JFP **23** (1): 1–37, 2013. © Cambridge University Press 2012
doi:10.1017/S0956796812000408 First published online 2 November 2012

An insider's look at LF type reconstruction: everything you (n)ever wanted to know

BRIGITTE PIENTKA
McGill University, Montreal, Quebec, Canada
(*e-mail:* 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 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 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 stacked atop the trusted LF type checker which makes systems like Twelf practical and usable are ill-understood, and remains mysterious to the user and possible future implementors. To our knowledge, no formal descriptions of LF's type reconstruction algorithm exist and, hence, our understanding of it remains *ad hoc*. 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), or Epigram (McBride & McKinna, 2004). All of these systems support some form of *ad-hoc* type reconstruction to infer omitted arguments in practice. Yet there is a substantial gap between the implemented algorithms and the existing formal descriptions for such algorithms. If formal descriptions exist at all (see, for example, Norell, 2007), then they do not tackle the full expressive power found in the actual system and do not try to establish a formal relationship between the source-level implicit syntax available to the user and the elaborated fully explicit syntax. This makes it sometimes difficult to understand why reconstruction fails and under what conditions the user can expect the algorithm to succeed. One notable exception is the work by Luther (2001) who considered the Calculus of Construction, but Coq's implemented reconstruction algorithm is different and remains *ad hoc*. It would also require an extension of Luther's work to, for example, treat inductive types, Sigma-types, and η-expansion.

This is especially unfortunate, because we are seeing a recent push toward incorporating logical framework technology into mainstream programming languages to support the tight integration of specifying program properties with proofs that these properties hold (Licata & Harper, Authors' personal observations; Licata *et al*., 2008; Pientka, 2008; Poswolsky & Schürmann, 2008). However, the lack of foundations for LF type reconstruction remains a major obstacle in achieving this goal and building practical, dependently typed programming systems.

This paper tries to rectify the situation. The contribution of this paper is threefold: (1) We provide a practical insider's guide to the challenges surrounding LF type reconstruction (see Section 2) based on how it is realized in systems such as Twelf (Pfenning & Schürmann, 1999), Beluga (Pientka & Dunfield, 2010), or Delphin (Poswolsky & Schürmann, 2009). The implementation of LF type reconstruction in these systems is based on the ideas described in Pfenning (1991) which also form the basis of this work. We discuss here challenges and subtleties arising in practice such as: inferring the type of free variables, inferring omitted (implicit) arguments, handling dependencies between free variables and meta-variables which represent omitted arguments, and η-expansion. (2) We give a formal theoretical foundation based on contextual modal types (Nanevski *et al*., 2008) for LF type reconstruction together with a soundness and completeness proof. 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 is to the best of our knowledge the first theoretical justification for the soundness and completeness of LF type reconstruction. (3) Our formal development mirrors our implementation of type reconstruction in OCaml as part of the Beluga language (Pientka, 2008; Pientka & Dunfield, 2008, 2010). We have tested it on all the examples from the Twelf repository,[1] and the performance of our implementation is competitive.

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

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

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 hope our description will make LF technology and issues around type reconstruction in the presence of dependent types 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. We also believe our work provides general insights into how to develop type reconstruction algorithms for dependently typed programming which supports pattern matching and recursion. We have already used the described methodology 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 should help to spread dependent types to mainstream 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 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 when designing a practical system. Dependently typed objects can be very verbose, because they need to track instantiations for type arguments. This is typically considered to be impractical in practice and most systems allow users to leave out some arguments that the system can infer. Similarly, quantifying explicitly over all index arguments occurring in a dependent type is tedious. We, hence, want to infer the type of free variables. Higher-order abstract syntax means type reconstruction must rely on higher-order unification which is, in general, undecidable. To improve usability, we also must handle issues regarding η-contraction and expansion, since the only meaningful objects in LF are those in β-η-long form. 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 giving a typical example which showcases LF technology and highlights some of the issues which arise during reconstruction. The example is a formalization of a subset of the natural deduction calculus in LF and presented in Figure 1. The source code of the signature is given on the left. First, a type `o` for formulae and a type `i` for individuals is defined. Next, two constructors for propositions, conjunction, and universal quantification are defined. Conjunction is encoded using the constructor `and` which takes in two propositions and returns a proposition. The universal quantifier is interesting, since we need to model the scope of the variable it quantifies over. This is achieved by defining a constructor

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

Source-level program	Signature storing reconstructed program and number of reconstructed arguments
`% Types for formulas, individuals`	
`o: type .`	`o (0): type .`
`i: type .`	`i (0): type .`
`% Constructors for propositions`	
`and: o → o → o.`	`and (0): o → o → o.`
`all:(i → o) → o.`	`all (0):(i → o) → o.`
`% Natural deduction rules`	
`nd: o → type .`	`nd (0): o → type`
`andI:`	`andI (2): Π A:o. Π B:o.`
`nd A → nd B`	`nd A → nd B`
`→ nd (and A B).`	`→ nd (and A B).`
`allI:`	`allI (1): Π A:i → o.`
`(Π a:i. nd (A a))`	`(Π a:i. nd (A a))`
`→ nd (all A).`	`→ nd (all (λx. A x).`
`allE:`	`allE (1): Π A:i → o.`
`nd (all A)`	`nd (all (λx. A x)`
`→ Π T:i. nd (A T)`	`→ Π T:i. nd (A T).`

Fig. 1. Encoding of natural deduction in the logical framework LF.

`all` which takes an LF abstraction of type `i → o` as an argument and returns an object of type `o`. Modeling 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. Key advantages of this technique are: (1) α-renaming is inherited from the meta-language (2) β-reduction in LF can be used to model substitution in the object language. Finally, we consider the implementation of the natural deduction rules for conjunction introduction and universal quantifier introduction and elimination. The type family `nd` which is indexed by propositions encodes the main judgment of natural deduction. Each inference rule 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)`." Hence, the type of the constant `allI` is defined as `(Π x:i.nd (A a)) → nd (all A)`. Finally, the rule for `allE` says: "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 (2012) and Harper & and Licata (2007).

The goal of type reconstruction is to elaborate the signature on the left to the fully explicit well-typed signature on the right. This is achieved in two steps. During the first step, we insert meta-variables for omitted arguments, infer the types of free variables, and eta-expand bound and free variables, if necessary. Once we have found the most general instantiation for the omitted arguments such that we obtain a well-typed object, we abstract over the remaining meta-variables yielding the signature

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

on the right. The final result of type-reconstruction is a fully explicit well-typed LF object in β-normal and η-long form.

2.1 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 constant `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. We will come back to this idea when we describe the inference algorithm more formally in Section 6.2.

We store together with each constant the number of inferred arguments (see the reconstructed signature on the right in Figure 1). Arguments that are inferred are also called 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 implemented within Elf (Pfenning, 1991). Subsequently, it was also implemented in Twelf (Pfenning & Schürmann, 1999) and Twelf users testify that it works well in practice. We illustrate its basic principle and challenges next.

2.2 Type reconstruction—intermediate: inferring omitted arguments

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

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

The constant `andI-allI` specifies the transformation of a formula $(\forall x.A(x)) \wedge B$ into $\forall x.(A(x) \wedge B)$. The type of this constant 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, she 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, so that 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.

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

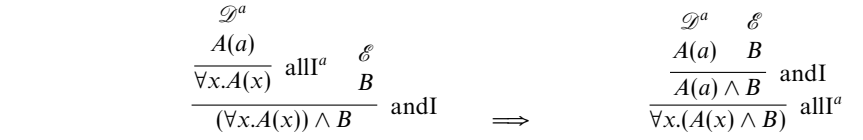

Fig. 2. Proof transformations.

Reconstruction translates 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 a name 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 is shown below. We highlight the inferred arguments in grey.

```
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)).
```

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

The reconstructed object contains the meta-variables `A1` and `A2` standing for 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

```
andI-allI:  Π A1:i → o. Π A2:o. Π D: Πx: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 a)  (D a) E)).
```

2.3 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 ___ (allI ___ D) T) 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 of 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)  (allI  (λx. A x)  D) T) 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 means that meta-variables characterizing holes may be instantiated with objects containing free variables; but,the type of free variables itself is unknown and

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

may contain meta-variables. Hence, there may be a circular dependency between meta-variables and free variables. There seems to be no easy way to avoid these circularities.

Fortunately, Reed (2009) pointed out that allowing circularities within meta-variables and free variables during unification is 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 that the final result is in β-normal and η-long form.

2.4 Type reconstruction: from simple types to dependent types

To understand the type reconstruction problem in LF better, we can try to relate and contrast it to type reconstruction for ML-like languages. Type reconstruction for ML-like languages refers to the problem of inferring the most general type or principal type for a given expression. The types normally considered are polymorphic, simple types. The input program does not change during type reconstruction and we can immediately verify that the input program indeed has the principal type which was inferred. In the algorithm for type reconstruction, we introduce meta-variables whenever the type of a variable or expression is unknown and the meta-variable is refined by first-order unification.

In LF, where we have dependent types, the type reconstruction problem is different. First, an LF object written by the user (the input) is not well-typed, because implicit arguments are allowed to be omitted. Second, it may contain free variables. Hence, type reconstruction for LF has a different task: Given an LF object written by the user, reconstruct a corresponding well-typed LF object. To infer omitted arguments, we introduce *meta-variables which stand for terms not types* and we rely on higher-order unification to find their most general instantiations. The problem of inferring most general omitted terms is orthogonal to the problem of inferring the most general type in simply typed languages.

While one could try to give a direct foundation to implicit LF syntax (see, for example, Reed, 2004), one advantage of reconstructing an explicit well-typed LF object is that we can verify independently the result of type reconstruction. While LF type reconstruction relies on higher-order unification and is complex, there is no need to trust it in practice, because we can verify the result independently by type checking. Moreover, algorithms which subsequently analyze LF signatures such as subordination (Virga, 1999) or coverage (Schürmann & Pfenning, 2003) have been developed for explicit LF. If we type check implicit LF signatures without elaborating them to explicit LF, we cannot directly take advantage of these algorithms. This paper provides a clear specification to LF type reconstruction relating an implicit LF object to its explicit LF counterpart. We assume the implicit LF object is

"approximately" well-typed objects[2] and concentrate on reconstructing omitted terms. To know which arguments were allowed to be omitted, type reconstruction for dependently typed objects must be type-directed. We show that if the reconstruction process succeeds, then the resulting explicit LF object is well-typed and is related to the implicit LF object written by the user. The reconstruction process is also complete in the sense that we will find the most general instantiation for the omitted terms, if they can be determined by higher-order pattern unification.

3 Implicit LF

In this section, we characterize the implicit syntax for LF which is closely related to the surface language and is the input language for the type reconstruction engine. 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 she 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.

Finally, 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. This allows us to concentrate on the main goal of reconstruction, namely inferring omitted terms and the types of free variables. We keep the implicit syntax small and concise (see Figure 3).

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 Figure 1. For convenience, we choose a spine representation (Cervesato & Pfenning, 2003). For example, the

[2] We introduce approximate typing on page 18.

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

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.

implicit object `(all λa. A a)` 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 she writes a term in its η-expanded form or not.

The subsequent development of type reconstruction will rely on exploiting certain syntactic properties about free variables to ensure that we can infer their types. For example, we want to know whether a free variable X is indeed applied to distinct bound variables (see also page 22). However, to check this condition effectively, we may need to η-contract objects. We consider here a special case: a given term m is simply the η-expansion of a bound variable. Hence, we define here the operation $\eta\mathsf{con}(m) = x$ which will verify that m is the η-expanded form of the term $x \cdot \mathsf{nil}$.

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

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 is to transform an implicit LF object into an equivalent explicit LF object which is in $\beta\eta$-long form.

4.1 Grammar

Explicit LF (see Figure 4) features meta-variables $u[\sigma]$, which are used to describe omitted implicit arguments following 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 them. The result of abstraction is a pure LF object which does not contain free variables nor meta-variables. In this development,

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

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	$\sigma ::= \cdot \mid \sigma, M \mid \sigma ; x$
Contexts	$\Psi ::= \cdot \mid \Psi, x{:}A$
Free variable contexts	$\Phi ::= \cdot \mid \Phi, X : A$
Meta-contexts	$\Upsilon ::= \cdot \mid \Upsilon, u{::}P[\Psi]$
Signature	$\Sigma ::= \cdot \mid \Sigma, a : (K, i) \mid \Sigma, c : (A, i)$

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

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. Moreover, we make a fine-grained distinction between atomic types, denoted with P, and general types, denoted with A, and similarly we distinguish normal terms M from neutral terms R. This allows us to discuss the type reconstruction algorithm more concisely.

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 stands for the set of meta-variables. Hence, the meta-variable context Υ is a characterization of 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 in Reed (2009).

In the simultaneous substitutions σ, we do not make the domain explicit. Rather we think of a substitution together with its domain Ψ where the ith element in σ corresponds to the ith declaration in Ψ.

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

Normal Terms

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

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

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

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.

Substitutions σ are built either from normal objects M or heads H. This is necessary since when we push σ through 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. Hence, 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 gives 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:

$\Upsilon;\Phi;\Psi \vdash M \quad \Leftarrow A$	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
$\Upsilon;\Phi;\Psi \vdash \sigma \quad \Leftarrow \Psi'$	Substitution σ has domain Ψ' and range Ψ

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

All judgments are in checking mode. For example, in the judgment for normal terms we check that M has type A. In the judgment for spines, the spine S has type A and target type P, i.e., S, A, and P are given. Recall that a spine S is associated with a head H to form a neutral term $H \cdot S$. The term $H \cdot S$ checks against a type P, if we can synthesize the type A of the head H and check that the spine has type A eventually yielding P. Let $S = M_1; \ldots; M_n; \mathsf{nil}$ and $A = \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.Q$ then $[M_1; \ldots; M_n]Q = P$.

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 meta-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.

The single substitution is written as $[M/x]^a_A(B)$ meaning we substitute M of type A for x in B. The superscript a indicates that the substitution applies to a type. The simultaneous substitution is written as $[\sigma]^a_\Psi(B)$ where Ψ is the domain of the substitution σ and B is a well-formed type in the context Ψ. Applying σ to the type B will then replace all the variables declared in Ψ with the corresponding term in σ.

Deciding whether two objects are equivalent is straightforward and essentially reduces to checking whether the two objects are α-equivalent. The only complication arises when checking equivalence of two substitutions. Since we have two equivalent ways to describe a simultaneous substitution which maps $y : \Pi B_1.B_2$ to itself, deciding whether two substitutions are equivalent must take into account η-contraction.

It is also worth stating when meta-variable contexts and free variable contexts are well-formed using the judgment $\vdash_\Phi \Upsilon\ \mathsf{mctx}$ and $\Upsilon \vdash \Phi\ \mathsf{fctx}$, respectively. The circularity between these two now becomes obvious; to define the well-formedness of Υ we rely on the free variable context Φ and *vice versa.* Moreover, circular dependencies within Υ and the free variable context Φ are not forbidden, since every element in Υ (or Φ) can itself depend on the entire context Υ (or Φ resp.). Given a free variable context Φ, Υ is a well-formed meta-context, if for each declaration $P[\Psi]$ in Υ, Ψ is a well-formed context referring to Φ and Υ (written as $\Upsilon; \Phi \vdash \Psi\ \mathsf{ctx}$) and P is a well-formed type in the context Φ, Υ and Ψ (written as $\Upsilon; \Phi; \Psi \vdash P \Leftarrow \mathsf{type}$).

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

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

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.

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

4.3 Substitutions

4.3.1 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 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 it is applied to a spine, the superscript s indicates it is applied to a substitution, and the superscript c indicates it is applied to a context. To guarantee that applying a substitution terminates, it is sufficient to consider the non-dependent approximation of the 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. Type approximations are 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.

$$\text{Type approximations } \alpha, \beta ::= \mathbf{a} \mid \alpha \to \beta$$

$$\begin{array}{ll} (\mathbf{a} \cdot S)^- & = \mathbf{a} \\ (\Pi x{:}A.B)^- & = A^- \to B^- \end{array}$$

Let us now consider the definition of hereditary substitution for normal terms. The definition for the other syntactic categories is straightforward.

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

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

redices.

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

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 sub-terms 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. However, such a substitution would be ill-typed in our setting, since R is not η-long form. On well-typed terms, the hereditary substitution operation will be defined. 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).

4.3.2 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 (Dowek *et al.*, 1996). Lowering replaces a meta-variable u of type $(\Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.P)[\Psi]$ with a term $\lambda x_1 \ldots \lambda x_n.v[\mathsf{id}(\Psi); x_1; \ldots; x_n]$ where the meta-variable v has type $P[\Psi, x_1{:}A_1, \ldots, x_n{:}A_n]$ (see also page 26 for the full definition). Because we only consider meta-variables of base types, contextual substitution does not need to be hereditarily defined.

Substitution for a meta-variable u which has type $P[\Psi]$ must carry a context and is written as $[\![\hat{\Psi}.R/u]\!]N$ and $[\![\hat{\Psi}.R/u]\!]\sigma$ where $\hat{\Psi}$ binds the variables in R and R has type P in the context Ψ. The context $\hat{\Psi}$ can be obtained from Ψ by dropping all the types and only retaining the names of the declarations. This allows us to consistently α-rename the bound variables in R when necessary and always achieve

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

that the bound variables in R are equal to the variables declared in the type of u. This complication can be eliminated in an implementation of our calculus based on de Bruijn indices.

Applying a single meta-substitution $\hat{\Psi}.R/u$ 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 sub-expressions. 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 with $[\sigma']^n_\Psi(R)$. 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 $[\sigma']^n_\Psi(R)$ 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 meta-variable u and write $[\![\hat{\Psi}.R/u]\!]_{P[\Psi]}(N)$, since when we encounter $N = u[\sigma]$ we compute $[\sigma']^n_\Psi R$ where we annotate the substitution $[\sigma']^n_\Psi$ with its domain. Subsequently, we usually omit this annotation on contextual substitutions to improve readability.

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.

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

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. We adapt the contextual substitution principle given by Reed (2009) to include the free variable context Φ. It can be extended to other judgments.

Theorem 4.2 (*Contextual Substitution Principles*)
If $\Upsilon' \vdash_{[\![\rho]\!]\Phi} \rho \Leftarrow \Upsilon$ and $\Upsilon; \Phi; \Psi \vdash M \Leftarrow A$ then
$\Upsilon'; [\![\rho]\!]\Phi; [\![\rho]\!]\Psi \vdash [\![\rho]\!]M \Leftarrow [\![\rho]\!]A$.

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.

For the 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

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

comparing two substitutions for equality.

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

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

We note that if $n = 0$ then we have $\eta\mathsf{exp}_{\mathbf{a}}(x) = x \cdot \mathsf{nil}$. 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.

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 . □

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 induction on A using the previous theorem. □

5 Equivalence relation between implicit and explicit terms

The goal of reconstruction is 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 14. 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:

$\psi \vdash m \approx M : \alpha$ term m is equivalent to term M at type approximation α

$\psi \vdash s \approx S : \alpha \Leftarrow \mathbf{a}$ spine s is equivalent to spine S at type approximation α and target type approximation $\mathbf{a}$

$\psi \vdash^i s \approx S : \alpha \Leftarrow \mathbf{a}$ spine s is equivalent to spine S at type approximation α and target type approximation $\mathbf{a}$ but the first i elements of S are irrelevant

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

Equivalence on normal objects

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

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

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

Equivalence on spines with omitted arguments

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

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: (1) both terms are lambda-abstractions. In this case, we check that the bodies are equivalent at type β in the extended context $\psi, x{:}\alpha$. (2) The implicit term is not in η-expanded form, while the explicit term is. In this case, we incrementally η-expand the implicit term m. We write $s_1 @ s_2$ for appending to the spine s_1 the spine s_2.

When we encounter a neutral term with a constant at the head, 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 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.

$\psi \vdash m \quad \Leftarrow \alpha$ term m has type approximation α

$\psi \vdash s : \alpha \Leftarrow \mathbf{a}$ spine s has type approximation α and target type approximation $\mathbf{a}$

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

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

We note that the approximate typing rules have access to a signature Σ which contains fully explicit types and kinds. In general, approximate typing is similar to the typing rules we find in the simply typed lambda-calculus with two exceptions: (1) we typically do not η-expand terms which are handled in our typing rules, and (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 . $\square$

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 an explicit object M (resp. S, A, K). The resulting object M is well-typed. The main judgments are as follows:

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

Reconstruction is type-directed. The given judgments 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:

$$\begin{array}{lll} & \vdash_{\Phi_1} \Upsilon_1 & \mathsf{mctx} \\ \Upsilon_1 & \vdash \Phi_1 & \mathsf{fctx} \\ \Upsilon_1;\Phi_1 & \vdash \Psi & \mathsf{ctx} \\ \Upsilon_1;\Phi_1;\Psi & \vdash A \Leftarrow & \mathsf{type} \end{array}$$

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 a free variable to Φ_1 once we encounter the free variable 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

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

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

In our implementation, meta-variables are implemented via references and have a global status. We collect the free variables together with their types as we traverse a given LF object and thread through the context of free variables. 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 When we reconstruct an atomic object $h \cdot s$ which is not of atomic type, we will η-expand it. η-expansion is done incrementally. Recall that we write $s_1 @ s_2$ for appending the spine s_1 to the spine s_2.

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

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 variable x will be expanded at a later point when it is necessary.

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

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

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

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

Synthesize spine

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

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

Check spine

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

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

Fig. 7. Reconstruction rules for LF.

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

There are four possible atomic objects $h \cdot s$ (bound head, constant head, known free head, unknown free head). Since different actions are required depending on the head h, our spine representation is particularly useful.

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 their η-expanded forms, 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 24).

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

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

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`, consider the object `D a` which is known to have type `nd X[p;a]`. We infer the type `Πx:i.nd X[p;x]` for the free variable `D`. Because `D` is only applied to the variable `a`, but not to the variable `p`, the bound variable `p` is left-over in `D`'s synthesized type. However, since `D` will eventually be 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 contextual type `o[q:i→o,x:i]`, such that it cannot depend on `q`. This is achieved by creating a meta-variable `Y` of 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. After pruning, the final type synthesized for the free variable `D` 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 can fail, if an undesired variable dependency is not prunable. For instance, given the assumptions `x:a, y:c x`, we encounter the object `D y x` and we will synthesize the type `Πy:c x.Πx:a.P` for the free variable `D`. However, we will never be able to prune away the first occurrence of `x` which occurs in the type of `y`. Pruning will fail and so would reconstructing a type for `D y x`. 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).

s is a pattern spine

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

Note we omit here the case where we encounter a term $X \cdot s$ where s is a non-pattern spine and we do not have the type of X. 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. One occurrence of the free variable X together with a pattern spine suffices to infer its type. We have not modeled 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 24.

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.

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

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

6.2 Working with spines

Let us consider the reconstruction of spines next. There are three different cases: (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.

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:

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

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

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

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 judgments 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 $x_1,\ldots,x_n$ that 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 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 it easier to establish a relationship between m and M which is important when establishing correctness of type reconstruction.

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

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 create a Π-type, however, 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 $M;S$.

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

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.

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

If i is zero, then no arguments need to be 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.

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

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.

Following Pientka (2003) and Dowek *et al.* (1996), we define lowering 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

For the theoretical development, we restrict ourselves to the decidable higher-order pattern unification fragment where all meta-variables must be applied to some

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

distinct bound variables. Throughout reconstruction, we maintain that the context for meta-variables and the free variable context are well-formed contexts according to our definition given on page 13. The result of reconstructing an implicit LF object m is a well-typed (explicit) LF object M and m is equivalent to M. Moreover, reconstruction generates a contextual substitution ρ with domain Υ_1 and range Υ_2, and we have

$$\Upsilon_2; \Phi_2 \; ; \; [\![\rho]\!]\Psi \vdash M \Leftarrow [\![\rho]\!]A.$$

To ensure variable dependencies are properly taken into account, we will need to prune the types 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 contexts Υ_2 and Φ_2 on the other hand is not a simple subset relation. Instead, we have $[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ where ρ is the pruning substitution s.t. $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$.

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 . □

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'.

The main purpose of the completeness theorem is to guarantee that type reconstruction will find the most general instantiations for the omitted arguments, if such instantiations exist and can be computed.

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

Because most general unifiers only exist for the pattern fragment, we will restrict occurrences of all meta-variables to patterns. Moreover, we assume that we know the (approximate) types of free variables. This is intrinsic to our definition of equivalence between m and M. However, the conditions under which we can infer the type of free variables have been clearly stated and justified in Section 6 (see page 22). Inferring the type of free variables is independent of showing that the reconstructed omitted arguments are most general.

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

Proof

Structural induction on the first derivation . □

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 (resp. 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

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

judgments:

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

We can read the first judgment as follows: Given a well-typed term M of type B in the context Ψ_0, Ψ with free variables Φ and meta-variables Υ, we generate a well-typed corresponding object N in the context Ψ_1, Ψ where Ψ_1 is an extension of Ψ_0 containing declarations for the free variables and meta-variables occurring in M. 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, since we pair up the bound variable name 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.

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

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 an 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 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.

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

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

Abstraction for normal terms

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

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

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

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

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

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

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

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

Fig. 8. Abstraction for LF objects.

type dependencies are erased from Ψ before computing the corresponding spine S.

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

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

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

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 . □

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 . □

8 Implementation

We implemented the two described phases of type reconstruction together with higher-order unification in OCaml as part of the Beluga implementation (Pientka & Dunfield, 2010). 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 bi-directional type checking and normal forms. This means we can omit typing annotations at λ-abstractions. We also 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 $u[\sigma] = R$ 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

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

fragment and we cannot guarantee that applying the inverse substitution $[\sigma]^{-1}$ to $v[\tau]$ is well-defined. 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.

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 paper, we follow the philosophy in systems such as Elf (Pfenning, 1991) and Twelf (Pfenning & Schürmann, 1999). If a constant declaration has an implicit quantifier-prefix, then these arguments must be omitted whenever we use this constant. Underscores may be used at any place where a term is legal. The analysis is done one constant at a time for LF declarations. We follow the same methodology in the implementation of 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, terms are η-expanded and missing type annotations at Π-types are inferred using type variables. This first phase is driven by approximate types and is similar to the well-known type inference problem in functional languages. In the second phase, omitted arguments and the full type of free variables are inferred similar to our description. There are two advantages to this approach: Twelf 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 one needs to formalize both phases and type and term reconstruction are intertwined. We also found it easier to explain the essential challenges in a one-pass algorithm which concentrates on reconstructing omitted terms but assumes that the given LF object is approximately well-typed. Our type reconstruction algorithm expects terms in β-normal form and throughout we work with objects in $\beta\eta$-normal form which are the only meaningful objects in LF. Using a bi-directional algorithm naturally enforces that we are working with $\beta\eta$-normal forms and allows us to omit type information in our terms; in particular, we do not need type labels on lambda-abstractions. As a consequence, when we reconstruct a term, we do not simultaneously reconstruct types and terms nor do we need to add type variables representing omitted type information. Instead, our type reconstruction algorithm concentrates on inferring

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

omitted terms and we characterize them via meta-variables. This disentangles the issues around type inference and reconstruction and simplifies our correctness proofs.

Unlike systems such as Agda (Norell, 2007), we provide no explicit way to specify that an argument should be synthesized or to explicitly override synthesis. Because the order of variables which occur in inferred arguments is unknown to the user, this would also be difficult. In the case where type reconstruction needs more information, the user needs to supply a typing annotation or explicitly specify the type of a variable to constrain the remaining type reconstruction problem. However, so far we have only discovered few cases where the user must supply explicit type information to ensure that all unification problems generated by type reconstruction can be solved.[3]

In contrast to Agda, 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, 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.

Elaboration from implicit to explicit syntax was first mentioned by Pollack (1990) although no concrete algorithm to reconstruct omitted arguments was given. Luther (2001) refined these ideas as part of the TYPELab project. He describes 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, reconstruction 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 that he 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, the theoretical description

[3] One such example arises in the implementation of the type preservation proof for System F.

https://doi.org/10.1017/S0956796812000408 Published online by Cambridge University Press

and proofs in Norell (2007) do not cover many practical features found in Agda. For example, there is no treatment of η-expansion, Σ-types and unit-types are not handled, meta-variables cannot be instantiated with function types, there is no treatment of pattern matching, and it does not treat universe hierarchies.

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 Barras & and Bernardo (2008), Miquel (2001), and Hagiya & and Toda (1994). In the setting of LF, 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 lose flexibility and sometimes more information must be supplied to guarantee that we can type check those compact LF objects. Another disadvantage is that many existing algorithms about LF signatures such as unification, subordination analysis, mode, termination, and coverage checking have been described on fully explicit LF signatures. If we do not elaborate implicit LF signatures to fully explicit LF signatures, then we must revise and reconsider the theoretical foundation for such algorithms.

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. To the best of our knowledge, this paper is the first theoretical description and justification of LF type reconstruction. Our work 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. Our formal development mirrors our implementation of type reconstruction in OCaml as part of the Beluga language. We believe this work is the first step to developing a theoretical justification for type reconstruction for dependently type programming languages which feature pattern matching and recursion in particular for type reconstruction for the Beluga language (Pientka & Dunfield, 2010).

Acknowledgments

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

References

Abadi, M., Cardelli, L., Curien, P.-L. & Lèvy, J.-J. (1990) Explicit substitutions. In *Proceedings of the 17th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, pp. 31–46.

Aydemir, B., Charguéraud, A., Pierce, B., Pollack, R. & Weirich, S. (2008) Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Wadler, P. (ed.). ACM Press, pp. 3–15r.

Barras, B. & Bernardo, B. (2008) The implicit calculus of constructions as a programming language with dependent types. In *Proceedings of the 11th International Conference on Foundations of Software Science and Computational Structures (FOSSACS'08)*, Amadio, R. M. (ed.), Lecture Notes in Computer Science (LNCS 4962). Springer, pp. 365–379.

Bertot, Y. & Castéran, P. (2004) *Interactive Theorem Proving and Program Development. Coq'art: The Calculus of Inductive Constructions*. Springer.

Boespflug, M. (2010) *Dedukti*. Available at: `http://www.lix.polytechnique.fr/dedukti`

Cervesato, I. & Pfenning, F. (2003) A linear spine calculus. *J. Logic Comput.* **13**(5), 639–688.

Crary, K. (2003) Toward a foundational typed assembly language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*. New Orleans, LA: ACM Press, pp. 198–212.

Dowek, G. (1993) The undecidability of typability in the lambda-pi-calculus. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications (TLCA '93)*. London: Springer-Verlag, pp. 139–145.

Dowek, G., Hardin, T. & Kirchner, C. (1995) Higher-order unification via explicit substitutions. In *Proceedings of the 10th Annual Symposium on Logic in Computer Science*, Kozen, D. (ed.). San Diego, CA: IEEE Computer Society Press, pp. 366–374.

Dowek, G., Hardin, T., Kirchner, C. & Pfenning, F. (1996) Unification via explicit substitutions: The case of higher-order patterns. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Maher, M. (ed.). Bonn: MIT Press, pp. 259–273.

Hagiya, M. & Toda, Y. (1994) On implicit arguments. In *Logic, Language and Computation: Festschrift in Honor of Satoru Takasu*. Jones, N. D., Hagiya, M. & Sato, M. (eds.), Lecture Notes in Computer Science, vol. 792. Springer, pp. 10–30.

Harper, R., Honsell, F. & Plotkin, G. (1993) A framework for defining logics. *J. ACM* **40**(1), 143–184.

Harper, R. & Licata, D. R. (2007) Mechanizing metatheory in a logical framework. *J. Funct. Program.* **17**(4–5), 613–673.

Lee, D. K., Crary, K. & Harper, R. (2007) Towards a mechanized metatheory of standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. New York: ACM Press, pp. 173–184.

Licata, D. R., Zeilberger, N. & Harper, R. (2008) Focusing on binding and computation. In *Proceedings of the 23rd Symposium on Logic in Computer Science*, Pfenning, F. (ed.). IEEE Computer Society Press, pp. 241–252.

Luther, M. (2001) More on implicit syntax. In *Proceedings of the First International Joint Conference on Automated Reasoning (IJCAR'01)*, Gore, R., Leitsch, A. & Nipkow, T. (eds.), Lecture Notes in Artificial Intelligence (LNAI) 2083. Springer, pp. 386–400.

McBride, C. & McKinna, J. (2004) The view from the left. *J. Funct. Program.* **14**(1), 69–111.

Miller, D. (1991) Unification of simply typed lambda-terms as logic programming. In *Proceedings of the 8th International Logic Programming Conference*. MIT Press, pp. 255–269.

Miquel, A. (2001) The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Aplications (TLCA'01)*, Abramsky, S. (ed.), Lecture Notes in Computer Science (LNCS 2044). Springer, pp. 344–359.

Nanevski, A., Pfenning, F. & Pientka, B. (2008) Contextual modal type theory. *ACM Trans. Comput. Logic* **9**(3), 1–49.

Necula, G. C. (1997) Proof-carrying code. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages (POPL'97)*. ACM Press, pp. 106–119.

Necula, G. C. & Lee, P. (1998) Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, Pratt, V. (ed.). Indianapolis, IN: IEEE Computer Society Press, pp. 93–104.

Norell, U. (2007 September) *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD. thesis, Department of Computer Science and Engineering, Chalmers University of Technology. Technical Report 33D.

Pfenning, F. (1991) Logic programming in the LF logical framework. In *Logical Frameworks*, Huet, G. & Plotkin, G. (eds.), Cambridge University Press, pp. 149–181.

Pfenning, F. (2012) *Computation and Deduction*. Cambridge University Press. In press.

Pfenning, F. & Schürmann, C. (1999) System description: Twelf — a meta-logical framework for deductive systems. In *16th International Conference on Automated Deduction (CADE-16)*, Ganzinger, H. (ed), Lecture Notes in Artificial Intelligence, vol. 1632. Springer, pp. 202–206.

Pientka, B. (2003) *Tabled Higher-Order Logic Programming*. PhD. thesis, Department of Computer Science, Carnegie Mellon University. CMU-CS-03-185.

Pientka, B. (2007) Proof pearl: The power of higher-order encodings in the logical framework LF. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'07)*, Schneider, K. & Brandt, J. (eds.), Lecture Notes in Computer Science (LNCS 4732). Springer, pp. 246–261.

Pientka, B. (2008) A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*. ACM Press, pp. 371–382.

Pientka, B. & Dunfield, J. (2008) Programming with proofs and explicit contexts. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*. ACM Press, pp. 163–173.

Pientka, B. & Dunfield, J. (2010) Beluga: A framework for programming and reasoning with deductive systems (System Description). In *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Giesl, J. & Haehnle, R. (eds.). Lecture Notes in Artificial Intelligence (LNAI 6173). Springer-Verlag, pp. 15–21.

Pollack, R. (1990). *Implicit syntax*. Proceedings of First Workshop on Logical Frameworks, Eds. G. Huet and G. Plotkin, pp 421–435

Poswolsky, A. B. & Schürmann, C. (2008) Practical programming with higher-order encodings and dependent types. In *Proceedings of the 17th European Symposium on Programming (ESOP '08)*, vol. 4960. Springer, pp. 93–107.

Poswolsky, A. & Schürmann, C. (2009) System description: Delphin—a functional programming language for deductive systems. In *Proceedings of the International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'08)*, Electronic Notes in Theoretical Computer Science (ENTCS), vol. 228. Elsevier, pp. 135–141.

Reed, J. (2004) Redundancy Elimination for LF. In *4th Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, Schürmann, C (ed.). Electronic Notes in Theoretical Computer Science (ENTCS) vol 199, pp 89–106.

Reed, J. (2009) Higher-order constraint simplification in dependent type theory. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'09)*, Felty, A & Cheney, J. (eds.). ACM Press, pp 49–56.

Schürmann, C. & Pfenning, F. (2003) A coverage checking algorithm for LF. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLS'03)*, Basin, D. & Wolff, B. (eds.). Springer, pp. 120–135.

Virga, R. (1999) *Higher-Order Rewriting with Dependent Types*. PhD. thesis, Department of Mathematical Sciences, Carnegie Mellon University. CMU-CS-99-167.

Watkins, K., Cervesato, I., Pfenning, F. & Walker, D. (2002) *A Concurrent Logical Framework I: Judgments and Properties*. Technical Report. CMU-CS-02-101. Department of Computer Science, Carnegie Mellon University.