve η-contraction. We will discuss this issue when we consider the rules for synthesizing a type A from pattern spine (see page 21).

Since free variables are thought to be quantified at the outside, the type A is not allowed to refer to any bound variables in Ψ. In other words, the type A must be closed. We therefore employ pruning to ensure that there exists a type A' s.t. $[\![\rho]\!]A = A'$ and where ρ is a pruning substitution which eliminates any undesirable bound variable dependencies. This is best illustrated by an example. Given the assumptions `p:i→o,a:i`, we have the object `D a` which is known to have type `nd X[p;a]`. The type we infer for the free variable `D` is `Πx:i.nd X[p;x]`. Because `D` is only applied to the variable `a`, but not the variable `p`, the bound variable `p` is left-over in the synthesized type for `D`. However, since the free variable `D` will be eventually bound at the outside and its type must be closed, it cannot contain any free variables. We hence prune the meta-variable `X` which has the

contextual type `o[q:i→o,x:i]` such that it can never depend on `q`. This is achieved by creating a meta-variable `Y` with type `o[x:i]` and replacing any occurrence of `X` with `Y[x]`. We call the contextual substitution which achieves this refinement of meta-variables a pruning substitution. Applying the pruning substitution to the synthesized type of the free variable will ensure it is closed. The final type synthesized for the free variable `D` after pruning, is `Πx:i.nd Y[x]`. In general, the pruning substitution ρ will restrict the meta-variables occurring in the synthesized type A in such a way that they are closed, i.e. $\Upsilon_2; [\![\rho]\!]\Phi_1; \cdot \vdash [\![\rho]\!]A \Leftarrow \mathsf{type}$. Pruning is an operation which is well-known in higher-order pattern unification algorithms (see for example (Pientka, 2003; Dowek *et al.*, 1995; Dowek *et al.*, 1996)).

$$\frac{\begin{array}{l}\text{s is a pattern spine}\\ X \notin \Phi_1 \qquad \Upsilon_1;\Phi_1;\Psi \vdash s \Leftarrow P \,/\, S : A \quad \Upsilon_1;\Phi_1;\Psi \vdash \mathsf{prune}\, A \Rightarrow (\Upsilon_2\,;\,\rho)\end{array}}{\Upsilon_1;\Phi_1;\Psi \vdash X\cdot s \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\, [\![\rho]\!]\Phi_1, X : [\![\rho]\!]A) X \cdot S}$$

Note, we omit here the case where we encounter a term $X \cdot s$ where s is a non-pattern spine. In this case, we cannot infer the type of X. Instead, we postpone reconstruction of $X \cdot s$ in the implementation, and reconsider this term later once we have encountered some term $X \cdot s'$ where s' is a pattern spine which enables us to infer a type for X. In practice one occurrence of the free variable X together with a pattern spine suffices to infer the type of it. We have not modelled this delaying of some parts of the terms explicitly here in these rules. This could be done using an extra argument, but not much is gained by formalizing this at this point. We formalize synthesizing a type from a pattern spine on page 21.

Reconstructing holes Finally, we allow holes written as _ in the source language. We restrict holes here to be of atomic type. This is however not strictly necessary, since one can eta-expand holes to allow more flexibility. If we encounter a hole of atomic type, we simply generate a meta-variable for it.

$$\frac{}{\Upsilon_1;\Phi_1;\Psi \vdash \_ \Leftarrow P \,/_{\mathsf{id}(\Upsilon_1)}\, (\Upsilon_1, u{::}P[\Psi]\,;\,\Phi_1) u[\mathsf{id}(\Psi)]}$$

Let us consider the reconstruction of spines next. There are three different ones: 1) Checking a spine against a given type A. 2) Synthesizing a type A from a spine s and its target type P. 3) Synthesizing a new spine and inferring omitted arguments given the type A.

6.2 Working with spines

Depending on the head of a term, spines are processed differently. In the simplest case, we make sure that the spine associated with a head is well-typed. We may however also use the spine s together with an overall type P of a term $X \cdot s$ to infer the type of the free variable X. Finally, maybe the most important case: given a spine s and a type A, we need to infer omitted arguments and produce a spine S which indeed checks against A.

6.2.1 Checking a spine

In the simplest case, we need to ensure the spine is well-typed. When we encounter an empty spine, we must ensure that the inferred type $a \cdot S'$ is equal to the expected type $a \cdot S$. This is achieved by unification. In this theoretical description we rely on decidable higher-order pattern unification to compute a substitution ρ under which both types are equal. In practice, unification could leave some constraints which will be stored globally and revisited periodically. Most importantly, we revisit these constraints once reconstruction is finished, and check whether these constraints can be solved.

$$\frac{\Upsilon_1;\Phi_1;\Psi \vdash a \cdot S' \doteq a \cdot S/(\rho\ ,\ \Upsilon_2)}{\Upsilon_1;\Phi_1;\Psi \vdash \mathsf{nil} : a \cdot S' \Leftarrow a \cdot S\ /_\rho\ (\Upsilon_2\ ;\ [\![\rho]\!]\Phi_1)\mathsf{nil}}$$

When we encounter a spine $m;s$ which is stipulated to have type $\Pi x{:}A.B$ then we first reconstruct m by checking it against A, and then continue to reconstruct the spine s by checking it against $[M/x]^a_A(B)$. Recall that only approximate types are necessary to guarantee termination of this substitution. We do not have to apply ρ_1 to A and annotate the substitution $[M/x]$ with $[\![\rho_1]\!]A$, since all dependencies in A will be erased before applying the substitution and $(A)^- = ([\![\rho]\!]A)^-$.

$$\frac{\begin{array}{l}\Upsilon_1;\Phi_1;\Psi \vdash m \Leftarrow A\ /_{\rho_1}\ (\Upsilon_2\ ;\ \Phi_2)M \\ \Upsilon_2;\Phi_2;[\![\rho_1]\!]\Psi \vdash s : ([M/x]^a_A([\![\rho_1]\!]B)) \Leftarrow [\![\rho_1]\!]P\ /_{\rho_2}\ (\Upsilon_3\ ;\ \Phi_3)S \qquad \rho = [\![\rho_2]\!]\rho_1\end{array}}{\Upsilon_1;\Phi_1;\Psi \vdash m;s : \Pi x{:}A.B \Leftarrow P\ /_\rho\ (\Upsilon_3\ ;\ \Phi_3)[\![\rho_2]\!]M;S}$$

6.2.2 Synthesizing a type A from a pattern spine s

When we encounter a free variable $X \cdot s$ whose type is still unknown, we synthesize its type from a type A and the spine s. We can only synthesize a type A from a spine s, if s is a pattern spine, i.e. a list of distinct bound variables. We employ the following judgment:

$$\Upsilon_1;\Phi_1;\Psi \quad \vdash s_{\text{pattern}} \quad \Leftarrow P \quad / S : A$$

Given a pattern spine s_{pattern} and the target type P of $X \cdot s$, we can synthesize the type A which X must have. In addition to the type A we map the pattern spine s_{pattern} in implicit LF to the corresponding pattern spine S in explicit LF. Because s_{pattern} is a pattern spine, we do not generate a new meta-variable context Υ_2, a new free variable context Φ_2 or a substitution ρ as we would in other judgements for reconstructing objects.

If s is empty, then we simply return P as the type.

$$\frac{}{\Upsilon_1;\Phi_1;\Psi \vdash \mathsf{nil} \Leftarrow P\ /\ \mathsf{nil} : P}$$

The more interesting case is when the spine has the form $m;s$. As mentioned earlier, given the target type P and a spine of distinct bound variables $x_1,\ldots,x_n$ which must have been declared in Ψ, we can infer a type A for X where $A = \Pi\Psi'.P'$ and there exists some renaming substitution $\sigma = x_1;\ldots;x_n$ with domain Ψ' and range Ψ s.t. $P' = [\sigma]^{-1}P$.

However, source-level terms may be η-expanded and we must take into consideration η-contraction. Unfortunately, η-contracting a source-level object m cannot be type-directed, since we do not yet know its type. As a consequence, we could have an ill-typed term m which could be reconstructed to some well-typed term M. For example, let $m = \lambda x.yx$

where y has some atomic type Q. η-contracting m yields some y. The corresponding η-expanded term of y will still be y, since it has atomic type. Our final result will be well-typed and preserve the invariants stated. This leads to the question whether one should accept such an ill-typed term from the source-level language. In this work, we take a conservative approach and assume that m must be approximately well-typed, i.e. if m can be η-contracted to some variable x and x has type A in the context Ψ, then m has type A^-. The guarantee that m is approximately well-typed will also make is easier to establish a relationship between m and M which is important when establishing correctness of type reconstruction.

$$\frac{\eta\mathsf{con}(m) = x \quad \Psi(x) = A \qquad \eta\mathsf{exp}_A(x) = M \quad \Psi^- \vdash m \Leftarrow A^- \qquad \Upsilon_1;\Phi_1;\Psi \vdash s \Leftarrow P \;/\; S : B \quad [y/x]^a_A B = B'}{\Upsilon_1;\Phi_1;\Psi \vdash m;s \Leftarrow P \;/\; M;S : \Pi y : A.B'}$$

In general, to synthesize the type of the spine $m;s$, we must find a B' s.t. $[M/y]B' = B$ where M is the reconstructed object of m. This is in general impossible. By lemma 4.4, we know that if M is the η-expanded form of a variable x at type A, then $[M/y]^a_A(B') = [x/y]B$. Hence the restriction to pattern spines will ensure that we only have to consider finding a B' s.t. $[x/y]^a_A B' = B$. To obtain B' we simply apply the inverse of $[x/y]^a_A$ to B, i.e. $B' = [x/y]^{-1}B = [y/x]^a_A B$.

The type we now infer for $m;s$ is composed of A, the type for m, and the type B which we infer for the spine s. Before we can however create a Π-type, we must possibly rename any occurrence of x with y.

The last question we must consider is what reconstructed spine should be returned. Since we require that the reconstructed spine is in η-expanded form we cannot return $x \cdot \mathsf{nil};S$ since x may not be normal if it is of function type. Hence, we first η-expand x to some object M and return $M;S$.

6.2.3 Inferring omitted arguments in a spine

Finally, the interesting case is how we in fact add missing arguments to a spine s. The judgment for inferring omitted arguments, takes as input the number i of arguments to be inferred as well as the type A which tells us what the type of each argument needs to be. In addition, we pass in the target type P.

$$\Upsilon_1;\Phi_1;\Psi \quad \vdash^i s : A \quad \Leftarrow P \quad /_\rho\ (\Upsilon_2\ ;\ \Phi_2)S$$

If i is zero, then no arguments need to by synthesized, and we simply reconstruct s by checking it against the type A and expected target type P.

$$\frac{\Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow P\ /_\rho\ (\Upsilon_2\ ;\ \Phi_2)S}{\Upsilon_1;\Phi_1;\Psi \vdash^0 s : A \Leftarrow P\ /_\rho\ (\Upsilon_2\ ;\ \Phi_2)S}$$

If i is not zero, we generate a new object M containing a meta-variable of atomic type by employing lowering. Intuitively, given a type $A = \Pi\Psi_q.Q$ in a context Ψ, we generate a meta-variable u of type $Q[\Psi,\Psi_q]$ and the term $M = \lambda\hat{\Psi}_q.u[\mathsf{id}(\Psi,\Psi_q)$. After substituting M into the expected type B of the remaining spine, we infer omitted arguments.

$$\frac{\Upsilon_1;\Phi;\Psi \vdash \mathsf{lower}\ A \Rightarrow (M, u{::}Q[\Psi'])\quad \Upsilon_1, u{::}Q[\Psi'];\Phi_1;\Psi \vdash^{i-1} s : [M/x]^a_A(B) \Leftarrow P\ /_\rho\ (\Upsilon_2\ ;\ \Phi_2)S \qquad \rho = \rho', \hat{\Psi}.R/u}{\Upsilon_1;\Phi_1;\Psi \vdash^{i} s : \Pi x{:}A.B \Leftarrow P\ /_{\rho'}\ (\Upsilon_2\ ;\ \Phi_2)[\![\rho]\!]M; S}$$

Since ρ is a substitution mapping meta-variables in $\Upsilon_1, u{::}Q[\Psi']$ to Υ_2, but in the conclusion we need to return a substitution which maps meta-variables from Υ_1 to Υ_2 we must take out the instantiation for $u{::}Q[\Psi']$ which should not be present and necessary anymore.

Lowering is also an operation well-known from the literature on higher-order pattern unification (see (Pientka, 2003; Dowek *et al.*, 1996)) and can be described as follows:

$$\frac{}{\Upsilon;\Phi;\Psi \vdash \mathsf{lower}\ P \Rightarrow (u[\mathsf{id}(\Psi)]\ ;\ u{::}P[\Psi])}$$

$$\frac{\Upsilon;\Phi;\Psi, x{:}A \vdash \mathsf{lower}\ B \Rightarrow (M\ ;\ u{::}P[\Psi'])}{\Upsilon;\Phi;\Psi \vdash \mathsf{lower}\ \Pi x{:}A.B \Rightarrow (\lambda x.M\ ;\ u{::}P[\Psi'])}$$

The judgment $\Upsilon;\Phi;\Psi \vdash \mathsf{lower}\ A \Rightarrow (M\ ;\ u{::}P[\Psi, \Psi'])$ can be read as follows: Given a context Ψ and a type A which is well-kinded in Ψ, we generate an η-expanded term M with a meta-variable u of atomic type $P[\Psi, \Psi']$ where $A = \Pi\Psi'.P$ and M has type A.

Lemma 6.1 (*Lowering*)
If $\Upsilon;\Phi;\Psi \vdash \mathsf{lower}\ A \Rightarrow (M\ ;\ u{::}P[\Psi'])$ then
$\Pi\Psi.A = \Pi\Psi'.P$ and $\Upsilon, u{::}P[\Psi']\ ;\ \Phi\ ;\ \Psi \vdash M \Leftarrow A$.

Proof
Structural induction on A. □

6.3 Soundness and completeness of LF reconstruction

Throughout reconstruction, we will maintain that the context for meta-variables and the free variable context are well-formed contexts according to our definition given on page 10. Reconstruction then guarantees that the LF object M (resp. S, A, K) is well-typed, a fact we will prove in this section. Reconstruction generates a contextual substitution ρ with domain Υ_2 and range Υ', and we have

$$\Upsilon';[\![\rho]\!]\Phi_2\ ,\ [\![\rho]\!]\Psi \vdash M \Leftarrow [\![\rho]\!]A.$$

In addition, m is equivalent to M. To ensure variable dependencies are properly taken into account, we will need to prune the type of existing meta-variables during reconstruction Because the type of meta-variables may be pruned and meta-variables are refined during unification, the relationship between the contexts Υ_1 and Φ_1 on the one hand and the context Υ_2 and Φ_2 on the other hand is not a simple subset relation. Instead, there exists a contextual substitution ρ s.t. $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ and $[\![\rho]\!]\Phi_1 \subseteq \Phi_2$.

In our implementation, variables bound in the context Ψ are represented using de Bruijn indices, while free variables are described using names. Since we do not yet know the types of the free variables their order cannot yet be determined and hence we cannot index them via de Bruijn indices. In the statement of soundness, we assume that all contexts, types etc. are well-formed. This is implicit in our statement. We are however explicit

about the well-formedness of the contexts we create. This will emphasize why we for example employ pruning and other restrictions during reconstruction. Finally, we can state and prove soundness of reconstruction.

Theorem 6.2 (*Soundness of reconstruction*)

1. If $\Upsilon_1;\Phi_1;\Psi \vdash m \Leftarrow A \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)M$ then
   $\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash M \Leftarrow [\![\rho]\!]A$ and $\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash m \approx M : A^-$
   $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ and $\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx and $[\![\rho]\!]\Phi_1 \subseteq \Phi_2$.
2. If $\Upsilon_1;\Phi_1;\Psi \vdash^i s : A \Leftarrow Q \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S$ then
   $\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash S : [\![\rho]\!]A \Leftarrow [\![\rho]\!]Q$ and $\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash^i s \approx S : A^- \Leftarrow Q^-$
   $[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ and $\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx.
3. If $\Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow Q \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S$ then
   $\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash S : [\![\rho]\!]A \Leftarrow [\![\rho]\!]Q$ and $\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash s \approx S : A^- \Leftarrow Q^-$
   $[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ and $\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx.
4. If $\Upsilon_1;\Phi_1;\Psi \vdash s \Leftarrow P \,/\, S : A$ then
   $\Upsilon_1;\Phi_1;\Psi \vdash S : A \Leftarrow P$ and $\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash s \approx S : A^- \Leftarrow P^-$ and
   $\Upsilon_1;\Phi_1;\Psi \vdash A \Leftarrow$ type.

Proof
Structural induction on the reconstruction judgment (see appendix). □

The presented type reconstruction algorithm is also complete in the following sense: if an implicit term m is equivalent to an explicit term M at type approximation α and M has type A s.t. $A^- = \alpha$, then type reconstruction will return some explicit term M' s.t. M is an instance of M'.

Our definition of equivalence between implicit and explicit terms relies on the fact that we already know the type of all the free variables. As a consequence, we never need to infer the type of a free variable in m during reconstruction.

To state the completeness theorem, we also need to reason about how the context of meta-variables evolves. In particular, we need to know that when we have a term M which has type A in the meta-variable context Υ, then there is a contextual substitution ρ_0 which allows us to move from a meta-variable context Υ_1 to the current meta-variable context Υ.

Theorem 6.3 (*Completeness of type reconstruction*)

1. If $\Psi^- \vdash m \approx M : A^-$ and $\Upsilon \vdash_{[\![\rho_0]\!]\Phi} \rho_0 \Leftarrow \Upsilon_1$
   and $\Upsilon;[\![\rho_0]\!]\Phi;[\![\rho_0]\!]\Psi \vdash M \Leftarrow [\![\rho_0]\!]A$ then
   $\Upsilon_1;\Phi;\Psi \vdash m \Leftarrow A \,/_\rho\, (\Upsilon_2\,;\,\Phi')M'$ and there exists a contextual substitution θ s.t.
   $[\![\theta]\!]\rho = \rho_0$ and $\Upsilon \vdash \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]M' = M$, $[\![\rho]\!]\Phi = \Phi'$.
2. If $\Psi^- \vdash s \approx S : A^- \Leftarrow P^-$ and $\Upsilon \vdash_{[\![\rho_0]\!]\Phi} \rho_0 \Leftarrow \Upsilon_1$
   and $\Upsilon;[\![\rho_0]\!]\Phi;[\![\rho_0]\!]\Psi \vdash S : [\![\rho_0]\!]A \Leftarrow [\![\rho_0]\!]P$ then
   $\Upsilon_1;\Phi;\Psi \vdash s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi')S'$ and there exists a contextual substitution θ s.t.
   $[\![\theta]\!]\rho = \rho_0$ and $\Upsilon \vdash \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]S' = S$, and $[\![\rho]\!]\Phi = \Phi'$.
3. If $\Psi^- \vdash^i s \approx S : A^- \Leftarrow P^-$ and $\Upsilon \vdash_{[\![\rho_0]\!]\Phi} \rho_0 \Leftarrow \Upsilon_1$
   and $\Upsilon;[\![\rho_0]\!]\Phi;[\![\rho_0]\!]\Psi \vdash S : [\![\rho_0]\!]A \Leftarrow [\![\rho_0]\!]P$ then
   $\Upsilon_1;\Phi;\Psi \vdash^i s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi')S'$ and there exists a contextual substitution θ s.t.
   $[\![\theta]\!]\rho = \rho_0$ and $\Upsilon \vdash \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]S' = S$, $[\![\rho]\!]\Phi = \Phi'$.

Proof
Structural induction on the first derivation (see appendix). □

7 Abstraction

A final step in the type reconstruction process is the abstraction over the free meta-variables and the free ordinary variables in a given declaration. Here we need to ensure that there exists some ordering for the free variables in Φ and the meta-variables in Υ. Given a LF object M (resp. A) with the free variables in Φ and the meta-variables in Υ, the correct order of Φ is determined by the order in which they occur in M (res. A). In our implementation, we traverse the object M (resp. A) and collect from left to right all meta-variables and free variables. This will determine their order. Intuitively, the variables occurring later in the term can only depend on the variables occurring earlier. Finally, all meta-variable occurrences $u[\sigma]$ where u has type $P[\Psi]$ are translated as follows: We first create a bound variable x of type $\Pi\Psi.P$, and translate the delayed substitution σ into a spine S whose elements are η-expanded. Any occurrence of $u[\sigma]$ is then replaced by $x \cdot S$. We first state the judgments:

$$\begin{array}{llllll} \Upsilon;\Phi & | & (\Psi_0)\Psi & \vdash & M & \Leftarrow B \quad / \ (\Psi_1)N \\ \Upsilon;\Phi & | & (\Psi_0)\Psi & \vdash & S:A & \Leftarrow P \quad / \ (\Psi_1)S' \\ \Upsilon;\Phi & | & (\Psi_0)\Psi & \vdash & \sigma & \Leftarrow \Psi' \quad / \ (\Psi_1)\sigma' \\ & & \Psi_0 & \vdash & \sigma & \Leftarrow \Psi_1 \quad / \quad S \end{array}$$

The final abstracted term N (resp. S', σ') must be a closed object, i.e. it does not refer to any meta-variables or free variables. The final result of abstraction then satisfies the following property:

$$(\cdot;\cdot) \mid \Psi_1, \Psi \vdash N \Leftarrow B$$

During abstraction, we take a slightly more liberal view of contexts Ψ_0 here since we pair the bound variable name up with either a free variable or a meta-variable. We may pair a free variable with a bound variable name even if we do not yet know the type of the variable. This is necessary to check for cycles.

$$\text{Context } \Psi \quad ::= \quad \cdot \mid \Psi, x{:}A \mid \Psi, X{\sim}x{:}A \mid \Psi, u{\sim}x{:}A \mid \Psi, X{\sim}x{:}\_ \mid \Psi, u{\sim}x{:}\_$$

From this more liberal context, we can obtain an ordinary LF context by dropping the association of a bound variable with a free variable or meta-variable respectively. Declarations $X{\sim}x{:}\_$ are dropped completely. We will do this silently when it is convenient.

When given a LF object which may still contain meta-variables and free variables from the context Υ and Φ respectively, we recursively traverse M (resp. the spine S or the substitution σ) and replace meta-variables and free variables with new bound variables. The context Ψ_0 keeps track of what meta-variable (and free variable) we already have replaced with a bound variable. This context which keeps track of these associations is threaded through. The result of abstracting over M is a new LF object N where all meta-variables and free variables have been replaced with bound variables listed in Ψ_1 and N is a pure LF object. We note that only the object we abstract over will contain meta-variables and free variables, but the context Ψ_0, Ψ and the types A, B, P are pure LF objects. The abstraction algorithm is presented in Figure 8. For better readability, we drop the meta-variable context Υ and the free variable context Φ both of which remain constant.

Abstraction for normal terms

$$\frac{(\Psi_0)\Psi, x{:}A \vdash M \Leftarrow B \;/\; (\Psi_1)N}{(\Psi_0)\Psi \vdash \lambda x.M \Leftarrow \Pi x{:}A.B \;/\; (\Psi_1)\lambda x.N} \qquad \frac{\Sigma(c) = (A, \_) \quad (\Psi_0)\Psi \vdash S : A \Leftarrow P \;/\; (\Psi_1)S'}{(\Psi_0)\Psi \vdash c \cdot S \Leftarrow P \;/\; (\Psi_1)c \cdot S'}$$

$$\frac{\Psi(x) = A \quad (\Psi_0)\Psi \vdash S : A \Leftarrow P \;/\; (\Psi_1)S'}{(\Psi_0)\Psi \vdash x \cdot S \Leftarrow P \;/\; (\Psi_1)x \cdot S'} \qquad \frac{X{\sim}x{:}A \in \Psi_0 \quad (\Psi_0)\Psi \vdash S : A \Leftarrow P \;/\; (\Psi_1)S'}{(\Psi_0)\Psi \vdash X \cdot S \Leftarrow P \;/\; (\Psi_1)x \cdot S'}$$

$$\frac{\begin{array}{l} X \notin \Psi_0 \qquad \Phi(X) = A \\ (\Psi_0, X{\sim}x{:}\_) \vdash A \Leftarrow \mathsf{type} \;/\; (\Psi_1)A' \qquad (\Psi_1, X{\sim}x{:}A')\Psi \vdash S : A' \Leftarrow P \;/\; (\Psi_2)S' \end{array}}{(\Psi_0)\Psi \vdash X \cdot S \Leftarrow P \;/\; (\Psi_2)x \cdot S'}$$

$$\frac{\begin{array}{llll} u \notin \Psi_0 \quad \Upsilon(u) = Q[\Psi_q] \quad [\sigma']^a_{\Psi'_q} Q' = P \quad x \text{ is new} & & & \\ (\Psi_0, u{\sim}x{:}\_) & \vdash \Psi_q & \mathsf{ctx} & / \; (\Psi_1)\Psi'_q \\ (\Psi_1) \qquad \Psi'_q & \vdash Q & \Leftarrow \mathsf{type} & / \; (\Psi_2)Q' \\ (\Psi_2, u{\sim}x{:}\Pi\Psi'_1.Q') \quad \Psi & \vdash \sigma & \Leftarrow \Psi'_q & / \; (\Psi_3)\sigma' \\ (\Psi_3) \qquad \Psi & \vdash \sigma' & \Leftarrow \Psi'_q & / \; S \end{array}}{(\Psi_0)\Psi \vdash u[\sigma] \Leftarrow P \;/\; (\Psi_3)x \cdot S}$$

$$\frac{u{\sim}x{:}\Pi\Psi_q.Q \in \Psi_0 \quad [\sigma']^a_{\Psi'_q} Q = P \quad (\Psi_0)\Psi \vdash \sigma \Leftarrow \Psi_q \;/\; (\Psi_1)\sigma' \quad \mathsf{subToSpine}_{\Psi_q}(\sigma') = S}{(\Psi_0)\Psi \vdash u[\sigma] \Leftarrow P \;/\; (\Psi_1)x \cdot S}$$

Abstraction for spines

$$\frac{}{(\Psi_0)\Psi \vdash \mathsf{nil} : P \Leftarrow P \;/\; (\Psi_0)\mathsf{nil}}$$

$$\frac{(\Psi_0)\Psi \vdash M \Leftarrow A \;/\; (\Psi_1)M' \quad (\Psi_1)\Psi \vdash S : [M'/x]^a_A(B) \Leftarrow P \;/\; (\Psi_2)S'}{(\Psi_0)\Psi \vdash M; S : \Pi x{:}A.B \Leftarrow P \;/\; (\Psi_2)M'; S'}$$

Abstraction of substitutions

$$\frac{}{(\Psi_0)\Psi \vdash \cdot \Leftarrow \cdot \;/\; (\Psi_0)\cdot} \qquad \frac{(\Psi_0)\Psi \vdash \sigma \Leftarrow \Psi' \;/\; (\Psi_1)\sigma' \quad (\Psi_1)\Psi \vdash M \Leftarrow [\sigma']^a_{\Psi'}(A) \;/\; (\Psi_2)M'}{(\Psi_0)\Psi \vdash \sigma, M \Leftarrow \Psi', x{:}A \;/\; (\Psi_2)(\sigma', M')}$$

$$\frac{(\Psi_0)\Psi \vdash \sigma \Leftarrow \Psi' \;/\; (\Psi_1)\sigma' \quad A = [\sigma']^a_{\Psi'}(A')(\Psi_0, \Psi)(x) = A}{(\Psi_0)\Psi \vdash \sigma; x \Leftarrow \Psi', x{:}A' \;/\; (\Psi_1)(\sigma'; x)}$$

Fig. 8. Abstraction for LF objects

In addition to the abstraction judgments over normal objects, spines and substitutions, we must be able to translate substitutions to spines. Recall that substitutions are associated with meta-variables $u[\sigma]$. When we encounter a meta-variable $u[\sigma]$ of type $P[\Psi]$, we generate a new bound variable of type $\Pi\Psi.P$ and translate the substitution to the corresponding spine. In fact, substitutions and spines are closely connected. We define a function $\mathsf{subToSpine}_\psi(\sigma)\ \ S = S'$ which expects a substitution σ together with its approximate typing context ψ and an accumulator S and will return the corresponding spine S'. Initially the accumulator argument S is the empty spine. We sometimes write $\mathsf{subToSpine}_\Psi(\sigma)\ S$, but keep in mind that type dependencies are erased from Ψ before computing the corresponding spine S.

$$
\begin{array}{llcll}
\mathsf{subToSpine}_{\Psi}\ \sigma & & = & \mathsf{toSpine}_{\psi}\ \sigma\ \mathsf{nil} & \text{where } \psi = \Psi^{-} \\
\mathsf{toSpine}_{\cdot}\ \cdot & S & = & S & \\
\mathsf{toSpine}_{\psi,x:\alpha}\ (\sigma;x) & S & = & \mathsf{toSpine}_{\psi}\ \sigma\ (\eta\mathsf{exp}_{\alpha}(x);S) & \\
\mathsf{toSpine}_{\psi,x:\alpha}\ (\sigma,M) & S & = & \mathsf{toSpine}_{\psi}\ \sigma\ (M;S) &
\end{array}
$$

Intuitively, the i-th argument in a substitution corresponds to the i-th argument in the spine. If the i-th argument was known to be a variable x then the i-th argument in the spine is the eta-expansion of the variable x. If the i-th argument in the substitution was a normal term M, the spine will have M at the i-th position. For translating substitutions to spines only approximate types matter, since approximate types carry enough information to support η-expansion. This simplifies the development.

Lemma 7.1 (*Substitutions relate to spines*)
If $\Psi \vdash \sigma \Leftarrow \Psi_0$ and $\Psi \vdash S : [\sigma]^{a}_{\Psi_0} A \Leftarrow P$ and $\mathsf{toSpine}_{\psi_0}\ \sigma\ S = S'$ where $\psi_0 = (\Psi_0)^{-}$
then $\Psi \vdash S' : \Pi\Psi_0.A \Leftarrow P$.

Proof
Structural induction on the first derivation (see appendix). □

‘

Theorem 7.2 (*Soundness of abstraction*)
Let $\vdash \Psi_0, \Psi\ \mathsf{ctx}$.

1. If $(\Psi_0)\Psi \vdash M \Leftarrow B \ /\ (\Psi_1)N$ and $\Psi_0, \Psi \vdash B \Leftarrow \mathsf{type}$ then
   $\Psi_1, \Psi \vdash N \Leftarrow B$ and $\Psi_0 \subseteq \Psi_1$ and $\vdash \Psi_1\ \mathsf{ctx}$.
2. If $(\Psi_0)\Psi \vdash S : A \Leftarrow P \ /\ (\Psi_1)S'$
   and $\Psi_0, \Psi \vdash A \Leftarrow \mathsf{type}$ and $\Psi_0, \Psi \vdash P \Leftarrow \mathsf{type}$ then
   $\Psi_1, \Psi \vdash S' : A \Leftarrow P$ and $\Psi_0 \subseteq \Psi_1$ and $\vdash \Psi_1\ \mathsf{ctx}$.
3. If $(\Psi_0)\Psi \vdash \sigma \Leftarrow \Psi' \ /\ (\Psi_1)\sigma'$ and $\vdash \Psi'\ \mathsf{ctx}$ then
   $\Psi_1, \Psi \vdash \sigma' \Leftarrow \Psi'$ and $\Psi_0 \subseteq \Psi_1$ and $\vdash \Psi_1\ \mathsf{ctx}$.

Proof
Structural induction on the abstraction judgment (see appendix). □

8 Implementation

We implemented the two described phases of type reconstruction together with higher-order unification in OCaml as part of the Beluga implementation. Instead of eagerly applying substitution as we have described in previous section, we employ explicit substitutions (Abadi *et al.*, 1990). There are a few subtle issues which arise when implementing type reconstruction and unification in this setting. The first one is working with η-expanded terms. This is convenient for type checking however as we described we need to be careful when dealing with substitutions, since elements in the substitution may not always be in η-expanded form. In our implementation, we try to keep substitutions in η-contracted form, since this facilitates checking whether a substitution is a pattern substitution.

Closest to our implementation is the one in the Twelf system. There are however several subtle differences. We list here a few examples which illustrate them: Our parser does not

accept objects which contain β-redices, because our implementation is centered around bidirectional type checking and normal forms. This means we can omit for example typing annotations at λ-abstractions. We also for example always create meta-variables as lowered, and hence we do not need to lower them during normalization. Another aspect is our higher-order unification implementation which distinguishes itself in the treatment of the cases outside of the pattern fragment. We have followed a slightly more conservative approach when considering solving the case $u[\sigma] = R$, i.e. where σ is a pattern substitution and u is a meta-variable. In this case, we must ensure that $[\sigma]^{-}1(R)$ exists. This is typically accomplished by pruning the meta-variables in R to obtain some R' s.t. $[\sigma]^{-}1(R')$ is guaranteed to exist. In our implementation, pruning will fail, if we encounter within R a meta-variable $v[\tau]$ outside the pattern fragment and where we cannot guarantee that applying the inverse substitution $[\sigma]^{-}1$ to it will indeed exist. In the Twelf implementation, pruning R will not fail. Instead, it will replace the offending meta-variable $v[\tau]$ with a new meta-variable $w[\mathsf{id}]$ and generate a constraint $v[\tau] = w[\mathsf{id}]$. While this seems sensible at first, it has been noted by Reed (2009) that this may lead to non-termination for some exotic examples.

Finally, we do not support type variables and consequently the user needs to provide at least a type skeleton at Π-abstractions. In practice, this did not seem to cause any real problems and only few Twelf examples needed this additional information and the overall design is simpler if we omit type variables. This feature can be added to our implementation in a pre-processing phase where we compute the approximate type. This a similar strategy as employed in the Twelf system.

At this point, we also do not support type annotations in the implementation, although adding them seems straightforward.

So far, we have tested our implementation on most of the examples from the Twelf example library, and our implementation is competitive.

9 Related work

Type reconstruction for dependently-typed languages is non-trivial, and is in general undecidable (Dowek, 1993). In this guide, we describe the philosophy in systems such as Elf (Pfenning, 1991) and Twelf (Pfenning & Schürmann, 1999). If a constant declaration has an implicit quantifier-prefix, then we must omit those arguments whenever we use this constants. Underscores may be used at any point in the term wherever a term was legal. The analysis is done one constant at a time for LF declarations. We follow the same methodology in the implementation LF type reconstruction in Beluga (Pientka & Dunfield, 2010). Unlike our one-pass reconstruction algorithm, the Twelf system implements type reconstruction in two phases. During the first phase, we for example η-expand terms and infer missing type annotations at Π-types. This first phase is driven by approximate types. In the second phase, we infer the instantiation of omitted arguments and the full type of free variables. There are two advantages to this approach: we can infer the type annotations in Π-types and it leads to improved and more meaningful error messages. On the other hand, establishing the overall correctness of such a two-phase type reconstruction algorithm is more cumbersome, since we need to formalize both phases. We found it also easier to explain the essential challenges in an one-pass algorithm which resembles in spirit the

one-pass Hindley-Milner-style type inference algorithms. Therefore, we prefer to view a two-phase algorithm for LF type reconstruction as an optimization of the presented one-pass algorithm.

Unlike systems such as Agda (Norell, 2007), we provide no explicit way to specify an argument should be synthesized or to explicitly override synthesis. In the case where type reconstruction needs more information, the user needs to supply a typing annotation. However so far we have only discovered one example where such an explicit type annotation is necessary because part of the unification problem is outside the pattern fragment.

In contrast to other systems, we also support free variables, and synthesize their type. This is important in practice. Specifying all variables together with their type up front is cumbersome in many dependently typed constant declarations. To ease the burden on the user, some systems (such as Agda) supports simply listing the variables occurring in a declaration without the type. This however requires that the user chose the right order. Even worse, the user must anticipate all the variables which could occur as implicit arguments.

More importantly, providing the user with the additional flexibility of not specifying the type of free variables leads to a circular dependency between meta-variables and free variables and we can only during abstraction check whether a linear ordering indeed exists.

Finally, we have no typing annotation on lambda-abstraction; this is unnecessary if we have a bi-directional type system, and in fact simplifies the reconstruction.

Elaboration from implicit to explicit syntax has been first mentioned by Pollack (1990) although no concrete algorithm to reconstruct omitted arguments is given. Luther (2001) refined these ideas as part of the TYPELab project. He describes an elaboration and reconstruction for the calculus of construction. This work is closely related to ours, but differs substantially in the presented foundation. Similar to our approach, Luther describes bi-directional elaboration. However, there are some substantial differences between his work and the one we describe here. Our bi-directional type system is driven by characterizing only canonical forms. This allows us for example to omit typing annotations at λ-abstractions while Luther's work does not. We use meta-variables characterized by contextual types during reconstruction. This allows us to generate a dependent type of a meta-variable and our reconstruction algorithm generates a context of dependently typed meta-variables. This allows us to explicitly track and reason about the instantiation of these meta-variables. Finally, reconstructing of a full dependently typed object is done in one pass in our setting, while in Luther's algorithm elaboration and reconstruction are intimately intertwined. This makes it harder to understand its correctness.

Norell (2007) discusses in his PhD thesis elaboration and type reconstruction for Agda which in turn is based on Epigram. Norell considers reconstruction for Martin Löfs type theory, but there is no treatment of synthesizing the type of free variables. The lack of contextual modal types means we cannot easily describe all the meta-variables occurring in a type reconstruction problem and reason about the substitutions for meta-variables. Instead, the foundation is centered around a system of constraints. From a more practical point of view, Agda does not support higher-order unification and not even higher-order pattern unification. The algorithm only solves fully applied patterns, i.e. meta-variables which are applied to all the bound variables in whose scope it occurs in. This avoids the need for pruning, and constraint handling, since unification for this fragment is essentially like first-order unification. This would be too weak in our setting.

Alternatively to reconstructing omitted arguments, one could try to give a direct foundation to implicit syntax. This has been explored in the past for the calculus of construction in (Miquel, 2001; Hagiya & Toda, 1994). In the setting of LF, Jason Reed (2004) provides a foundation for type checking a more compact representation of LF objects directly without first reconstructing it. This is particularly useful in applications where compact proof objects matter such as proof-carrying code. Unfortunately, we loose flexibility and sometimes more information must be supplied to guarantee that we can type check those compact LF objects.

10 Conclusion

We have presented a foundation for type reconstruction in the dependently typed setting of the logical framework LF together with soundness of reconstruction and practical implementation experience. Reconsidering type reconstruction using contextual modal types allows us to reason about substitutions for meta-variables. Our presented guide highlights some of the difficulties we encounter such as inferring the type of a free variable, ensuring bound variable dependencies, η-expansion and η-contraction, and abstraction over free and meta-variables. We believe this foundation is particularly important as we consider type reconstruction for dependently type programming languages which feature pattern matching and recursion.

In the future, we plan to extend the presented reconstruction framework to formally describe type reconstruction for programming with dependently-typed higher-order data as found in Beluga (Pientka & Dunfield, 2008).

Acknowledgements

I would like to thank Andreas Abel, Joshua Dunfield, Renaud Germain, Stefan Monnier, and Jason Reed for their discussions and comments on earlier drafts. In particular, Jason Reed's comments lead me to simplify the treatment of eta-contraction and eta-expansion.

References

Abadi, Martin, Cardelli, Luca, Curien, Pierre-Louis, & Lèvy, Jean-Jacques. (1990). Explicit substitutions. *Pages 31–46 of: 17th annual ACM sigplan-sigact symposium on principles of programming languages, san francisco, california*. ACM Press.

Aydemir, B., Charguéraud, Arthur, Pierce, B., Pollack, Randy, & Weirich., Stephanie. (2008). Engineering formal metatheory. *Page to appear of:* Wadler, Phil (ed), *35th annual ACM sigplan-sigact symposium on principles of programming languages*. ACM.

Bertot, Yves, & Castéran, Pierre. (2004). *Interactive theorem proving and program development. coq'art: The calculus of inductive constructions*. Springer.

Boespflug, Mathieu. (2010). *The Dedukti Proof Checker*.

Cervesato, Iliano, & Pfenning, Frank. (2003). A linear spine calculus. *Journal of logic and computation*, **13**(5), 639–688.

Crary, Karl. (2003). Toward a foundational typed assembly language. *Pages 198–212 of: 30th acm sigplan-sigact symposium on principles of programming languages (popl'03)*. New Orleans, Louisisana: ACM-Press.

Dowek, Gilles. (1993). The undecidability of typability in the lambda-pi-calculus. *Pages 139–145 of: International conference on typed lambda calculi and applications(tlca '93)*. London, UK: Springer-Verlag.

Dowek, Gilles, Hardin, Thérèse, & Kirchner, Claude. (1995). Higher-order unification via explicit substitutions. *Pages 366–374 of:* Kozen, D. (ed), *Proceedings of the tenth annual symposium on logic in computer science*. San Diego, California: IEEE Computer Society Press.

Dowek, Gilles, Hardin, Thérèse, Kirchner, Claude, & Pfenning, Frank. (1996). Unification via explicit substitutions: The case of higher-order patterns. *Pages 259–273 of:* Maher, M. (ed), *Proceedings of the joint international conference and symposium on logic programming*. Bonn, Germany: MIT Press.

Hagiya, Masami, & Toda, Yozo. (1994). On implicit arguments. *Pages 10–30 of:* Jones, Neil D., Hagiya, Masami, & Sato, Masahiko (eds), *Logic, language and computation, festschrift in honor of satoru takasu*. Lecture Notes in Computer Science, vol. 792. Springer.

Harper, Robert, & Licata, Daniel R. (2007). Mechanizing metatheory in a logical framework. *Journal of functional programming*, **17**(4-5), 613–673.

Harper, Robert, Honsell, Furio, & Plotkin, Gordon. (1993). A framework for defining logics. *Journal of the acm*, **40**(1), 143–184.

Lee, Daniel K., Crary, Karl, & Harper, Robert. (2007). Towards a Mechanized Metatheory of Standard ML. *Pages 173–184 of: 34th annual acm sigplan-sigact symposium on principles of programming languages(popl'07)*. New York, NY, USA: ACM Press.

Licata, Daniel R., & Harper, Robert. 2007 (July). *An extensible theory of indexed types*. Unpublished manuscript.

Licata, Daniel R., Zeilberger, Noam, & Harper, Robert. (2008). Focusing on binding and computation. *Pages 241–252 of:* Pfenning, F. (ed), *23rd symposium on logic in computer science*. IEEE Computer Society Press.

Luther, Marko. (2001). More on implicit syntax. *Pages 386–400 of:* Gore, R., Leitsch, A., & Nipkow, T. (eds), *Proceedings of the first international joint conference on automated reasoning, siena, italy*. Lecture Notes in Artificial Intelligence (LNAI) 2083. Springer.

McBride, Conor, & McKinna, James. (2004). The view from the left. *Journal of functional programming*, **14**(1), 69–111.

Miller, Dale. (1991). Unification of simply typed lambda-terms as logic programming. *Pages 255–269 of: Eighth international logic programming conference*. Paris, France: MIT Press.

Miquel, Alexandre. (2001). The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. *Pages 344–359 of:* Abramsky, S. (ed), *5th international conference on typed lambda calculi and aplications*. Lecture Notes in Computer Science (LNCS 2044). Springer.

Nanevski, Aleksandar, Pfenning, Frank, & Pientka, Brigitte. (2008). Contextual modal type theory. *Acm transactions on computational logic*, **9**(3), 1–49.

Necula, George C. (1997). Proof-carrying code. *Pages 106–119 of: 24th annual symposium on principles of programming languages (popl'97)*. ACM Press.

Necula, George C., & Lee, Peter. (1998). Efficient representation and validation of logical proofs. *Pages 93–104 of:* Pratt, Vaughan (ed), *Proceedings of the 13th annual symposium on logic in computer science (lics'98)*. Indianapolis, Indiana: IEEE Computer Society Press.

Norell, Ulf. 2007 (sep). *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology. Technical Report 33D.

Pfenning, Frank. (1991). Logic programming in the LF logical framework. *Pages 149–181 of:* Huet, Gérard, & Plotkin, Gordon (eds), *Logical frameworks*. Cambridge University Press.

Pfenning, Frank. (1997). *Computation and deduction*.

Pfenning, Frank, & Schürmann, Carsten. (1999). System description: Twelf — a meta-logical framework for deductive systems. *Pages 202–206 of:* Ganzinger, H. (ed), *Proceedings of the 16th international conference on automated deduction (cade-16)*. Lecture Notes in Artificial Intelligence, vol. 1632. Springer.

Pientka, Brigitte. (2003). *Tabled higher-order logic programming*. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University. CMU-CS-03-185.

Pientka, Brigitte. (2007). Proof pearl: The power of higher-order encodings in the logical framework LF. *Pages 246–261 of:* Schneider, Klaus, & Brandt, Jens (eds), *20th international conference on theorem proving in higher order logics (tphols'07), kaiserslautern, germany*. Lecture Notes in Computer Science (LNCS 4732). Springer.

Pientka, Brigitte. (2008). A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. *Pages 371–382 of: 35th annual ACM sigplan-sigact symposium on principles of programming languages (popl'08)*. ACM Press.

Pientka, Brigitte, & Dunfield, Joshua. (2008). Programming with proofs and explicit contexts. *Pages 163–173 of: Acm sigplan symposium on principles and practice of declarative programming (ppdp'08)*. ACM Press.

Pientka, Brigitte, & Dunfield, Joshua. (2010). Beluga:a Framework for Programming and Reasoning with Deductive Systems (System Description). Giesl, Jürgen, & Haehnle, Reiner (eds), *5th international joint conference on automated reasoning (ijcar'10)*. Lecture Notes in Artificial Intelligence (LNAI).

Pollack, Robert. (1990). *Implicit syntax*. Informal Proceedings of First Workshop on Logical Frameworks, Antibes.

Poswolsky, Adam B., & Schürmann, Carsten. (2008). Practical programming with higher-order encodings and dependent types. *Page 93 of: Proceedings of the 17th european symposium on programming (esop '08)*, vol. 4960. Springer.

Reed, Jason. 2004 (July). Redundancy elimination for LF. Schürmann, C. (ed), *Fourth workshop on logical frameworks and meta-languages (lfm '04)*.

Reed, Jason. (2009). Higher-order constraint simplification in dependent type theory. Felty, A., & Cheney, J. (eds), *International workshop on logical frameworks and meta-languages: Theory and practice (lfmtp'09)*.

Watkins, Kevin, Cervesato, Iliano, Pfenning, Frank, & Walker, David. (2002). *A concurrent logical framework I: Judgments and properties*. Tech. rept. CMU-CS-02-101. Department of Computer Science, Carnegie Mellon University.

A Appendix

A.1 Ordinary substitution

In the definition for ordinary substitutions, we need to be careful because the only meaningful terms are those in canonical form. To ensure that substitution preserves canonical form, we use a technique pioneered by (Watkins *et al.*, 2002) and described in detail in (Nanevski *et al.*, 2008). The idea is to define *hereditary substitution* as a primitive recursive functional that always returns a canonical object.

In the formal development, it is simpler if we can stick to non-dependent types. We therefore first define type approximations α and an erasure operation $()^-$ that removes dependencies. Before applying any hereditary substitution $[M/x]^a_A(B)$ we first erase dependencies to obtain $\alpha = A^-$ and then carry out the hereditary substitution formally as $[M/x]^a_\alpha(B)$. A similar convention applies to the other forms of hereditary substitutions. Types relate to type approximations via an erasure operation $()^-$ which we overload to work on types.

$$\begin{array}{rcl} \text{Type approximations} \quad \alpha,\beta & ::= & a \mid \alpha \to \beta \\ (a\, N_1 \ldots N_n)^- & = & a \\ (\Pi x{:}A \ldots B)^- & = & A^- \to B^- \end{array}$$

We can define $[M/x]^n_\alpha(N)$, $[M/x]^r_\alpha(R)$, and $[M/x]^s_\alpha(\sigma)$ by nested induction, first on the structure of the type approximation α and second on the structure of the objects N, R and σ. In other words, we either go to a smaller type approximation (in which case the objects can become larger), or the type approximation remains the same and the objects become smaller. The following hereditary substitution operations are defined in Figure A 1.

$$\begin{array}{rcll} [M/x]^n_\alpha(N) & = & N' & \text{Normal terms } N \\ [M/x]^l_\alpha(S) & = & S' & \text{Spine } S \\ [M/x]^s_\alpha(\sigma) & = & \sigma' & \text{Substitutions } \sigma \end{array}$$

Next, we concentrate on eliminating possible redices which may have been created in the case $[M/x]^n_\alpha(x \cdot S)$ using the definition of $\mathsf{reduce}(M : \alpha, S)$.

$$\begin{array}{lcl} \mathsf{reduce}(\lambda y.M : \alpha_1 \to \alpha_2, (N;S)) & = & M'' \\ \quad \text{where} [N/y]^n_{\alpha_1} M = M' \text{ and } \mathsf{reduce}(M' : \alpha_2, S) = M'' & & \\ \mathsf{reduce}(R : a, \mathsf{nil}) & = & R \\ \mathsf{reduce}(M : \alpha, S) & & \text{fails} \quad \text{otherwise} \end{array}$$

We first compute the result of applying the substitution $[M/x]$ to the spine S which yields the spine S'. Second, we reduce any possible redices which are created using the given definition.

Substitution may fail to be defined only if substitutions into the subterms are undefined. The side conditions $y \notin \mathsf{FV}(M)$ and $y \neq x$ do not cause failure, because they can always be satisfied by appropriately renaming y. However, substitution may be undefined if we try for example to substitute an atomic term R for x in the term $x \cdot S$ where the spine S is non-empty. The substitution operation is well-founded since recursive appeals to the substitution operation take place on smaller terms with equal type A, or the substitution operates on smaller types (see the case for $\mathsf{reduce}(\lambda y.M : \alpha_1 \to \alpha_2, (N;S))$).

Normal terms

$$
\begin{array}{lll}
[M/x]^n_\alpha(\lambda y.N) = \lambda y.N' & & \text{where } N' = [M/x]^n_\alpha(N) \\
 & & \text{choosing } y \notin \mathsf{FV}(M) \\
 & & \text{and } y \neq x \\
[M/x]^r_\alpha(u[\sigma]) = u[\sigma'] & & \text{where } \sigma' = [M/x]^s_\alpha(\sigma) \\
[M/x]^n_\alpha(\mathbf{c}\cdot S) = \mathbf{c}\cdot S' & & \text{where } S' = [M/x]^l_\alpha S \\
[M/x]^n_\alpha(X\cdot S) = X\cdot S' & & \text{where } S' = [M/x]^l_\alpha S \\
[M/x]^n_\alpha(x\cdot S) = \mathsf{reduce}(M:\alpha, S') & & \text{where } S' = [M/x]^l_\alpha S \\
[M/x]^n_\alpha(y\cdot S) = y\cdot S' & & \text{where } y \neq x \\
 & & \text{and } S' = [M/x]^l_\alpha S
\end{array}
$$

Spines

$$
\begin{array}{lll}
[M/x]^l_\alpha(\mathsf{nil}) = \mathsf{nil} & & \\
[M/x]^l_\alpha(N;S) = N';S' & & \text{where } N' = [M/x]^n_\alpha N \text{ and } S' = [M/x]^l_\alpha S
\end{array}
$$

Substitutions

$$
\begin{array}{lll}
[M/x]^s_\alpha(\cdot) = \cdot & & \\
[M/x]^s_\alpha(\sigma,N) = (\sigma',N') & & \text{where } \sigma' = [M/x]^s_\alpha\sigma \text{ and } N' = [M/x]^n_\alpha N \\
[M/x]^s_\alpha(\sigma;y') = (\sigma';y') & & \text{where } \sigma' = [M/x]^s_\alpha\sigma \text{ and } x \neq y' \\
[M/x]^s_\alpha(\sigma;x) = (\sigma',M) & & \text{where } \sigma' = [M/x]^s_\alpha\sigma
\end{array}
$$

Fig. A 1. Ordinary substitution

We write $\alpha \leq \beta$ and $\alpha < \beta$ if α occurs in β (as a proper subexpression in the latter case). If the original term is not well-typed, a hereditary substitution, though terminating, cannot always return a meaningful term. We formalize this as failure to return a result. However, on well-typed terms, hereditary substitution will always return well-typed terms. This substitution operation can be extended to types for which we write $[M/x]^a_\alpha(A)$. The operation $[M/x]^*_\alpha(\_)$ where $* = \{a,n,l,s\}$ terminates, either by returning a result or failing after a finite number of steps.

Theorem 4.1 (*Hereditary Substitution Principles*)

If $\Delta;\Psi \vdash M \Leftarrow A$ and $\Delta;\Psi,x{:}A,\Psi' \vdash J$ then
$\Delta;\Psi,[M/x]^*_\alpha\Psi' \vdash [M/x]^*_\alpha(J)$ where $* = \{a,n,l,s\}$.

Building on (Nanevski *et al.*, 2008), we can also define simultaneous substitution $[\sigma]^n_\psi(M)$ (respectively $[\sigma]^l_\psi(S)$ and $[\sigma]^s_\psi(\sigma)$). We write ψ for the context approximation of Ψ which is defined using the erasure operation $()^-$.

$$
\begin{array}{lcl}
(\cdot)^- & = & \cdot \\
(\Psi,x{:}A)^- & = & (\Psi)^-,x{:}(A)^-
\end{array}
$$

A.2 Substitution for contextual variables

In this section, we summarize contextual substitution. Just as we annotated the substitution $[M/x]_A$ with the type of the variable x, we will annotate the contextual substitution $[\![\Psi.R/u]\!]_{P[\Psi]}$ with the type of the meta-variable $P[\Psi]$.

$$
\begin{array}{lcll}
[\![\hat{\Psi}.R/u]\!]^n_{P[\Psi]}(\lambda y.N) & = & \lambda y.N' & \text{where } N' = [\![\hat{\Psi}.R/u]\!]^n_{P[\Psi]}N \\
[\![\hat{\Psi}.R/u]\!]^n_{P[\Psi]}(\mathbf{c}\cdot S) & = & \mathbf{c}\cdot S' & \text{where } S' = [\![\hat{\Psi}.R/u]\!]^l_{P[\Psi]}S \\
[\![\hat{\Psi}.R/u]\!]^n_{P[\Psi]}(X\cdot S) & = & X\cdot S' & \text{where } S' = [\![\hat{\Psi}.R/u]\!]^l_{P[\Psi]}S \\
[\![\hat{\Psi}.R/u]\!]^n_{P[\Psi]}(x\cdot S) & = & x\cdot S' & \text{where } S' = [\![\hat{\Psi}.R/u]\!]^l_{P[\Psi]}S \\
[\![\hat{\Psi}.R/u]\!]^n_{P[\Psi]}(u[\sigma]) & = & R' & \text{where } \sigma' = [\![\hat{\Psi}.R/u]\!]^s_{P[\Psi]}\sigma \text{ and } R' = [\sigma']^n_{\Psi}R \\
[\![\hat{\Psi}.R/u]\!]_P(v[\sigma]) & = & v[\sigma'] & \text{where } \sigma' = [\![\hat{\Psi}.R/u]\!]^s_{P[\Psi]}\sigma \text{ and provided } v \neq u \\
[\![\hat{\Psi}.R/u]\!]^l_{P[\Psi]}(\mathsf{nil}) & = & \mathsf{nil} & \\
[\![\hat{\Psi}.R/u]\!]^l_{P[\Psi]}(N;S) & = & N';S' & \text{where } N' = [\![\hat{\Psi}.R/u]\!]^n_{P[\Psi]}N \text{ and } S' = [\![\hat{\Psi}.R/u]\!]^l_{P[\Psi]}S \\
[\![\hat{\Psi}.R/u]\!]^s_{P[\Psi]}(\cdot) & = & \cdot & \\
[\![\hat{\Psi}.R/u]\!]^s_{P[\Psi]}(\sigma,N) & = & \sigma',N' & \text{where } \sigma' = [\![\hat{\Psi}.R/u]\!]^s_{P[\Psi]}\sigma \text{ and } N' = [\![\hat{\Psi}.R/u]\!]^n_{P[\Psi]}N \\
[\![\hat{\Psi}.R/u]\!]^s_{P[\Psi]}(\sigma;x) & = & \sigma';x & \text{where } \sigma' = [\![\hat{\Psi}.R/u]\!]^s_{P[\Psi]}\sigma
\end{array}
$$

Fig. A 2. Substitution for meta-variables

Applying $[\![\hat{\Psi}.R/u]\!]$ to the closure $u[\sigma]$ first obtains the simultaneous substitution $\sigma' = [\![\Psi.R/u]\!]\sigma$, but instead of returning $R[\sigma']$, it proceeds to eagerly apply σ' to R. We define the operations in Figure A 2.

We note that maintaining canonical forms is easy since we enforce that every occurrence of a meta-variable must have base type. The computation of σ' recursively invokes $[\![\hat{\Psi}.R/u]\!]$ on σ, a constituent of $u[\sigma]$. Then τ' is applied to R, but applying simultaneous substitutions has already been defined without appeal to meta-variable substitution. $[\![\hat{\Psi}.R/u]\!]^*_{P[\Psi]}(\_)$ and where $* \in \{a,n,l,s\}$ terminate, either by returning a result or failing after a finite number of steps.

Theorem 4.2 (*Contextual Substitution Principles*)
If $\Delta_1;\Phi \vdash R \Leftarrow P$ and $\Delta_1, u{::}P[\Phi], \Delta_2;\Psi \vdash J$ then
$\Delta_1, [\![\hat{\Psi}.R/u]\!]^*_{P[\Psi]}\Delta_2; [\![\hat{\Psi}.R/u]\!]^*_{P[\Psi]}\Psi \vdash [\![\hat{\Psi}.R/u]\!]^*_{P[\Psi]}J$ where $* = \{a,n,l,s\}$.

A.3 Key lemmas about η-expansion

Theorem 4.4

1. If $\Upsilon;\Phi;\Psi_1, x{:}A, \Psi_2 \vdash N \Leftarrow B$ then $[\eta\mathsf{exp}_A(x)/x]^n_A N = N$.
2. If $\Upsilon;\Phi;\Psi_1, x{:}A, \Psi_2 \vdash S : B \Leftarrow P$ then $[\eta\mathsf{exp}_A(x)/x]^l_A S = S$.
3. If $\Upsilon;\Phi;\Psi_1, x{:}A, \Psi_2 \vdash \sigma \Leftarrow \Psi$ then $[\eta\mathsf{exp}_A(x)/x]^s_\alpha \sigma = \sigma$.
4. If $\Upsilon;\Phi;\Psi \vdash M \Leftarrow A$ then $[M/y]^n_A(\eta\mathsf{exp}_A(y)) = M$.

Proof
By mutual induction on N, S, and A.

Statement 1

$$\textbf{Case 1}\ \ \mathscr{D} = \frac{\Upsilon;\Phi;\Psi_1, x{:}A, \Psi_2, y{:}B_1 \vdash N' \Leftarrow B_2}{\Upsilon;\Phi;\Psi_1, x{:}A, \Psi_2 \vdash \lambda y.N' \Leftarrow \Pi y{:}B_1.B_2}$$

$[\eta\mathsf{exp}_A(x)/x]^n_A N' = N'$ by i.h.
$[\eta\mathsf{exp}_A(x)/x]^n_A(\lambda y.N') = \lambda y.[\eta\mathsf{exp}_A(x)/x]^n_A N' = \lambda y.N'$ by subst. properties

Case 2 $\mathscr{D} = \dfrac{\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash S : A \Leftarrow P}{\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash x\cdot S \Leftarrow P}$

$[\eta\mathsf{exp}_A(x)/x]^l_A S = S$ by i.h.
$[\eta\mathsf{exp}_A(x)/x]^n_A(x\cdot S) =$ to show
$= \mathsf{reduce}(\eta\mathsf{exp}_A(x) : A^-, S) =$ by substitution definition
w.l.g. $A = \Pi\Psi.P$ and $\Psi = x_1{:}B_1,\ldots,x_n{:}B_n$
$= \mathsf{reduce}(\lambda\hat{\Psi}.x\cdot(\eta\mathsf{exp}_{B_1}(x_1);\ldots;\eta\mathsf{exp}_{B_n}(x_n);\mathsf{nil}) : A^-, S)$ by definition of $\eta\mathsf{exp}$
w.l.g. $S = M_1;\ldots;M_n;\mathsf{nil}$ and $|\Psi| = n$
$= [M_n/x_n]^n_{B_n}\ldots[M_1/x_1]^n_{B_1}(x\cdot(\eta\mathsf{exp}_{B_1}(x_1);\ldots;\eta\mathsf{exp}_{B_n}(x_n);\mathsf{nil})))$
by n-times reduce-definition
$= x\cdot([M_1/x_1]^n_{B_1}\eta\mathsf{exp}_{B_1}(x_1);\ldots;[M_n/x_n]^n_{B_n}\eta\mathsf{exp}_{B_n}(x_n);\mathsf{nil})$
by def. substitution and lemma 4.3
$x\cdot M_1;\ldots;M_n;\mathsf{nil}$ since $[M_i/x_i]^n_{B_i}(\eta\mathsf{exp}_{B_i}(x_i) = M_i$ for all i by i.h. (3)
$= x\cdot S$ by previous lines

Statement 2

Case 1 $\mathscr{D} = \dfrac{}{\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash \mathsf{nil} : P \Leftarrow P}$

$[\eta\mathsf{exp}_A(x)/x]^l_A\mathsf{nil} = \mathsf{nil}$ by definition

Case 2 $\mathscr{D} = \dfrac{\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash M \Leftarrow B_1 \quad \Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash S : [M/x]^n_{B_1}B_2 \Leftarrow P}{\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash M;S : \Pi x{:}B_1.B_2 \Leftarrow P}$

$(B_2)^- = ([M/x]^a_{B_1}B_2)^-$ by erasure
$[\eta\mathsf{exp}_A(x)/x]^n_A M = M$ by i.h. (1)
$[\eta\mathsf{exp}_A(x)/x]^l_A S = S$ by i.h. (2)
$[\eta\mathsf{exp}_A(x)/x]^l_A(M;S) = [\eta\mathsf{exp}_A(x)/x]^n_A M;[\eta\mathsf{exp}_A(x)/x]^l_A S$ by definition

Statement 3

Case 1 $\mathscr{D} = \dfrac{}{\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash \cdot \Leftarrow \Psi}$

$[\eta\mathsf{exp}_A(x)/x]^s_A(\cdot) = \cdot$ by rule

Case 2 $\mathscr{D} = \dfrac{\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash \sigma \Leftarrow \Psi \quad \Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash M \Leftarrow B}{\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash \sigma,M \Leftarrow \Psi,y{:}B}$

$[\eta\mathsf{exp}_A(x)/x]^n_A(M) = M$ by i.h. (1)
$[\eta\mathsf{exp}_A(x)/x]^s_A(\sigma) = \sigma$ by i.h. (3)
$[\eta\mathsf{exp}_A(x)/x]^s_A(\sigma,M) = [\eta\mathsf{exp}_A(x)/x]^s_A\sigma,[\eta\mathsf{exp}_A(x)/x]^n_A M = \sigma,M$ by substitution

Case 3 $\mathscr{D} = \dfrac{\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash \sigma \Leftarrow \Psi \quad (\Psi_1,x{:}A,\Psi_2)(y) = B' \quad [\sigma]^s_\Psi B' = B}{\Upsilon;\Phi;\Psi_1,x{:}A,\Psi_2 \vdash \sigma;y \Leftarrow \Psi,y{:}B}$

$[\eta\mathsf{exp}_A(x)/x]^s_A(\sigma) = \sigma$ by i.h. (3)
$[\eta\mathsf{exp}_A(x)/x]^s_A(\sigma;x) = [\eta\mathsf{exp}_A(x)/x]^s_A(\sigma), \eta\mathsf{exp}_A(x)$ by substitution
$= \sigma;x$ since $\eta\mathsf{con}(\eta\mathsf{exp}_A(x)) = x$ by lemma 4.3

Statement 4 Let $A = \Pi\Psi.P$ and $M = \lambda\hat{\Psi}.R$, since M checks against type A. Moreover, let $\Psi = x_1{:}B_1,\ldots,x_n{:}B_n$.

$[\lambda\hat{\Psi}.R/y]^n_A(\eta\mathsf{exp}_A(y)) = [\lambda\hat{\Psi}.R/y]^n_A(\lambda\hat{\Psi}.y\cdot(\eta\mathsf{exp}_{B_1}(x_1);\ldots;\eta\mathsf{exp}_{B_n}(x_n);\mathsf{nil}))$
by definition of $\eta\mathsf{exp}$
$= \lambda\hat{\Psi}.\mathsf{reduce}(\lambda\hat{\Psi}.R : A^-, [\lambda\hat{\Psi}.R/y]^n_A(\eta\mathsf{exp}_{B_1}(x_1);\ldots;\eta\mathsf{exp}_{B_n}(x_n);\mathsf{nil})))$
by definition
$= \lambda\hat{\Psi}.\mathsf{reduce}(\lambda\hat{\Psi}.R : A^-, (\eta\mathsf{exp}_{B_1}(x_1);\ldots;\eta\mathsf{exp}_{B_n}(x_n);\mathsf{nil}))$
since $y \notin \mathsf{FV}(\eta\mathsf{exp}_{B_i}(y_i))$ (see also lemma 4.3)
$= \lambda\hat{\Psi}.[\eta\mathsf{exp}_{B_n}(x_n)/x_n]^n_{B_n}\ldots[\eta\mathsf{exp}_{B_1}(x_1)/x_1]^n_{B_1}R$ by n-times reduce
$= \lambda\hat{\Psi}.R$ by n-times i.h. (1)
□

Lemma 5.1
If $\eta\mathsf{con}(m) = x$ and $\Psi^- \vdash m \Leftarrow A^-$ and $\Psi(x) = A$ then $\Psi^- \vdash m \approx \eta\mathsf{exp}_A(x) : A^-$.

Proof
Induction on A.

Let $A = \Pi\Psi_0.Q_0$ and $\Psi_0 = y_1{:}B_1,\ldots,y_n{:}B_n$.
Moreover, by definition of η-contraction we have $m = \lambda y_1\ldots\lambda y_k.x\cdot(m_1;\ldots m_k;\mathsf{nil})$ where for all i, $\eta\mathsf{con}(m_i) = y_i$.
$\Psi^- \vdash m \Leftarrow A^-$ by assumption
$\Psi^-, y_1{:}B_1{}^-,..,y_k{:}B_k{}^- \vdash x\cdot(m_1;\ldots m_k;\mathsf{nil}) \Leftarrow (\Pi y_{k+1}{:}B_{k+1},\ldots y_n{:}B_n.Q_0)^-$
by typing inversion
$\Psi^-, y_1{:}B_1{}^-,..,y_k{:}B_k{}^-, y_{k+1}{:}B_{k+1}{}^-,\ldots y_n{:}B_n{}^-$
$\vdash x\cdot(m_1;\ldots;m_k;(y_{k+1}\cdot\mathsf{nil});\ldots;(y_n\cdot\mathsf{nil});\mathsf{nil}) \Leftarrow (Q_0)^-$
by typing inversion
for all $1 \le j \le k$, we have
$\Psi^-, y_1{:}B_1{}^-,..,y_k{:}B_k{}^-, y_{k+1}{:}B_{k+1}{}^-,\ldots y_n{:}B_n{}^- \vdash m_j \Leftarrow B_j{}^-$ by typing inversion
for all $k+1 \le j \le n$, we have $\eta\mathsf{con}(y_j\cdot\mathsf{nil}) = y_j$ and
$(\Psi^-, y_1{:}B_1{}^-,..,y_k{:}B_k{}^-, y_{k+1}{:}B_{k+1}{}^-,\ldots y_n{:}B_n{}^-)(y_j) = B_j{}^-$ and
$(\Psi^-, y_1{:}B_1{}^-,..,y_k{:}B_k{}^-, y_{k+1}{:}B_{k+1}{}^-,\ldots y_n{:}B_n{}^-) \vdash y_j\cdot\mathsf{nil} \Leftarrow B_j{}^-$ by typing rules
for all $1 \le i \le k$, we have
$(\Psi^-, y_1{:}B_1{}^-,..,y_k{:}B_k{}^-, y_{k+1}{:}B_{k+1}{}^-,\ldots y_n{:}B_n{}^-) \vdash m_i \approx \eta\mathsf{exp}_{B_i}(y_i){:}B_i{}^-$ by i.h.
for all $k+1 \le i \le n$, we have
$(\Psi^-, y_1{:}B_1{}^-,..,y_k{:}B_k{}^-, y_{k+1}{:}B_{k+1}{}^-,\ldots y_n{:}B_n{}^-) \vdash y_i\cdot\mathsf{nil} \approx \eta\mathsf{exp}_{B_i}(y_i){:}B_i{}^-$ by i.h.
$(\Psi^-, y_1{:}B_1{}^-,..,y_k{:}B_k{}^-, y_{k+1}{:}B_{k+1}{}^-,\ldots y_n{:}B_n{}^-)$
$\vdash m_1;\ldots;m_k;(y_{k+1}\cdot\mathsf{nil});\ldots;(y_n\cdot\mathsf{nil});\mathsf{nil}$

$$\approx \eta\mathsf{exp}_{B_1}(y_1);\ldots;\eta\mathsf{exp}_{B_k}(y_k);\ \eta\mathsf{exp}_{B_{k+1}}(y_{k+1});\ldots;\eta\mathsf{exp}_{B_n}(y_n);\mathsf{nil} : (\Pi\Psi_0.Q_0)^- \Leftarrow (Q_0)^-$$

$$(\Psi^-, y_1{:}B_1{}^-, .\,, y_k{:}B_k{}^-, y_{k+1}{:}B_{k+1}{}^-, \ldots y_n{:}B_n{}^-)$$
$$\vdash x\cdot m_1;\ldots;m_k;(y_{k+1}\cdot\mathsf{nil});\ldots;(y_n\cdot\mathsf{nil});\mathsf{nil}$$
$$\approx x\cdot(\eta\mathsf{exp}_{B_1}(y_1);\ldots;\eta\mathsf{exp}_{B_k}(y_k);\eta\mathsf{exp}_{B_{k+1}}(y_{k+1});\ldots;\eta\mathsf{exp}_{B_n}(y_n);\mathsf{nil}) \Leftarrow (Q_0)^-$$

by typing rule

$$\Psi^- \vdash \lambda y_1\ldots\lambda y_k.x\cdot m_1;\ldots;m_k;\mathsf{nil}$$
$$\approx \lambda y_1\ldots\lambda y_n.x\cdot(\eta\mathsf{exp}_{B_1}(y_1);\ldots;\eta\mathsf{exp}_{B_n}(y_n);\mathsf{nil}) \Leftarrow (\Pi\Psi_0.Q_0)^-$$

by typing rules (η-rule and lambda-rule)

$$\Psi^- \vdash \lambda y_1\ldots\lambda y_k.x\cdot m_1;\ldots;m_k;\mathsf{nil} \approx \eta\mathsf{exp}_{\Pi\Psi_0\cdot Q_0}(x) \Leftarrow (\Pi\Psi_0.Q_0)^-$$

by definition of η-expansion

□

A.4 Soundness and completeness proof for type reconstruction

Theorem 6.2 (*Soundness of reconstruction*)

1. If $\Upsilon_1;\Phi_1;\Psi \vdash m \Leftarrow A\ /_\rho\ (\Upsilon_2\ ;\ \Phi_2)M$ then
   $\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash M \Leftarrow [\![\rho]\!]A$ and $\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash m \approx M : A^-$
   $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ and $\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx and $[\![\rho]\!]\Phi_1 \subseteq \Phi_2$.
2. If $\Upsilon_1;\Phi_1;\Psi \vdash^i s : A \Leftarrow Q\ /_\rho\ (\Upsilon_2\ ;\ \Phi_2)S$ then
   $\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash S : [\![\rho]\!]A \Leftarrow [\![\rho]\!]Q$ and $\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash^i s \approx S : A^- \Leftarrow Q^-$
   $[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ and $\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx.
3. If $\Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow Q\ /_\rho\ (\Upsilon_2\ ;\ \Phi_2)S$ then
   $\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash S : [\![\rho]\!]A \Leftarrow [\![\rho]\!]Q$ and $\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash s \approx S : A^- \Leftarrow Q^-$
   $[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ and $\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx.
4. If $\Upsilon_1;\Phi_1;\Psi \vdash s \Leftarrow P\ /\ S : A$ then
   $\Upsilon_1;\Phi_1;\Psi \vdash S : A \Leftarrow P$ and $\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash s \approx S : A^- \Leftarrow P^-$ and
   $\Upsilon_1;\Phi_1;\Psi \vdash A \Leftarrow \mathsf{type}$.

Proof
Proof by structural induction on the reconstruction judgment (see appendix).

Statement (1) : Normal terms

Case $\mathscr{D} = \dfrac{\Upsilon_1;\Phi_1;\Psi,x{:}A \vdash m \Leftarrow B\ /_\rho\ (\Upsilon_2\ ;\ \Phi_2)M}{\Upsilon_1;\Phi_1;\Psi \vdash \lambda x.m \Leftarrow \Pi x{:}A.B\ /_\rho\ (\Upsilon_2\ ;\ \Phi_2)\lambda x.M}$

$\Upsilon_2;\Phi_2;[\![\rho]\!](\Psi,x{:}A) \vdash M \Leftarrow [\![\rho]\!]B$	by i.h. (1)
$\Upsilon_2{}^-;\Phi_2{}^-;(\Psi,x{:}A)^- \vdash m \approx M : B^-$	by i.h. (1)
$[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$	by i.h. (1)
$\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx	by i.h. (1)
$\Upsilon_2{}^-;\Phi_2{}^-;\Psi^-,x{:}A^- \vdash m \approx M : B^-$	by definition of erasure
$\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash \lambda x.m \approx \lambda x.M : \Pi x{:}A^-.B^-$	by equivalence relation $\approx$
$\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash \lambda x.m \approx \lambda x.M : (\Pi x{:}A.B)^-$	by definition of erasure
$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi,x{:}[\![\rho]\!]A \vdash M \Leftarrow [\![\rho]\!]B$	by definition of substitution

$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash \lambda x.M \Leftarrow \Pi x{:}[\![\rho]\!]A.[\![\rho]\!]B$ by typing rule
$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash \lambda x.M \Leftarrow [\![\rho]\!]\Pi x{:}A.B$ by definition of substitution

$$\textbf{Case } \mathscr{D} = \frac{\Sigma(\mathbf{c}) = (A,i) \quad \Upsilon_1;\Phi_1;\Psi \vdash^i s : A \Leftarrow P /_\rho (\Upsilon_2 ; \Phi_2)S}{\Upsilon_1;\Phi_1;\Psi \vdash \mathbf{c}\cdot s \Leftarrow P /_\rho (\Upsilon_2 ; \Phi_2)\mathbf{c}\cdot S}$$

$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash S : [\![\rho]\!]A \Leftarrow [\![\rho]\!]P$ by i.h. (2)
$\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash^i s \approx S : A^- \Leftarrow P^-$ by i.h. (2)
$[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ by i.h. (2)
$\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx by i.h. (2)
$\Sigma(\mathbf{c}) = (A,i)$ and A is closed by assumptions
$\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash \mathbf{c}\cdot s \approx \mathbf{c}\cdot S \Leftarrow P^-$ by equivalence relation $\approx$
$[\![\rho]\!]A = A$ by substitution definition
$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash \mathbf{c}\cdot S \Leftarrow [\![\rho]\!]P$ by typing rule

$$\textbf{Case } \mathscr{D} = \frac{x{:}A \in \Psi \quad \Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow P /_\rho (\Upsilon_2 ; \Phi_2)S}{\Upsilon_1;\Phi_1;\Psi \vdash x\cdot s \Leftarrow P /_\rho (\Upsilon_2 ; \Phi_2)x\cdot S}$$

$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash S : [\![\rho]\!]A \Leftarrow [\![\rho]\!]P$ by i.h. (3)
$\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash s \approx S : A^- \Leftarrow P^-$ by i.h. (3)
$[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ by i.h. (3)
$\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx by i.h. (3)
$x{:}[\![\rho]\!]A \in [\![\rho]\!]\Psi$ by previous lines
$\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash x\cdot s \approx x\cdot S : P^-$ by equivalence relation $\approx$
$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash x\cdot S \Leftarrow [\![\rho]\!]P$ by typing rule

$$\textbf{Case } \mathscr{D} = \frac{\begin{array}{ll} s \text{ is a pattern spine} & \Upsilon_1;\Phi_1;\Psi \vdash s \Leftarrow P \,/\, S : A \\ X \notin \Phi_1 & \Upsilon_1;\Phi_1;\Psi \vdash \mathsf{prune}\, A \Rightarrow (\Upsilon_2 ; \rho) \end{array}}{\Upsilon_1;\Phi_1;\Psi \vdash X\cdot s \Leftarrow P /_\rho (\Upsilon_2 ; [\![\rho]\!]\Phi_1, X : [\![\rho]\!]A)X\cdot S}$$

$\Upsilon_1;\Phi_1;\Psi \vdash S : A \Leftarrow P$ by i.h. (4)
$\Upsilon_1{}^-;\Phi_1{}^-;\Psi^- \vdash s \approx S : A^- \Leftarrow P^-$ by i.h. (4)
$\Upsilon_2 \vdash_{[\![\rho]\!]\Phi_1} \rho \Leftarrow \Upsilon_1$, $\vdash_{[\![\rho]\!]\Phi_1} \Upsilon_2$ mctx and $\Upsilon_2 \vdash [\![\rho]\!]\Phi_1$ fctx by correctness of pruning
$[\![\rho]\!]\Phi_1 \subseteq [\![\rho]\!]\Phi_1, X : [\![\rho]\!]A$ by definition
$\Upsilon_2;[\![\rho]\!]\Phi_1;\cdot \vdash [\![\rho]\!]A \Leftarrow \mathsf{type}$ by correctness of pruning
$\Upsilon_2;[\![\rho]\!]\Phi_1;[\![\rho]\!]\Psi \vdash [\![\rho]\!]S : [\![\rho]\!]A \Leftarrow [\![\rho]\!]P$ by substitution lemma
$\Upsilon_2;[\![\rho]\!]\Phi_1, X{:}[\![\rho]\!]A;[\![\rho]\!]\Psi \vdash [\![\rho]\!]S : [\![\rho]\!]A \Leftarrow [\![\rho]\!]P$ by weakening
$\Upsilon_2;[\![\rho]\!]\Phi_1, X{:}[\![\rho]\!]A;[\![\rho]\!]\Psi \vdash S : [\![\rho]\!]A \Leftarrow [\![\rho]\!]P$ S is a pattern spine and $[\![\rho]\!]S = S$
$([\![\rho]\!]\Phi_1, x{:}[\![\rho]\!]A)(X) = [\![\rho]\!]A$ by definition
$\Upsilon_2;[\![\rho]\!]\Phi_1, X : [\![\rho]\!]A;[\![\rho]\!]\Psi \vdash X\cdot S \Leftarrow [\![\rho]\!]P$ by typing rule
$\Upsilon_1 \vdash \Phi_1$ fctx by assumption
$\Upsilon_2 \vdash [\![\rho]\!]\Phi_1$ fctx by previous line (see correctness of pruning)
$\Upsilon_2 \vdash [\![\rho]\!]\Phi_1, X{:}[\![\rho]\!]A$ fctx by typing rule
$(\Phi_1, X{:}A)^- = ([\![\rho]\!]\Phi_1, X{:}[\![\rho]\!]A)^-$ by erasure definition
$\Upsilon_2{}^-;([\![\rho]\!]\Phi_1, X{:}[\![\rho]\!]A)^-;\Psi^- \vdash s \approx S : [\![\rho]\!]A^- \Leftarrow P^-$ by substitution

and erasure properties

$\Upsilon_2{}^-;(\llbracket\rho\rrbracket\Phi_1, X{:}\llbracket\rho\rrbracket A)^-;\Psi^- \vdash X\cdot s \approx X\cdot S \Leftarrow P^-$ by equivalence relation $\approx$

Case $\mathscr{D} = \dfrac{\Phi_1(X) = A \quad \Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow P\,/_\rho\,(\Upsilon_2\,;\,\Phi_2)S}{\Upsilon_1;\Phi_1;\Psi \vdash X\cdot s \Leftarrow P\,/_\rho\,(\Upsilon_2\,;\,\Phi_2)X\cdot S}$

$\Upsilon_2;\Phi_2;\llbracket\rho\rrbracket\Psi \vdash S : \llbracket\rho\rrbracket A \Leftarrow \llbracket\rho\rrbracket P$ by i.h. (3)
$\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash s \approx S : A^- \Leftarrow P^-$ by i.h. (3)
$\llbracket\rho\rrbracket\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ by i.h. (3)
$\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx by i.h. (3)
$\Phi_2(X) = \llbracket\rho\rrbracket A$ by assumption and previous line
$\Upsilon_2;\Phi_2;\llbracket\rho\rrbracket\Psi \vdash X\cdot S \Leftarrow \llbracket\rho\rrbracket P$ by typing rule
$\Upsilon_2{}^-;\Phi_2{}^-;\Psi^- \vdash X\cdot s \approx X\cdot S \Leftarrow P^-$ by equivalence relation $\approx$

Case $\mathscr{D} = \dfrac{\Upsilon_1;\Phi_1;\Psi, x{:}A \vdash h\cdot(s@((x\cdot \mathsf{nil});\mathsf{nil})) \Leftarrow B\,/_\rho\,(\Upsilon_2\,;\,\Phi_2)M}{\Upsilon_1;\Phi_1;\Psi \vdash h\cdot s \Leftarrow \Pi x{:}A.B\,/_\rho\,(\Upsilon_2\,;\,\Phi_2)\lambda x.M}$

$\Upsilon_2;\Phi_2;\llbracket\rho\rrbracket(\Psi, x{:}A) \vdash M \Leftarrow \llbracket\rho\rrbracket B$ by i.h. (1)
$\Upsilon_2{}^-;\Phi_2{}^-;(\Psi, x{:}A)^- \vdash h\cdot s@((x\cdot\mathsf{nil});\mathsf{nil}) \approx M : B^-$ by i.h. (1)
$\llbracket\rho\rrbracket\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ by i.h. (1)
$\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi} \Upsilon_2$ mctx by i.h. (1)
$\Upsilon_2;\Phi_2;\llbracket\rho\rrbracket\Psi, x{:}\llbracket\rho\rrbracket A \vdash M \Leftarrow \llbracket\rho\rrbracket B$ by definition of substitution
$\Upsilon_2;\Phi_2;\llbracket\rho\rrbracket\Psi \vdash \lambda x.M \Leftarrow \Pi x{:}\llbracket\rho\rrbracket A.\llbracket\rho\rrbracket B$ by typing rule
$\Upsilon_2;\Phi_2;\llbracket\rho\rrbracket\Psi \vdash \lambda x.M \Leftarrow \llbracket\rho\rrbracket(\Pi x{:}A.B)$ by def. of substitution
$\Upsilon_2{}^-;\Phi_2{}^-;(\Psi)^- \vdash h\cdot s \approx \lambda x.M : \Pi x{:}A^-.B^-$ by equivalence relation $\approx$

Case $\mathscr{D} = \dfrac{}{\Upsilon_1;\Phi_1;\Psi \vdash \_ \Leftarrow P\,/_{\mathsf{id}(\Upsilon_1)}\,(\Upsilon_1, u{::}P[\Psi]\,;\,\Phi_1)u[\mathsf{id}(\Psi)]}$

$\Upsilon_1 \vdash \Phi_1$ fctx and $\vdash_{\Phi_1} \Upsilon_1$ mctx by assumptions
$\Upsilon_1;\Phi_1;\Psi \vdash P \Leftarrow \mathsf{type}$ by assumption
$\Upsilon_1;\Phi_1 \vdash \Psi$ ctx by assumptions
$\Upsilon_1;\Phi;\Psi \vdash P \Leftarrow \mathsf{type}$ by assumption
$\Upsilon_1, u{::}P[\Psi];\Phi;\Psi \vdash P \Leftarrow \mathsf{type}$ by weakening
$\vdash_{\Phi} \Upsilon_1, u{::}P[\Psi]$ mctx by typing rules
$\Upsilon_1, u{::}P[\Psi] \vdash_{\Phi_1} \mathsf{id}(\Upsilon_1) \Leftarrow \Upsilon_1$ by typing rules
$\Upsilon_1, u{::}P[\Psi]\,;\,\Phi_1;\Psi \vdash \mathsf{id}(\Psi) \Leftarrow \Psi$ by typing rule
$\Upsilon_1, u{::}P[\Psi]\,;\,\Phi_1;\Psi \vdash u[\mathsf{id}(\Psi)] \Leftarrow [\mathsf{id}(\Psi)]^a_\Psi P$ by typing rule
$P = [\mathsf{id}(\Psi)]^a_\Psi P = ([\mathsf{id}(\Psi)]^a_\Psi P) = \llbracket\mathsf{id}(\Upsilon_1)\rrbracket P$ by definition
$\llbracket\mathsf{id}(\Upsilon_1)\rrbracket\Phi_1 = \Phi_1$ by definition of substitution
$\llbracket\mathsf{id}(\Upsilon_1)\rrbracket\Phi_1 \subseteq \Phi_1$ by definition
$\Upsilon_1, u{::}P[\Psi] \vdash \Phi_1$ fctx by weakening
$(\Upsilon_1, u{::}P[\Psi])^-;\Phi_1{}^-;\Psi^- \vdash \_ \approx u[\mathsf{id}(\Psi)] : P^-$ by equivalence relation $\approx$

Statement (2) : Synthesizing spine

$$\textbf{Case}\ \mathscr{D} = \frac{\Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow P\ /_{\rho}\ (\Upsilon_2\ ;\ \Phi_2)S}{\Upsilon_1;\Phi_1;\Psi \vdash^0 s : A \Leftarrow P\ /_{\rho}\ (\Upsilon_2\ ;\ \Phi_2)S}$$

$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash S : [\![\rho]\!]A \Leftarrow [\![\rho]\!]P$	by i.h. (3)
$[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$	by i.h. (3)
$\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx	by i.h. (3)
$\Upsilon_2^-;\Phi_2^-;\Psi^- \vdash s \approx S : A^- \Leftarrow P^-$	by i.h. (3)
$\Upsilon_2^-;\Phi_2^-;\Psi^- \vdash^0 s \approx S : A^- \Leftarrow P^-$	by equivalence relation $\approx$

$$\textbf{Case}\ \mathscr{D} = \frac{\begin{array}{l}\Upsilon_1;\Phi_1;\Psi \vdash \mathsf{lower}\ A \Rightarrow (M, u{::}Q[\Psi'])\\ \Upsilon_1, u{::}Q[\Psi'];\Phi_1;\Psi \vdash^{i-1} s : [M/x]^a_A(B) \Leftarrow P\ /_{\rho}\ (\Upsilon_2\ ;\ \Phi_2)S \quad \rho = \rho', \widehat{\Psi'}.R/u\end{array}}{\Upsilon_1;\Phi_1;\Psi \vdash^i s : \Pi x{:}A.B \Leftarrow P\ /_{\rho'}\ (\Upsilon_2\ ;\ \Phi_2)([\![\rho]\!]M);S}$$

$\Upsilon_1;\Phi_1 \vdash \Psi'$ ctx and $\Upsilon_1;\Phi_1;\Psi' \vdash Q \Leftarrow$ type	by lowering
$\cdot \vdash_{\Phi_1} \Upsilon_1$ mctx	by assumption
$\cdot \vdash_{\Phi_1} \Upsilon_1, u{::}Q[\Psi']$ ctx	by typing rules
$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash S : [\![\rho]\!]([M/x]^a_A B) \Leftarrow [\![\rho]\!]P$	by i.h. (2)
$[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1, u{::}Q[\Psi']$	by i.h. (2)
$\Upsilon_2^-;\Phi_2^-;\Psi^- \vdash^{i-1} s \approx S : ([M/x]^a_A B)^- \Leftarrow P^-$	by i.h. (2)
$\Upsilon_2^-;\Phi_2^-;\Psi^- \vdash^{i-1} s \approx S : B^- \Leftarrow P^-$	since $B^- = [M/x]^a_A(B)^-$
$\Upsilon_2^-;\Phi_2^-;\Psi^- \vdash^{i} s \approx ([\![\rho]\!]M);S : (\Pi x{:}A.B)^- \Leftarrow P^-$	by equivalence relation $\approx$
$\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx	by i.h. (2)
$\Upsilon_2 \vdash_{\Phi_2} \rho', \widehat{\Psi'}.R/u \Leftarrow \Upsilon_1, u{::}Q[\Psi']$	since $\rho = \rho', \widehat{\Psi'}.R/u$
$\Upsilon_2 \vdash_{\Phi_2} \rho' \Leftarrow \Upsilon_1$	by inversion
$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash S : [[\![\rho]\!]M/x]^a_A([\![\rho]\!]B) \Leftarrow [\![\rho]\!]P$	by definition of substitution
$\Upsilon_1;\Phi_1 \vdash \Psi$ ctx and	
$\Upsilon_1;\Phi_1;\Psi \vdash \Pi x{:}A.B \Leftarrow$ type and	
$\Upsilon_1;\Phi_1;\Psi \vdash P$ type	by assumption
$\Pi\Psi.A = \Pi\Psi'.Q$ and $\Upsilon_1, u{::}Q[\Psi'];\Phi_1;\Psi \vdash M \Leftarrow A$	by lemma lowering
$\Upsilon_2;\Phi_2;[\![\rho']\!]\Psi \vdash S : [[\![\rho]\!]M/x]^a_A([\![\rho']\!]B) \Leftarrow [\![\rho']\!]P$	by strengthening
$\Upsilon_2;\Phi_2;[\![\rho]\!]\Psi \vdash [\![\rho]\!]M \Leftarrow [\![\rho]\!]A$	by subst. lemma
$\Upsilon_2;\Phi_2;[\![\rho']\!]\Psi \vdash [\![\rho]\!]M \Leftarrow [\![\rho']\!]A$	by strengthening
$\Upsilon_2;\Phi_2;[\![\rho']\!]\Psi \vdash [\![\rho]\!]M;S : \Pi x{:}[\![\rho']\!]A.[\![\rho']\!]B) \Leftarrow [\![\rho']\!]P$	by typing rule
$\Upsilon_2;\Phi_2;[\![\rho']\!]\Psi \vdash [\![\rho]\!]M;S : [\![\rho']\!](\Pi x{:}A.B) \Leftarrow [\![\rho']\!]P$	by def. of substitution

Statement (3) : Checking spine

$$\textbf{Case}\ \mathscr{D} = \frac{\begin{array}{l}\Upsilon_1;\Phi_1;\Psi \vdash m \Leftarrow A\ /_{\rho_1}\ (\Upsilon_2\ ;\ \Phi_2)M\\ \Upsilon_2;\Phi_2;[\![\rho_1]\!]\Psi \vdash s : [M/x]^a_A([\![\rho_1]\!]B) \Leftarrow [\![\rho_1]\!]P\ /_{\rho_2}\ (\Upsilon_3\ ;\ \Phi_3)S \quad \rho = [\![\rho_2]\!]\rho_1\end{array}}{\Upsilon_1;\Phi_1;\Psi \vdash s : \Pi x{:}A.B \Leftarrow P\ /_{\rho}\ (\Upsilon_3\ ;\ \Phi_3)[\![\rho_2]\!]M;S}$$

$\Upsilon_2;\Phi_2;[\![\rho_1]\!]\Psi \vdash M \Leftarrow [\![\rho_1]\!]A$	by i.h. (1)
$\Upsilon_2^-;\Phi_2^-;\Psi^- \vdash m \approx M : A^-$	by i.h. (1)
$[\![\rho_1]\!]\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho_1 \Leftarrow \Upsilon_1$	by i.h. (1)

$\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx — by i.h. (1)
$\Upsilon_3; \Phi_3; [\![\rho]\!]\Psi \vdash S : [\![\rho_2]\!]([M/x]^a_A [\![\rho_1]\!]B) \Leftarrow [\![\rho]\!]P$ — by i.h. (3)
$\Upsilon_3{}^-; \Phi_3{}^-; ([\![\rho_1]\!]\Psi)^- \vdash s \approx S : ([M/x]^a_A [\![\rho_1]\!]B)^- \Leftarrow ([\![\rho_1]\!]P)^-$ — by i.h. (3)
$\Upsilon_3{}^-; \Phi_3{}^-; \Psi^- \vdash s \approx S : B^- \Leftarrow P^-$ — since $([M/x]^a_A [\![\rho_1]\!]B)^- = B^-$ and $([\![\rho_1]\!]P)^- = P^-$ and $([\![\rho_1]\!]\Psi)^- = \Psi^-$
$\Upsilon_3{}^-; \Phi_3{}^-; \Psi^- \vdash m \approx M : A^-$ — by substitution and erasure property
$\Upsilon_3{}^-; \Phi_3{}^-; \Psi^- \vdash (m;s) \approx (M;S) : \Pi x{:}A.B^- \Leftarrow P^-$ — by equivalence relation $\approx$
$[\![\rho_2]\!]\Phi_2 \subseteq \Phi_3$ and $\Upsilon_3 \vdash_{\Phi_3} \rho_2 \Leftarrow \Upsilon_2$ — by i.h. (3)
$\Upsilon_3 \vdash \Phi_3$ fctx and $\vdash_{\Phi_3} \Upsilon_3$ mctx — by i.h. (3)
$\Upsilon_3; \Phi_3; [\![\rho]\!]\Psi \vdash [\![\rho_2]\!]M \Leftarrow [\![\rho]\!]A$ — by substitution lemma and subst. def.
$[\![\rho_2]\!]([M/x]^a_A([\![\rho_1]\!]B)) = [[\![\rho_2]\!]M/x]^a_A([\![\rho]\!]B)$ — by substitution definition
$\Upsilon_3; \Phi_3; [\![\rho]\!]\Psi \vdash S : ([[\![\rho_2]\!]M/x]^a_A [\![\rho]\!]B) \Leftarrow [\![\rho]\!]P$ — by previous line
$\Upsilon_3; \Phi_3; [\![\rho]\!]\Psi \vdash [\![\rho_2]\!]M; S : \Pi x{:}[\![\rho]\!]A.[\![\rho]\!]B \Leftarrow [\![\rho]\!]P$ — by typing rule
$\Upsilon_3; \Phi_3; [\![\rho]\!]\Psi \vdash [\![\rho_2]\!]M; S : [\![\rho]\!]\Pi x{:}A.B \Leftarrow [\![\rho]\!]P$ — by substitution definition

Case $\mathscr{D} = \dfrac{\Upsilon_1; \Phi_1; \Psi \vdash a \cdot S' \doteq a \cdot S / (\rho\ ,\ \Upsilon_2)}{\Upsilon_1; \Phi_1; \Psi \vdash \mathsf{nil} : a \cdot S' \Leftarrow a \cdot S\ /_\rho\ (\Upsilon_2\ ;\ [\![\rho]\!]\Phi_1)\mathsf{nil}}$

$[\![\rho]\!](a \cdot S) = [\![\rho]\!](a \cdot S')$ — by correctness of HOP unification
$\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ where $\Phi_2 = [\![\rho]\!]\Phi_1$ — by correctness of HOP unification
$\Upsilon_2 \vdash \Phi_2$ fctx — by correctness of HOP unification
$\vdash_{\Phi_2} \Upsilon_2$ mctx — by correctness of HOP unification
$[\![\rho]\!]\Phi_1 \subseteq [\![\rho]\!]\Phi_1$ — by definition
$\Upsilon_2; [\![\rho]\!]\Phi_1; [\![\rho]\!]\Psi \vdash \mathsf{nil} : [\![\rho]\!](a \cdot S') \Leftarrow [\![\rho]\!](a \cdot S)$ — by typing rule
$(\mathbf{a} \cdot S')^- = (\mathbf{a} \cdot S)^- = \mathbf{a}$ — by definition of erasure
$\Upsilon_2{}^-; \Phi_1{}^-; \Psi^- \vdash \mathsf{nil} \approx \mathsf{nil} : (\mathbf{a} \cdot S')^- \Leftarrow (\mathbf{a} \cdot S)^-$ — by equivalence relation $\approx$

Statement (4) : Synthesize type from spine

Case $\mathscr{D} = \dfrac{}{\Upsilon_1; \Phi_1; \Psi \vdash \mathsf{nil} \Leftarrow P\ /\ \mathsf{nil} : P}$

$\Upsilon_1; \Phi_1; \Psi \vdash \mathsf{nil} : P \Leftarrow P$ — by typing rule
$\Upsilon_1; \Phi_1; \Psi \vdash P \Leftarrow \mathsf{type}$ — by assumption
$\Upsilon_1{}^-; \Phi_1{}^-; \Psi^- \vdash \mathsf{nil} \approx \mathsf{nil} : P^- \Leftarrow P^-$ — by equivalence relation $\approx$

Case $\mathscr{D} = \dfrac{\begin{array}{lcll} \eta\mathsf{con}(m) & = & x & \Psi(x) = A \\ \eta\mathsf{exp}_A(x) & = & M & \Psi^- \vdash m \Leftarrow A^- \end{array} \qquad \Upsilon_1; \Phi_1; \Psi \vdash s \Leftarrow P\ /\ S : B \quad B = [x/y]B'}{\Upsilon_1; \Phi_1; \Psi \vdash m; s \Leftarrow P\ /\ M; S : \Pi y : A.B'}$

$\Upsilon_2; \Phi_2; \Psi \vdash S : B \Leftarrow P$ — by i.h. (4)
$\Upsilon_2{}^-; \Phi_2{}^-; \Psi^- \vdash s \approx S : B^- \Leftarrow P^-$ — by i.h. (4)
$\Upsilon_2; \Phi_2; \Psi \vdash B \Leftarrow \mathsf{type}$ — by i.h. (4)
$\Psi(x) = A$ — by assumption
$\Upsilon_1; \Phi_1; \Psi \vdash M \Leftarrow A$ — by eta-expansion lemma 4.5

$([x/y]^a_A(B') = [M/y]^a_A(B')$ by eta-expansion lemma 4.4
$\Upsilon_1;\Phi_1;\Psi \vdash S : [M/x]^a_A(B') \Leftarrow P$ by previous lines
$\Upsilon_1;\Phi_1;\Psi \vdash M;S : \Pi y{:}A.B' \Leftarrow P$ by typing rule
$\Upsilon_1;\Phi_1;\Psi, y{:}A \vdash B' \Leftarrow \mathsf{type}$ by inverting subst.
$\Upsilon_1;\Phi_1;\Psi \vdash \Pi y{:}A.B' \Leftarrow \mathsf{type}$ by kinding rules
$\Upsilon_1{}^-;\Phi_1{}^-;\Psi^- \vdash m \approx M : A^-$ by lemma 5.1
$\Upsilon_1{}^-;\Phi_1{}^-;\Psi^- \vdash m;s \approx M;S : (\Pi x{:}A.B)^- \Leftarrow P^-$ by equivalence relation $\approx$
$\square$

Theorem 6.3 (*Completeness of type reconstruction*)

1. If $\Psi^- \vdash m \approx M : A^-$ and $\Upsilon \vdash_{[\![\rho_0]\!]\Phi} \rho_0 \Leftarrow \Upsilon_1$
   and $\Upsilon;[\![\rho_0]\!]\Phi;[\![\rho_0]\!]\Psi \vdash M \Leftarrow [\![\rho_0]\!]A$ then
   $\Upsilon_1;\Phi;\Psi \vdash m \Leftarrow A /_\rho (\Upsilon_2 ; \Phi')M'$ and there exists a contextual substitution θ s.t. $[\![\theta]\!]\rho = \rho_0$ and $\Upsilon \vdash \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]M' = M$, $[\![\rho]\!]\Phi = \Phi'$.
2. If $\Psi^- \vdash s \approx S : A^- \Leftarrow P^-$ and $\Upsilon \vdash_{[\![\rho_0]\!]\Phi} \rho_0 \Leftarrow \Upsilon_1$
   and $\Upsilon;[\![\rho_0]\!]\Phi;[\![\rho_0]\!]\Psi \vdash S : [\![\rho_0]\!]A \Leftarrow [\![\rho_0]\!]P$ then
   $\Upsilon_1;\Phi;\Psi \vdash s : A \Leftarrow P /_\rho (\Upsilon_2 ; \Phi')S'$ and there exists a contextual substitution θ s.t. $[\![\theta]\!]\rho = \rho_0$ and $\Upsilon \vdash \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]S' = S$, and $[\![\rho]\!]\Phi = \Phi'$.
3. If $\Psi^- \vdash^i s \approx S : A^- \Leftarrow P^-$ and $\Upsilon \vdash_{[\![\rho_0]\!]\Phi} \rho_0 \Leftarrow \Upsilon_1$
   and $\Upsilon;[\![\rho_0]\!]\Phi;[\![\rho_0]\!]\Psi \vdash S : [\![\rho_0]\!]A \Leftarrow [\![\rho_0]\!]P$ then
   $\Upsilon_1;\Phi;\Psi \vdash^i s : A \Leftarrow P /_\rho (\Upsilon_2 ; \Phi')S'$ and there exists a contextual substitution θ s.t. $[\![\theta]\!]\rho = \rho_0$ and $\Upsilon \vdash \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]S' = S$, $[\![\rho]\!]\Phi = \Phi'$.

Proof
Induction on the first derivation.

Case: $\mathscr{D} = \dfrac{\Psi^-, x{:}A^- \vdash m \approx M : B^-}{\Psi^- \vdash \lambda x.m \approx \lambda x.M : (\Pi x{:}A.B)^-}$

$\Upsilon \vdash_{[\![\rho_0]\!]\Phi} \rho_0 \Leftarrow \Upsilon_1$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi;[\![\rho_0]\!]\Psi \vdash \lambda x.M \Leftarrow [\![\rho_0]\!](\Pi x{:}A.B)$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi;[\![\rho_0]\!]\Psi, x{:}[\![\rho_0]\!]A \vdash M \Leftarrow [\![\rho_0]\!]B$ by substitution and inversion
$\Upsilon_1;\Phi;\Psi, x{:}A \vdash m \Leftarrow B /_\rho (\Upsilon_2 ; \Phi')M'$ by i.h. (1)
$\Upsilon \vdash_{[\![\rho_0]\!]\Phi} \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]M' = M$ and $[\![\theta]\!]\rho = \rho_0$ and $[\![\rho]\!]\Phi = \Phi'$ by i.h. (1)
$\Upsilon_1;\Phi;\Psi \vdash \lambda x.m \Leftarrow \Pi x{:}A.B /_\rho (\Upsilon_2 ; \Phi_2)\lambda x.M'$ by rules
$\lambda x.[\![\theta]\!]M' = \lambda x.M$ by equality
$[\![\theta]\!]\lambda x.M' = \lambda x.M$ by substitution

Case $\mathscr{D} = \dfrac{\Psi^-, x{:}A^- \vdash h \cdot s@((x \cdot \mathsf{nil});\mathsf{nil}) \approx M : B^-}{\psi \vdash h \cdot s \approx \lambda x.M : (\Pi x{:}A.B)^-}$

$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \rho_0 \Leftarrow \Upsilon_1$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi_1;[\![\rho_0]\!]\Psi \vdash \lambda x.M \Leftarrow [\![\rho_0]\!]\Pi x{:}A.B$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi;[\![\rho_0]\!]\Psi, x{:}[\![\rho_0]\!]A \vdash M \Leftarrow [\![\rho_0]\!]B$ by substitution and inversion
$\Upsilon_1;\Phi_1;\Psi, x{:}A \vdash h \cdot s@((x \cdot \mathsf{nil});\mathsf{nil}) \Leftarrow B /_\rho (\Upsilon_2 ; \Phi_2)M'$ by i.h. (1)
$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]M' = M$, $[\![\theta]\!]\rho = \rho_0$ and $[\![\rho]\!]\Phi_1 = \Phi_2$ by i.h. (1)

$\Upsilon_1;\Phi_1;\Psi \vdash h \cdot s \Leftarrow \Pi x{:}A.B \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)\lambda x.M$ by rules
$\lambda x.[\![\theta]\!]M' = \lambda x.M$ by equality
$[\![\theta]\!]\lambda x.M' = \lambda x.M$ by substitution

$$\textbf{Case}\ \mathscr{D} = \frac{\Sigma(c) = (A,i) \quad \Psi^- \vdash^i s \approx S : A^- \Leftarrow P^-}{\Psi^- \vdash \mathbf{c}\cdot s \approx \mathbf{c}\cdot S : P^-}$$

$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \rho_0 \Leftarrow \Upsilon_1$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi_1;[\![\rho_0]\!]\Psi \vdash \mathbf{c}\cdot S \Leftarrow [\![\rho_0]\!]P$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi_1;[\![\rho_0]\!]\Psi \vdash S : A \Leftarrow [\![\rho_0]\!]P$ by inversion
$A = [\![\rho_0]\!]A$ since A is closed
$\Upsilon_1;\Phi_1;\Psi \vdash^i s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S'$ by i.h. (2)
$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]S' = S$, and $[\![\theta]\!]\rho = \rho_0$ and $[\![\rho]\!]\Phi_1 = \Phi_2$ by i.h.
$\Upsilon_1;\Phi_1;\Psi \vdash \mathbf{c}\cdot s \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)\mathbf{c}\cdot S'$ by rule
$\mathbf{c}\cdot [\![theta]\!]S' = \mathbf{c}\cdot S$ by equality
$[\![\theta]\!](\mathbf{c}\cdot S') = \mathbf{c}\cdot S$ by substitution

$$\textbf{Case}\ \mathscr{D} = \frac{\Psi(x) = A^- \quad \Psi^- \vdash s \approx S : A^- \Leftarrow P^-}{\Psi^- \vdash x\cdot s \approx x\cdot S : P^-}$$

$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \rho_0 \Leftarrow \Upsilon_1$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi_1;[\![\rho_0]\!]\Psi \vdash x\cdot S \Leftarrow [\![\rho_0]\!]P$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi_1;[\![\rho_0]\!]\Psi \vdash S : [\![\rho_0]\!]A \Leftarrow [\![\rho_0]\!]P$ by inversion
$\Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S'$ by i.h. (2)
$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]S' = S$, and $[\![\theta]\!]\rho = \rho_0$ and $[\![\rho]\!]\Phi_1 = \Phi_2$ by i.h. (2)
$\Psi(x) = A$ since $([\![\rho_0]\!]\Psi)(x) = [\![\rho_0]\!]A$
$\Upsilon_1;\Phi_1;\Psi \vdash x\cdot s \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)x\cdot S'$ by rule
$x\cdot [\![theta]\!]S' = \mathbf{c}\cdot S$ by equality
$[\![\theta]\!](x\cdot S') = \mathbf{c}\cdot S$ by substitution

$$\textbf{Case}\ \mathscr{D} = \frac{([\![\rho_0]\!]\Phi_1)(X) = [\![\rho_0]\!]A \quad \Psi^- \vdash s \approx S : ([\![\rho_0]\!]A)^- \Leftarrow P^-}{\Psi^- \vdash X\cdot s \approx X\cdot S : P^-}$$

$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \rho_0 \Leftarrow \Upsilon_1$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi_1;[\![\rho_0]\!]\Psi \vdash X\cdot S \Leftarrow [\![\rho_0]\!]P$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi_1;[\![\rho_0]\!]\Psi \vdash S : [\![\rho_0]\!]A \Leftarrow [\![\rho_0]\!]P$ by inversion
$A^- = ([\![\rho_0]\!]A)^-$ by erasure
$\Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S'$ by i.h. (2)
$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]S' = S$, $[\![\theta]\!]\rho = \rho_0$ and $[\![\rho]\!]\Phi_1 = \Phi_2$ by i.h. (2)
$\Phi_1(X) = A$ since $([\![\rho_0]\!]\Phi_1)(X) = [\![\rho_0]\!]A$
$\Upsilon_1;\Phi_1;\Psi \vdash X\cdot s \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)X\cdot S'$ by rule
$X\cdot [\![theta]\!]S' = X\cdot S$ by equality

$[\![\theta]\!](X \cdot S') = X \cdot S$ by substitution

Case $\mathscr{D} = \dfrac{}{\Psi^- \vdash \_ \approx R : P^-}$

$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \rho_0 \Leftarrow \Upsilon_1$ by assumption
$\Upsilon; [\![\rho_0]\!]\Phi_1; [\![\rho_0]\!]\Psi \vdash R \Leftarrow [\![\rho_0]\!]P$ by assumption
$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \rho_0, \hat{\Psi}.R/u \Leftarrow \Upsilon_1, u{::}P[\Psi]$ by typing rule
$\Upsilon_1, u{::}P[\Phi] \vdash_{\Phi_1} \mathsf{id}(\Upsilon_1) \Leftarrow \Upsilon_1$ by definition
let θ be $\rho_0, \hat{\Psi}.R/u$.
$[\![\rho_0, \hat{\Psi}.R/u]\!](u[\mathsf{id}(\Psi)]) = R$ by definition
$[\![\mathsf{id}(\Upsilon_1)]\!]\Phi = \Phi$ by definition
$[\![\rho_0, \hat{\Psi}.R/u]\!]\mathsf{id}(\Upsilon_1) = \rho_0$ by definition
$\Upsilon_1; \Phi_1; \Psi \vdash \_ \Leftarrow P \,/_{\mathsf{id}(\Upsilon_1)}\, (\Upsilon_1, u{::}P[\Psi]\ ;\ \Phi_1)u[\mathsf{id}(\Psi)]$ by rule

Case $\mathscr{D} = \dfrac{}{\Psi^- \vdash \mathsf{nil} \approx \mathsf{nil} : (\mathbf{a} \cdot S)^- \Leftarrow (\mathbf{a} \cdot S')^-}$

$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \rho_0 \Leftarrow \Upsilon_1$ by assumption
$\Upsilon; [\![\rho_0]\!]\Phi_1; [\![\rho_0]\!]\Psi \vdash \mathsf{nil} : [\![\rho_0]\!](\mathbf{a} \cdot S) \Leftarrow [\![\rho_0]\!](\mathbf{a} \cdot S')$ by assumption
$[\![\rho_0]\!](\mathbf{a} \cdot S) = [\![\rho_0]\!](\mathbf{a} \cdot S')$ by typing inversion
let $P = \mathbf{a} \cdot S'$ and $Q = \mathbf{a} \cdot S$.
$\Upsilon_1; \Phi_1; \Psi \vdash Q \doteq P/(\rho\ ,\ \Upsilon_2)$ and there exists a θ s.t. $[\![\theta]\!]\rho = \rho_0$
and $\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \theta \Leftarrow \Upsilon_2$ and $\Upsilon_2 \vdash \rho \Leftarrow \Upsilon_1$ completeness of hop
$\Upsilon_1; \Phi; \Psi \vdash \mathsf{nil} : Q \Leftarrow P \,/_{\rho}\, (\Upsilon_2\ ;\ \Phi_2)\mathsf{nil}$

Case $\mathscr{D} = \dfrac{\Psi^- \vdash m \approx M : A^- \quad \Psi \vdash s \approx S : B^- \Leftarrow P^-}{\Psi^- \vdash (m;s) \approx (M;S) : (\Pi x{:}A.B)^- \Leftarrow P^-}$

$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \rho_0 \Leftarrow \Upsilon_1$ by assumption
$\Upsilon; [\![\rho_0]\!]\Phi_1; [\![\rho_0]\!]\Psi \vdash M;S : [\![\rho_0]\!](\Pi x{:}A.B) \Leftarrow [\![\rho_0]\!]P$ by assumption
$\Upsilon; [\![\rho_0]\!]\Phi_1; [\![\rho_0]\!]\Psi \vdash M \Leftarrow [\![\rho_0]\!]A$
$\Upsilon; [\![\rho_0]\!]\Phi_1; [\![\rho_0]\!]\Psi \vdash S : [M/x]^a_A[\![\rho_0]\!]B \Leftarrow [\![\rho_0]\!]P$ by inversion
$\Upsilon_1; \Phi_1; \Psi \vdash m \Leftarrow A \,/_{\rho_1}\, (\Upsilon_2\ ;\ \Phi_2)M$ by i.h. (1)
there exists some θ s.t. $\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]M' = M$ and $[\![\theta]\!]\rho_1 = \rho_0$ by i.h. (1)
$[\![\rho_1]\!]\Phi_1 = \Phi_2$ by i.h. (1)
$[\![\rho_0]\!]\Phi_1 = [\![\theta]\!][\![\rho_1]\!]\Phi_1 = [\![\theta]\!]\Phi_2$ by previous lines
$B^- = ([M/x]^a_A[\![\rho]\!]B)^- = ([\![\rho]\!]B)^-$ by erasure
$\Upsilon; [\![\theta]\!]\Phi_2; [\![\theta]\!]([\![\rho_1]\!]\Psi) \vdash S : [\![\theta]\!]([M'/x]^a_A([\![\rho_1]\!]B)) \Leftarrow [\![\theta]\!]([\![\rho_1]\!]P)$ by $\rho_0 = [\![\theta]\!]\rho_1$

$\Upsilon_2; \Phi_2; [\![\rho_1]\!]\Psi \vdash s : [M'/x]^a_A([\![\rho_1]\!]B) \Leftarrow [\![\rho_1]\!]P \,/_{\rho_2}\, (\Upsilon_3\ ;\ \Phi_3)S'$ by i.h. (2)
there exists some θ' s.t. $\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \theta' \Leftarrow \Upsilon_3$ and $[\![\theta']\!]S' = S$ and $[\![\rho_2]\!]\Phi_2 = \Phi_3$
and $\theta = [\![\theta']\!]\rho_2$, and therefore $M = [\![\theta]\!]M' = [\![\theta']\!][\![\rho_2]\!]M'$

Let $\rho = [\![\rho_2]\!]\rho_1$ and

$\Upsilon_1;\Phi_1;\Psi \vdash m;s : \Pi x{:}A.B \Leftarrow P \,/_\rho\, (\Upsilon_3\,;\,\Phi_3)([\![\rho_2]\!]M';S')$ by rule
and θ' is a contextual substitution s.t. $\rho_0 = [\![\theta']\!]\rho$
$\Phi_3 = [\![\rho_2]\!]\Phi_2 = [\![\rho_2]\!][\![\rho_1]\!]\Phi_1 = [\![\rho]\!]\Phi_1$, and $[\![\theta']\!]([\![\rho_2]\!]M';S') = (M;S)$ by equality

Case $\mathscr{D} = \dfrac{\Psi^- \vdash s \approx S : A^- \Leftarrow P^-}{\Psi^- \vdash^0 s \approx S : A^- \Leftarrow P^-}$

$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \rho_0 \Leftarrow \Upsilon_1$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi_1;[\![\rho_0]\!]\Psi \vdash S : [\![\rho_0]\!]A \Leftarrow [\![\rho_0]\!]P$ by assumption
$\Upsilon_1;\Phi_1;\Psi \vdash s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S'$ by i.h.
there exists some θ s.t. $\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]\rho = \rho_0$ and $[\![\theta]\!]S' = S$ by i.h.
$\Upsilon_1;\Phi_1;\Psi \vdash^0 s : A \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S'$ by rule

Case $\mathscr{D} = \dfrac{\Psi \vdash^{i-1} s \approx S : B^- \Leftarrow P^-}{\Psi^- \vdash^i s \approx (M;S) : (\Pi x{:}A.B) \Leftarrow P^-}$

$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \rho_0 \Leftarrow \Upsilon_1$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi_1;[\![\rho_0]\!]\Psi \vdash M;S : [\![\rho_0]\!](\Pi x{:}A.B) \Leftarrow [\![\rho_0]\!]P$ by assumption
$\Upsilon;[\![\rho_0]\!]\Phi_1;[\![\rho_0]\!]\Psi \vdash S : [M/x]^a_A[\![\rho_0]\!]B \Leftarrow [\![\rho_0]\!]P$ by inversion
$\Upsilon;[\![\rho_0]\!]\Phi_1;[\![\rho_0]\!]\Psi \vdash M \Leftarrow [\![\rho_0]\!]A$ by inversion
W.l.g. let $A = \Pi\Psi_0.Q_0$ then $M = \lambda\hat{\Psi}_0.R$ and
$\Upsilon;[\![\rho_0]\!]\Phi_1;[\![\rho_0]\!]\Psi,[\![\rho_0]\!]\Psi_0 \vdash R \Leftarrow [\![\rho_0]\!]Q_0$ by inversion
Let $\Psi_1 = \Psi,\Psi_0$.
$\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \rho_0,\hat{\Psi}_1.R/u \Leftarrow \Upsilon_1,u{::}Q_0[\Psi_1]$ and $\rho'_0 = \rho_0,\hat{\Psi}_1.R/u$ by typing rules

$\Upsilon;[\![\rho'_0]\!]\Phi_1;[\![\rho'_0]\!]\Psi \vdash S : [\lambda\hat{\Psi}_0.R/x]^a_A[\![\rho_0]\!]B \Leftarrow [\![\rho'_0]\!]P$ by previous lines
$[\lambda\hat{\Psi}_0.R/x]^a_A[\![\rho_0]\!]B = [\![\rho'_0]\!]([\lambda\hat{\Psi}_0.u[\mathsf{id}(\Psi_1)]/x]^a_A B)$ by substitution
$\Upsilon;[\![\rho'_0]\!]\Phi_1;[\![\rho'_0]\!]\Psi \vdash S : [\![\rho'_0]\!]([\lambda\hat{\Psi}_0.u[\mathsf{id}(\Psi_1)]/x]^a_A B) \Leftarrow [\![\rho'_0]\!]P$ by previous lines

$\Upsilon_1,u{::}Q_0[\Psi_1];\Phi_1;\Psi \vdash^{i-1} s : [\lambda\hat{\Psi}_0.u[\mathsf{id}(\Psi_1)]/x]^a_A B \Leftarrow P \,/_\rho\, (\Upsilon_2\,;\,\Phi_2)S'$
there exists θ s.t. $\Upsilon \vdash_{[\![\rho_0]\!]\Phi_1} \theta \Leftarrow \Upsilon_2$ and $[\![\theta]\!]\rho = \rho'_0$ and $[\![\rho]\!]\Phi_1 = \Phi_2$ by i.h.
and $[\![\theta]\!]S' = S$ by i.h.
$[\![\theta]\!]\rho = \rho'_0 = \rho_0,\hat{\Psi}_1.R/u$ by previous lines
$[\![\theta]\!]\rho = [\![\theta]\!]\rho',\hat{\Psi}_1.[\![\theta]\!]R'/u$ by equality and previous lines
$\Upsilon_2 \vdash_{[\![\rho]\!]\Phi_1} \rho \Leftarrow \Upsilon_1,u{::}Q_0[\Psi_1]$ by invariant
$\Upsilon_2 \vdash_{[\![\rho]\!]\Phi_1} \rho \Leftarrow \Upsilon_1$ by typing inversion
$[\![\theta]\!]\rho = [\![\theta]\!]\rho',[\![\theta]\!](\hat{\Psi}_1.R')/u = \rho_0,\hat{\Psi}_1.R/u$ and $[\![\theta]\!]R' = R$
$[\![\theta]\!][\![\rho]\!](\lambda\hat{\Psi}_0.u[\mathsf{id}(\Psi_1)]) = \lambda\hat{\Psi}_0.R = M$ recall by previous lines
$\Upsilon_1;\Phi_1;\Psi \vdash \mathsf{lower}\, A \Rightarrow (Q_0[\Psi_1],\lambda\hat{\Psi}_0.u[\mathsf{id}(\Psi_1)])$ by definition of lowering
$\Upsilon_1;\Phi_1;\Psi \vdash^i s : \Pi x{:}A.B \Leftarrow P \,/_{\rho'}\, (\Upsilon_2\,;\,\Phi_2)([\![\rho]\!](\lambda\hat{\Psi}_0.u[\mathsf{id}(\Psi_1)]);S')$ by rule
$[\![\theta]\!]([\![\rho]\!](\lambda\hat{\Psi}_0.u[\mathsf{id}(\Psi_1)]);S') = M;S$ by substitution and equality rules

□

A.5 Soundness proof for abstraction

Before showing the soundness proof for abstraction, we give the proof that well-typed substitutions relate to well-typed spines.

Lemma 7.1 (*Substitutions relate to spines*)
If $\Psi \vdash \sigma \Leftarrow \Psi_0$ and $\Psi \vdash S : [\sigma]^a_{\Psi_0} A \Leftarrow P$ and $\mathsf{toSpine}_{\psi_0}\ \sigma\ S = S'$ where $\psi_0 = (\Psi_0)^-$
then $\Psi \vdash S' : \Pi\Psi_0.A \Leftarrow P$.

Proof
Proof by induction on the first derivation.

Case $\mathscr{D} = \dfrac{}{\Psi \vdash \cdot \Leftarrow \cdot}$

$\mathsf{toSpine}_{\cdot}\ \cdot\ S = S$ by definition
$\Psi \vdash S : A \Leftarrow P$ by assumption

Case $\mathscr{D} = \dfrac{\Psi \vdash \sigma \Leftarrow \Psi_0 \quad \Psi \vdash M \Leftarrow [\sigma]^a_{\Psi_0}(B)}{\Psi \vdash \sigma, M \Leftarrow \Psi_0, x{:}B}$

$\Psi \vdash S : [\sigma, M]^a_{\Psi_0, x:B} A \Leftarrow P$ by assumption
$\mathsf{toSpine}_{(\psi_0, x:\beta)}\ (\sigma, M)\ S = S'$ by assumption
$\mathsf{toSpine}_{\psi_0}\ \sigma\ (M; S) = S'$ by definition
$\Psi \vdash M; S : \Pi x{:}[\sigma]^a_{\Psi_0} B.[\sigma; x]^a_{\Psi_0, x:B} A \Leftarrow P$ by typing rules
$\Psi \vdash M; S : [\sigma]^a_{\Psi_0}(\Pi x{:}B.A) \Leftarrow P$ by substitution properties
$\Psi \vdash S' : \Pi\Psi_0 \Pi x{:}B.A \Leftarrow P$ by i.h.
□

Theorem 7.2 (*Soundness of abstraction*)
Let $\vdash \Psi_0, \Psi\ \mathsf{ctx}$.

1. If $(\Psi_0)\Psi \vdash M \Leftarrow B \,/\, (\Psi_1)N$ and $\Psi_0, \Psi \vdash B \Leftarrow \mathsf{type}$ then
   $\Psi_1, \Psi \vdash N \Leftarrow B$ and $\Psi_0 \subseteq \Psi_1$ and $\vdash \Psi_1\ \mathsf{ctx}$.
2. If $(\Psi_0)\Psi \vdash S : A \Leftarrow P \,/\, (\Psi_1)S'$
   and $\Psi_0, \Psi \vdash A \Leftarrow \mathsf{type}$ and $\Psi_0, \Psi \vdash P \Leftarrow \mathsf{type}$ then
   $\Psi_1, \Psi \vdash S' : A \Leftarrow P$ and $\Psi_0 \subseteq \Psi_1$ and $\vdash \Psi_1\ \mathsf{ctx}$.
3. If $(\Psi_0)\Psi \vdash \sigma \Leftarrow \Psi' \,/\, (\Psi_1)\sigma'$ and $\vdash \Psi'\ \mathsf{ctx}$ then
   $\Psi_1, \Psi \vdash \sigma' \Leftarrow \Psi'$ and $\Psi_0 \subseteq \Psi_1$ and $\vdash \Psi_1\ \mathsf{ctx}$.

Proof
By induction on the abstraction judgment.

Case $\mathscr{D} = \dfrac{(\Psi_0)\Psi, x{:}A \vdash M \Leftarrow B \,/\, (\Psi_1)N}{(\Psi_0)\Psi \vdash \lambda x.M \Leftarrow \Pi x{:}A.B \,/\, (\Psi_1)\lambda x.N}$

$\Psi_1, \Psi, x{:}A \vdash N \Leftarrow B$ and $\Psi_0 \subseteq \Psi_1$ by i.h. (1)
$\Psi_1, \Psi \vdash N \Leftarrow \Pi x{:}A.B$ by typing rule

$$\textbf{Case}\ \mathscr{D} = \frac{\Sigma(\mathbf{c}) = (A, \_) \quad (\Psi_0)\Psi \vdash S : A \Leftarrow P \ /\ (\Psi_1)S'}{(\Psi_0)\Psi \vdash \mathbf{c}\cdot S \Leftarrow P \ /\ (\Psi_1)\mathbf{c}\cdot S'}$$

$\Psi_1, \Psi \vdash S' : A \Leftarrow P$ and $\Psi_0 \subseteq \Psi_1$ by i.h. (2)
$\Psi_1, \Psi \vdash \mathbf{c}\cdot S' \Leftarrow P$ by typing rule

$$\textbf{Case}\ \mathscr{D} = \frac{X{\sim}x{:}A \in \Psi_0 \quad (\Psi_0)\Psi \vdash S : A \Leftarrow P \ /\ (\Psi_1)S'}{(\Psi_0)\Psi \vdash X\cdot S \Leftarrow P \ /\ (\Psi_1)x\cdot S'}$$

$\Psi_1, \Psi \vdash S' : A \Leftarrow P$ and $\Psi_0 \subseteq \Psi_1$ by i.h. (2)
$X{\sim}x{:}A \in \Psi_1$ by previous line
$\Psi_1, \Psi \vdash x\cdot S' \Leftarrow P$ by typing rule

$$\textbf{Case}\ \mathscr{D} = \frac{\begin{array}{llllll} X \notin \Psi_0 \quad \Phi(X) = A & (\Psi_0, X{\sim}x{:}\_) & \cdot & \vdash A \Leftarrow \mathsf{type} & /\ (\Psi_1)A' \\ & (\Psi'_1, X{\sim}x{:}A') & \Psi & \vdash S : A' \Leftarrow P & /\ (\Psi_2)S' \end{array}}{(\Psi_0)\Psi \vdash X\cdot S \Leftarrow P \ /\ (\Psi_2)x\cdot S'}$$

x is new by premises
$\vdash \Psi_0\ \mathsf{ctx}$ by assumption
$\Psi_1, \cdot \vdash A' \Leftarrow \mathsf{type}$ and $\vdash \Psi_1\ \mathsf{ctx}$ and $\Psi_0 \subseteq \Psi_1$ by i.h.
$\vdash \Psi_1, X \sim x{:}A'\ \mathsf{ctx}$ by typing rules
$\Psi_0 \subseteq (\Psi_1, X \sim x{:}A')$ by prefix property
$\Psi_2, \Psi \vdash S' : A' \Leftarrow P$ and $\Psi_1, X \sim x{:}A' \subseteq \Psi_2$ and $\vdash \Psi_2\ \mathsf{ctx}$ by i.h. (2)
$\Psi_2(x) = A'$ since $(\Psi_1, X{\sim}x{:}A')(x) = A'$ and previous line
$\Psi_2, \Psi \vdash x\cdot S' \Leftarrow P$ by typing rules
$\Psi_0 \subseteq \Psi_2$ by transitivity

Case

$$\mathscr{D} = \frac{\begin{array}{llllll} u \notin \Psi_0 \quad \Upsilon(u) = Q[\Psi_q] \quad [\sigma']^a_{\Psi'_q} Q' = P & & & & & \\ (\Psi_0, u{\sim}x{:}\_) & & \vdash \Psi_q & \mathsf{ctx} & /\ (\Psi_1)\Psi'_q & \\ (\Psi_1) & \Psi'_q & \vdash Q & \Leftarrow \mathsf{type} & /\ (\Psi_2)Q' & \\ (\Psi_2, u{\sim}x{:}\Pi\Psi'_q.Q') & \Psi & \vdash \sigma & \Leftarrow \Psi'_q & /\ (\Psi_3)\sigma' & \mathsf{subToSpine}_{\Psi'_q}(\sigma') = S \end{array}}{(\Psi_0)\Psi \vdash u[\sigma] \Leftarrow P \ /\ (\Psi_3)x\cdot S}$$

x is new by assumption
$\Psi_1 \vdash \Psi'_q\ \mathsf{ctx}$ and $\Psi_0 \subseteq \Psi_1$ and $\vdash \Psi_1\ \mathsf{ctx}$ by i.h.
$\Psi_2 \vdash \Psi'_q\ \mathsf{ctx}$ by weakening
$\Psi_2, \Psi'_q \vdash Q' \Leftarrow \mathsf{type}$ and $\Psi_1 \subseteq \Psi_2$ and $\vdash \Psi_2\ \mathsf{ctx}$ by i.h.
$\vdash \Psi_2, X{\sim}x{:}\Pi\Psi'_q.Q'\ \mathsf{ctx}$ by typing rules
$\Psi_3, \Psi \vdash \sigma' \Leftarrow \Psi'_q$ and $\Psi_2, X{\sim}x{:}\Pi\Psi'_q.Q' \subseteq \Psi_3$ by i.h. (3)
$[\sigma']^a_{\Psi'_q}(Q') = P$ by premise
$\mathsf{subToSpine}_{\Psi'_q}(\sigma') = \mathsf{toSpine}_{\Psi'_q}(\sigma')\ \mathsf{nil} = S$ by definition
$\Psi_3, \Psi \vdash \mathsf{nil} : [\sigma']^a_{\Psi'_q}(Q') \Leftarrow P$ by typing rule
$\Psi_3, \Psi \vdash S : \Pi\Psi'_q.Q' \Leftarrow P$ by lemma 7.1

$x{:}\Pi\Psi'_q.Q' \in \Psi_3$ by previous lines
$\Psi_3, \Psi \vdash x \cdot S \Leftarrow P$ by typing rule

$$\textbf{Case } \mathscr{D} = \frac{\begin{array}{c} X{\sim}x{:}\Pi\Psi_q.Q \in \Psi_0 \quad [\sigma']^a_{\Psi_q} Q = P \\ (\Psi_0)\Psi \vdash \sigma \Leftarrow \Psi_q \;/\; (\Psi_1)\sigma' \quad \mathsf{subToSpine}_{\Psi_q}\ \sigma' = S \end{array}}{(\Psi_0)\Psi \vdash u[\sigma] \Leftarrow P \;/\; (\Psi_1)x \cdot S'}$$

$\Psi_1, \Psi \vdash \sigma' \Leftarrow \Psi_q$ and $\Psi_0 \subseteq \Psi_1$ and $\vdash \Psi_1$ ctx by i.h. (3)
$\mathsf{subToSpine}_{\Psi'_q}(\sigma') = \mathsf{toSpine}_{\Psi'_q}\ (\sigma')\ \mathsf{nil} = S$ by definition
$\Psi_3, \Psi \vdash \mathsf{nil} : [\sigma']^a_{\Psi'_q}(Q') \Leftarrow P$ by typing rule
$\Psi_1, \Psi \vdash S : \Pi\Psi_q.Q \Leftarrow P$ by lemma 7.1
$\Psi_1, \Psi \vdash x \cdot S \Leftarrow P$ by typing rule

$$\textbf{Case } \mathscr{D} = \frac{}{(\Psi_0)\Psi \vdash \mathsf{nil} : P \Leftarrow P \;/\; (\Psi_0)\mathsf{nil}}$$

$\cdot \vdash \Psi_0\Psi$ ctx and $\Psi_0, \Psi \vdash P \Leftarrow \mathsf{type}$ by assumption
$\Psi_0, \Psi \vdash \mathsf{nil} : P \Leftarrow P$ by typing rule
$\Psi_0 \subseteq \Psi_0$ by definition
$\vdash \Psi_0$ ctx by assumption

$$\textbf{Case } \mathscr{D} = \frac{(\Psi_0)\Psi \vdash M \Leftarrow A \;/\; (\Psi_1)M' \quad (\Psi_1)\Psi \vdash S : [M'/x]^a_A(B) \Leftarrow P \;/\; (\Psi_2)S'}{(\Psi_0)\Psi \vdash M;S : \Pi x{:}A.B \Leftarrow P \;/\; (\Psi_2)M';S'}$$

$\Psi_1, \Psi \vdash M' \Leftarrow A$ by i.h. (1)
$\vdash \Psi_1$ ctx and $\Psi_0 \subseteq \Psi_1$ by i.h. (1)
$\Psi_0, \Psi \vdash \Pi x{:}A.B \Leftarrow \mathsf{type}$ by assumption
$\Psi_0, \Psi, x{:}A \vdash B \Leftarrow \mathsf{type}$ by typing inversion
$\Psi_1, \Psi, x{:}A \vdash B \Leftarrow \mathsf{type}$ by weakening using $\Psi_0 \subseteq \Psi_1$
$\Psi_1, \Psi \vdash [M'/x]^a_A(B) \Leftarrow \mathsf{type}$ by substitution lemma
$\Psi_2, \Psi \vdash S' : [M'/x]^a_A(B) \Leftarrow P$ by i.h. (2)
$\vdash \Psi_2$ ctx and $\Psi_1 \subseteq \Psi_2$ by i.h. (2)
$\Psi_2, \Psi \vdash M' \Leftarrow A$ by weakening ($\Psi_1 \subseteq \Psi_2$)
$\Psi_2, \Psi \vdash M';S' : \Pi x{:}A.B \Leftarrow P$ by typing rule

$$\textbf{Case } \mathscr{D} = \frac{(\Psi_1)\Psi \vdash M \Leftarrow [\sigma']^a_{\Psi'}A \;/\; (\Psi_2)M' \quad (\Psi_0)\Psi \vdash \sigma \Leftarrow \Psi' \;/\; (\Psi_1)\sigma'}{(\Psi_0)\Psi \vdash \sigma, M \Leftarrow \Psi', x{:}A \;/\; (\Psi_2)\ (\sigma', M')}$$

$\vdash (\Psi', x{:}A)$ ctx by assumption
$\vdash \Psi'$ ctx and $\Psi' \vdash A \Leftarrow \mathsf{type}$ by typing rules
$\Psi_1, \Psi \vdash \sigma' \Leftarrow \Psi'$ by i.h. (3)
$\vdash \Psi_1$ and $\Psi_0 \subseteq \Psi_1$ by i.h. (3)
$\Psi_1, \Psi \vdash [\sigma']^a_{\Psi'}(A) \Leftarrow \mathsf{type}$ by substitution lemma
$\Psi_2, \Psi \vdash M' \Leftarrow [\sigma']^a_{\Psi'}A$ by i.h. (1)

$\vdash \Psi_2$ ctx and $\Psi_1 \subseteq \Psi_2$ by i.h. (1)
$\Psi_2, \Psi \vdash \sigma' \Leftarrow \Psi'$ by weakening
$\Psi_2, \Psi \vdash \sigma', M' \Leftarrow \Psi', x{:}A$ by typing rule

Case $\mathscr{D} = \dfrac{(\Psi_0, \Psi)(x) = A' \quad [\sigma']^a_{\Psi'}(A) = A' \quad (\Psi_0)\Psi \vdash \sigma \Leftarrow \Psi' \;/\; (\Psi_1)\sigma'}{(\Psi_0)\Psi \vdash \sigma; x \Leftarrow \Psi', x{:}A \;/\; (\Psi_2)\,(\sigma', x)}$

$\vdash (\Psi', x{:}A)$ ctx by assumption
$\vdash \Psi'$ ctx and $\Psi' \vdash A \Leftarrow$ type by typing rules
$\Psi_1, \Psi \vdash \sigma' \Leftarrow \Psi'$ by i.h. (3)
$\vdash \Psi_1$ and $\Psi_0 \subseteq \Psi_1$ by i.h. (3)
$\Psi_1, \Psi \vdash [\sigma']^a_{\Psi'}(A) \Leftarrow$ type by substitution lemma
$(\Psi_1, \Psi)(x) = A'$ and $[\sigma']^a_{\Psi'}(A) = A'$ by previous lines
$\Psi_1, \Psi \vdash \sigma'; x \Leftarrow \Psi', x{:}A$ by typing rule

Case $\mathscr{D} = \dfrac{}{(\Psi_0)\Psi \vdash \cdot \Leftarrow \cdot \;/\; (\Psi_0)\cdot}$

$\cdot \vdash \Psi_0, \Psi$ ctx and $\Psi_0, \Psi \vdash P \Leftarrow$ type by assumption
$\Psi_0, \Psi \vdash \cdot \Leftarrow \cdot$ by typing rule

□