A thesis submitted to McGill University in partial fulfilment of the requirements of the degree of Master of Science

Implementation of a Dependently Typed Functional Programming Language

Renaud Germain

renaud.germain@cs.mcgill.ca

School of Computer Science

McGill University, Montreal

April 21, 2010

Abstract

In recent years, dependent type systems have gathered interest because they make it possible to express stronger properties about programs. However, they are also very verbose. In this thesis, we show how to eliminate some of this verbosity (for the user) by doing reconstruction over dependent types. More precisely, we present the work done in the implementation of the Beluga programming language. Our goal is to present the key issues arising in reconstruction and give a formal and accessible description of ideas that have been around for some time but never given the spotlight. We also prove the soundness of our reconstruction algorithm.

Résumé

Au cours des dernières années, les types dépendants ont reçu un intérêt particulier parce qu'ils permettent d'exprimer des propriétés plus précises sur les programmes. Cependant, ces systèmes de typage sont aussi très redondants. Dans cette thèse, nous allons expliquer comment éliminer une partie de cette redondance en reconstruisant ces types dépendants. Plus précisément, nous présenterons le travail fait sur l'implémentation du langage Beluga. Notre but est de présenter les problématiques importantes liées à la reconstruction de types dépendants et de présenter de façon formelle et accessible certaines idées qui, malgré le fait qu'elle ne sont pas toutes récentes, sont toujours restées dans l'ombre jusqu'à maintenant. Une preuve de correction de notre algorithme de reconstruction est aussi donnée.

Acknowledgement

I would like to thank my supervisor Brigitte Pientka for her guidance, comprehension, academic and financial support throughout my studies and in particular during the writing of this thesis.
I also want to thank my family, friends and my fiancée for their constant trust, love and support throughout the years (and extra year) that it took me to write it.

Contents

Chapter 1

Introduction

1.1 Motivation

Over the last few years, dependent types and higher order abstract syntax (HOAS) have received a lot of attention as technical means to represent programs and reason about them. The idea behind dependent types is to have types that depend on values. For example, we could index the type of lists with their length.

```
nat: type.
z: nat.
s: nat -> nat.
list: nat -> type.
```

Given the two constructors for natural numbers zero (`z`) and successor (`s`), the types of (natural number) list constructors would then become:

```
nil: list z.
cons: nat -> list N -> list (s N).
```

Using these definitions, we could be more precise about the types of list functions like append (`list N -> list M -> list (M + N)`), reverse (`list N -> list`

`N`), etc. In general, dependent types will allow us to express stronger properties about the nature of computations.

In addition to dependent types, the language we describe in this thesis also supports higher order abstract syntax. This technique allows elegant representation of object-languages containing binders by representing binders in the object-language by binder in the meta-language. As such, it simplifies the system by eliminating the need to check for alpha equivalence. Under its apparent simplicity, HOAS also avoids to the language theorist the need to implement the various substitution operations, a traditionally error-prone task, by reducing the problem to β-reductions in the meta-language.

However, having a language that encompass dependent types and unrestricted recursion, as proposed in Cayenne [Augustsson, 1998], would also make type checking undecidable. Moreover, this would prevent us from writing proofs about an object-language represented using HOAS, as full recursion would allow exotic terms (i.e. type correct meta-language terms that would not correspond to any object-language terms) to be present, thus breaking adequacy of the encoding.

While some systems (Agda [Norell, 2007], Epigram [Altenkirch et al., 2005], chose to provide a weaker form of recursion (structural), at the cost of being less fit for use as a programming language, an alternative solution to this problem is to split the language between a data level and a computation level and consequently, keep type checking tractable. This is the approach followed in Delphin [Poswolsky and Schürmann, 2008], Beluga [Pientka, 2008] and the idea behind type polarity proposed by Licata [Licata and Harper, 2009].

One must note that dependently typed lambda calculi are also relevant to the area of proof theory as they can be used to define different logics. However, this will not be the main focus of this thesis.

Dependent types are also by nature, pretty redundant. To take the list example again, the reader might have noticed that `N` and `M` were actually free variables in the types of `cons`, append and reverse. To be really formal, we should have quantified over those variables. The type of `cons` would then become

```
cons:  Π N:nat .  nat -> list N -> list (s N).
```

However, this also means that whenever we want to use `cons` in the future, we will have to supply it with an *additional* argument `N`. This verbosity will only increase the more complex the functions become. Still, one might wonder if we could do without those explicit indices given how simple it is to deduce the type of `N` and `M` in the previous example.

Ideas on what information can be omitted in a user-level syntax have been around since [Pollack, 1990]. While omitting redundant information might seem at first like a mere convenience, Luther found in [Luther, 2001] that a 50% reduction in the size of terms could be achieved for a variant of the calculus of construction. This reduction in size is also of interest to the area of proof-carrying code [Necula, 1997], as the terms are carried around with the code.

There exists two main schools of thought on how to eliminate that redundancy for the programmer. One approach is to formalize an implicit, more compact, language that is shown to be equivalent to the original dependently typed language. This implicit language is then checked directly. This approach also has the potential of being more efficient. The other approach is to present a lightweight language to the user and then, reconstruct the program to a fully explicit form. This has the advantage to confine the redundancy elimination to a pre-phase without modifying the rest of the infrastructure. That leads, amongst other things, to a simpler (and easier to trust) type checker and the possibility to reuse existing algorithms for coverage, unification, termination, etc. This is our approach with Beluga.

1.2 Contribution

This thesis will present the work we have done so far regarding type checking and type reconstruction for the Beluga language. As hinted before, Beluga is a dependently typed functional programming language that supports HOAS. The language is split between a data layer that is essentially the logical framework

LF [Harper et al., 1993] and a computation layer that supports full recursion and performs computation over LF objects.

While the idea of implementing LF and performing type reconstruction are not new, this is to our knowledge, the first time it had been presented in a formalized way. The already existing descriptions are informal and vague on many important aspects [Pfenning, 1991], and therefore type reconstruction remains a black art. Moreover, the fact that Delphin is using Twelf [Pfenning and Schürmann, 1999] as a back-end leaves Beluga as one of the few competing implementation of the LF logical framework. As such, our general goal is to spread knowledge about LF technology by giving a tutorial on how to implement such a language in a realistic, efficient and provably correct way.

As such, our contributions are the following:

- We extend the contextual modal type theory [Nanevski et al., 2008] with the context variables and parameter variables (as proposed in [Pientka, 2008]) and formalize it with a spine notation [Cervesato and Pfenning, 2003] and explicit (delayed) substitution [Abadi et al., 1990]. The first extension will allow us to parameterize computation with contexts. The two others to provide an efficient implementation.

- We provide a foundation for type reconstruction over a dependently typed λ-calculus (LF, Beluga's object layer) and a proof of soundness of our algorithm.

- We provide an implementation of these features. The prototype is available on the Beluga website:

  `http://complogic.cs.mcgill.ca/beluga/`

- We also give a survey of other dependently typed systems that include some form of type reconstruction.

1.3 Structure of the thesis

To illustrate key concepts of the work done here, we first present in Chapter 2 a motivating example taken from the Twelf repository. The part we use consists in definitions for cartesian closed categories and a simply-typed lambda calculus and a translation between the two. Then, Chapter 3 will introduce some background to our work. Chapter 4 contains the formalization of Beluga data-level layer with explicit substitutions and spine notation. The reconstruction algorithm is given in Chapter 5. Finally, some related work is discussed in Chapter 6.

Chapter 2

Example: cartesian closed categories

In this chapter we introduce an example to illustrate key concepts of the work done in this thesis. In particular, we show how to represent a theory in LF using higher order abstract syntax. Throughout this presentation, we will highlight the challenges that reconstruction will have to address. Then, we will present a Beluga program to show how to reason over LF data.

The example is taken from the Twelf repository and due to Andrzej Filinski. Because Beluga's object layer is essentially LF, the syntax for our signature will be quite similar to the one in Twelf. For reference purpose, we annotate the figures with the name of the file from which code comes from. The complete source code can be found in Beluga's examples suite.

The sequence of declarations `c` : A. and `a` : K. is what we call a LF-signature Σ. `a` and `c` are called constants and A and K stand respectively for a type and a kind. These are introduced more formally in Chapter 3. The syntax used here is mostly the user-level one. The arrow "`A -> B`" is syntactic sugar for the type $\Pi x{:}A.B$ where variable x does not occur in type B. Capital letters are what we call *free variables* and λ-abstractions are written with a backslash (\). The

percent (%) sign is used to delimit comments.

2.1 Cartesian closed categories

Category theory has proven useful as a way to represent various mathematical structures. In particular, it is often used to model lambda calculi. In its simplest form a category is composed of only three elements: a class of objects, a class of morphisms over these objects and an operation of composition over morphisms that is associative and possesses an identity element.

In order to model a lambda calculus, a category will need three additional properties: a terminal object, products and exponentials. A object is terminal if for any object in the category, there exists a morphism from this object to the terminal object. A category is said to have products if for every pair of objects X and Y in the category, the product $X \times Y$ is an object of the category. We will not describe the notion of exponential immediately, as it is a little more complex and would bring no additional understanding at this point, but let's just say that it roughly corresponds to the notion of evaluation in a lambda calculus. A category having these three properties is called cartesian closed.

With these ideas in mind, we are now ready to formalize them in Beluga. We will describe how these various encodings are done and we then give the full definition for a cartesian closed category in figure 2.1 for ease of reference.

The first notions we formalize are those of objects (`obj`) and morphisms (`mor`). Objects are abstracted over and do not refer to anything else, so we introduce them as simple type constants. Morphism, on the other hand are always characterized by their source object (or domain) and their target object (or range), so we also define morphisms as type constants, but we index them with two objects. The definition for a category is completed by the composition operation, which we write (`@`) and the identity morphism (`idc`) over all objects. The usefulness of dependent types is already illustrated here by those two last constants: by indexing the type of morphisms with their domain/range, we have enough to describe precisely what kind of morphism the composition operation

and identity morphism describe.

Before continuing further, we make a few remarks on syntax. First, the arrow (`A -> B`) is syntactic sugar for a $\Pi x{:}A.B$ abstraction when B does not depend on the value of x. Therefore, the constant `mor` is represented internally as $\{x{:}\texttt{obj}\}\{y{:}\texttt{obj}\}\texttt{type}$. Note that we write Π-abstractions using the braces ({}) syntax, as it is the way we write them in ASCII.

Another remark is on the variables appearing in the definition for identity and composition. While the reader can easily make sense of these definitions, free variables do not make sense when reasoning formally. For this reason, the first step of the reconstruction algorithm will be to infer a type for these variables and reconstruct a Π-abstraction. For example, the reconstructed constant for identity will be (reconstructed part in red)

```
{A:obj}.morA A
```

This indeed make sense, as we need an identity morphism for every object, hence the Π-quantifier.

In general, it is not always possible to reconstruct the type of a free variable. We are able to reconstruct a type for a free variable when it appears, at least once, applied to a list of distinct bound variables (we call it a *pattern spine*, more on this in Chapter 5). As `A` above was not applied to anything, its type could be inferred.

Free variables are the base of our reconstruction algorithm. Indeed, since we wrote `A` implicitly above, whenever we will use the constant `idc` in the future, the system will expect no arguments (providing one would raise an error) and will reconstruct one argument, that is, the term that stands for the variable `A`. We could also have written `A` explicitly, but this would have made the code more verbose, for the definition *and* use of `idc`.

We give another example of reconstruction with the composition constant (`@`).

The reconstructed form will be

```
@ : {A:obj}{B:obj}{C:obj} mor B C -> mor A B -> mor A C.
```

We see that the Π prefix grows larger with the number of free variables while not adding much information. This gives an idea of the verbosity introduced by explicit dependent types. The term will grow even larger when free variables will be dependently typed. Similarly to the `idc` definition, the constant `@` will expect two arguments — of type `mor B C` and `mor A B` respectively — when we use it later on and the reconstruction algorithm will reconstruct three implicit arguments (`A`, `B` and `C`).

Terminal object is defined next. The definition is quite simple: it only mentions that `one` is actually an object and that for every object `A` there exists a morphism from `A` to `one`.

Products definition is quite simple too. We add in the projections (`fst` and `snd`) and the `pair` constant that describe the morphisms needed for a category to "have products".

As we said before, exponentials model the notion of function evaluation. As such, we define the `arrow` objects, that model the functions of type $A \mapsto B$ (`arrow A B`). The `app` constant serves to model applications and says that there exists a morphism that takes an object representing the pair of a function and its argument and returns an object representing the result. The dual constant `cur` says that whenever we have a morphism from the pair `A B` to `C`, we also have a morphism from `A` to "functions" from `B` to `C`. In a programmer's language, that means that whenever we have a "function" from a product the curried form of this function also exists, hence the constant's name.

2.2 Lambda calculus

Similarly, a simply typed lambda calculus with pairs is defined (fig. 2.2). Terms (`term`) are the base objects and come indexed with their type. We reuse the

```
% Basic category
obj : type.
mor : obj -> obj -> type.
idc : mor A A.
@   : mor B C -> mor A B -> mor A C.

% Terminal object
one   : obj.
drop  : mor A one.

% Products
cross : obj -> obj -> obj.
fst   : mor (cross A B) A.
snd   : mor (cross A B) B.
pair  : mor A B -> mor A C -> mor A (cross B C).

% Exponentials
arrow : obj -> obj -> obj.
app   : mor (cross (arrow B C) B) C.
cur   : mor (cross A B) C -> mor A (arrow B C).
```

Figure 2.1: Cartesian closed categories (ccc.elf)

definitions of objects defined in cartesian closed categories to stand for types in our lambda calculus. Products and exponentials therefore become the product and functional type respectively. The calculus only characterizes well-type terms and is quite minimal. It consists of λ-abstractions (`llam`) and applications (`lapp`), together with a constructor for pairs (`lpair`), projections (`lfst` and `lsnd`) and the base object unit (`lunit`).

A notable feature is the representation of λ-abstractions using HOAS. The parenthesizing in the `llam` declaration tells us that this constant expects *one* argument of *functional* type. In the full example, there is a notion of term convertibility written as

```
conv : term A -> term A -> type.
```

The rules are given in figure 2.3.

The usefulness of HOAS shines when it comes to writing the case for lambda

```
term : obj -> type.

% Functions
llam : (term A -> term B) -> term (arrow A B).
lapp : term (arrow A B) -> term A -> term B.

% Pairs
lpair : term A -> term B -> term (cross A B).
lfst  : term (cross A B) -> term A.
lsnd  : term (cross A B) -> term B.

% Unit
lunit : term one.
```

Figure 2.2: Simply-typed lambda-calculus (lambda.elf)

```
c_refl : conv E E.

c_fst  : conv (lfst E) (lfst E')
          <- conv E E'.

c_snd  : conv (lsnd E) (lsnd E')
          <- conv E E'.

c_pair : conv (lpair E1 E2) (lpair E1' E2')
          <- conv E1 E1'
          <- conv E2 E2'.

c_lam  : conv (llam (\x . E x)) (llam (\x . E' x))
          <- ({x: term _} -> conv (E x) (E' x)).

c_app  : conv (lapp E1 E2) (lapp E1' E2')
          <- conv E1 E1'
          <- conv E2 E2'.
```

Figure 2.3: Convertibility in the lambda calculus

terms congruence: we only have to verify that `E` and `E'` are convertible for an abstract `x`. The judgement is completely parametric on the meta-level variable x. Other cases are simple recursions and `c_refl` says that every term is convertible to itself.

2.3 Translation function

Translation from cartesian closed categories into lambda calculus is given by the `conc` constant. It is quite simple and works as follows: morphisms are translated to functions of the same type and these are later composed using constructors from the lambda calculus.

It is when the `cid` constant is reconstructed that we encounter our first instance of a meta-variable. As explained earlier, the first argument of `conc` (`idc` here) has to have type `mor A B`, which is indeed the case. However, because `idc` has been reconstructed to be a Π abstraction, the reconstruction for `cid` must take that into account. The algorithm will introduce a meta-variable as the first argument to `idc` so the type will still be well-formed. Unification will later be used to find the right instantiation for the meta-variable. In the present case, reconstruction will return the type

```
{A:obj} conc A A (idc A) (\x . x)
```

One might remark that we wrote some free variables in their η-expanded form. While it might make the functional type of some variable more explicit in some cases, Beluga is able to make the translation internally. For example, writing simply `conc F M` as the premise of the `ccur` case would have been equivalent and correct.

The definition for `conc` is the standard way to write such a translation in Twelf. We might think of `conc` as a function that takes a morphism and returns a function, but still, it is defined as a dependent type like everything else before it, without any reference to the way we think about it. To that end, the logical

```
conc  : mor A B -> (term A -> term B) -> type.

cid   : conc idc (\x . x).

ccomp : conc (@ F G) (\x . (M (N x)))
         <- conc G (\x . N x)
         <- conc F (\x . M x).

cunit : conc drop (\x . lunit).

cpair : conc (pair F G) (\x . (lpair (M x) (N x)))
         <- conc G (\x . N x)
         <- conc F (\x . M x).

cfst  : conc fst (\x . (lfst x)).

csnd  : conc snd (\x . (lsnd x)).

ccur  : conc (cur F) (\a . llam (\b . M (lpair a b)))
         <- conc F (\x . M x).

capp  : conc app (\a . lapp (lfst a) (lsnd a)).
```

Figure 2.4: Translation from ccc combinators to lambda-terms (conc.elf)

framework Twelf allows us to specify a "mode" for `conc`, written as

```
%mode conc +D -E.
```

This means that the first argument (the morphism) is to be thought of as an input and the second one as an output. This is needed, together with other declarations, if we want to verify that the translation is indeed a function (i.e. if it covers all the (morphisms) cases and does terminate).

This shows one of Twelf's limitation and one of the reasons behind Beluga design. Even if Beluga's computational layer is not the focus of this thesis, we show what the translation would look like when written as a Beluga function in figure 2.5. The function is written using a simple pattern matching on its only argument, `d`. The main difference is that we write functional objects as objects that depend on a context instead of as λ-abstractions. For example, the case for `idc` returns `[x] x` instead of `(\x . x)`. The `let` forms are syntactic sugar for pattern matching (`case` constructs).

Writing function like this has the advantage of making them built into the language. Therefore, termination and coverage checkers become part of the theory instead of being ad-hoc, patched on, features.

2.4 Congruence proof

While the `conc` example was interesting in itself, it only worked with closed objects. We present here an example to illustrate the use of context variables and parameter variables in Beluga. The function we present in figure 2.6 is a proof that the convertibility defined before (see fig. 2.3) is preserved under functions.

More precisely, the function takes as an input a function, together with a proof of convertibility between two terms and returns a proof of convertibility between the results of applying the function on the two terms. Again, `llam` is the interesting case here. Because the argument of `llam` is a function, we have to

```
rec conc : (mor A B)[] -> (term B)[u:term A] =
    fn d => case d of
        [] idc =>
           [x] x

      | [] (@ F G) =>
           let [x:term A] (M x)  = conc ([] F) in
           let [x:term A] (N x)  = conc ([] G) in
               [x] (M (N x))

      | [] drop =>
           [x] lunit

      | [] (pair F G)  =>
           let [x:term A] (M x) = conc ([] F) in
           let [x:term A] (N x) = conc ([] G) in
               [x] lpair (M x) (N x)

      | [] fst =>
          [x] (lfst x)

      | [] snd =>
           [x] (lsnd x)

      | [] (cur F) =>
           let [x:term (cross A B)] (M x) = conc ([] F) in
               [a] (llam (\b . (M (lpair a b))))

      | [] app =>
           [a] (lapp (lfst a) (lsnd a));
```

Figure 2.5: Translation from ccc combinators to lambda-terms in Beluga (conc.bel)

reason with open terms if we are to examine the body of the function. For that reason, whenever we encounter a binder, we add a variable to the context and call the `cong` function recursively. The fact that argument `c` depends on context `g` reflects this idea by saying that the proof can refer to "free" variables (the ones in `g` actually). Except for the binder issue, the treatment is quite similar to the other recursive cases (`lfst`, `lsnd`, `lpair` and `lapp`).

When we hit a base case that is one of these "free" variable, we use the fact that a variable will be convertible to itself and return `c_refl` as the proof. This case is written using a parameter variable `#p`. In context `[g, x:term A]`, `#p .. ` will match against any variable contained in `g` but not `x`. This is because `x` represents the input of the generic function `M` and not a binder we encountered in the past. For `#p` to match against any variable contained in the context, we would have to write `#p .. x`.

Because `x` is the input of function `M`, hitting the base case `x` means that `M` is actually the identity function. Therefore, the proof we got as an input is also a proof that two terms are convertible once we apply the identity to them. This is the reason why we simply return `c` in this case.

The base case for `lunit` uses the fact that `lunit` is convertible to itself by `c_refl`.

On a more technical level, using context `g` means we had to define a general shape for it. The schema definition for `ctx` says that `g` consists of variables of type `term a` for some `a`.

Abstraction for context are written with `FN` and abstraction for meta-variables — `M` representing the function in this case — are written with `mlam`, in a way that mimics the usual λ-abstractions (written `fn x => ...`).

A final remark on this example is that it is not always possible to reconstruct the type of every (free) pattern variable. In this case, we provide the types explicitly, as in the `lfst`, `lsnd` and `lapp` cases, at the cost of verbosity.

```
schema ctx = some [a:obj] term a;

rec cong : {g:(ctx)*} {M::(term B)[g, x:term A]}
           (conv (E ..) (E' ..))[g] ->
           (conv (M .. (E ..)) (M .. (E' ..)))[g] =
   FN g => mlam M => fn c =>
      case ([g,x:term _] M .. x) of
         [g, x:term A] x => c

       | [g, x:term A] #p .. => [g] c_refl

       | [g, x:term A] lunit => [g] c_refl

       | [g, x:term A] (lpair (M1 .. x) (M2 .. x)) =>
          let [g] LC1 .. = cong [g] <g, x . M1 .. x> c in
          let [g] LC2 .. = cong [g] <g, x . M2 .. x> c in
              [g] (c_pair (LC1 ..) (LC2 ..))

       | {N::(term (cross B1 B2))[g,x:term A]}
         [g, x:term A] (lfst (N .. x)) =>
          let [g] LC .. = cong [g] <g, x . N .. x> c in
              [g] (c_fst (LC ..))

       | {N::(term (cross B1 B2))[g,x:term A]}
         [g, x:term A] (lsnd (N .. x)) =>
          let [g] LC .. = cong [g] <g, x . N .. x> c in
              [g] (c_snd (LC ..))

       | [g, x:term A] llam (\y. N .. y x) =>
          let [g] D .. = c in
          let [g, y:term B1'] (LC .. y) =
                cong [g, y:term _] <g, y, x . N .. y x> ([g, y] D ..) in
              [g] (c_lam (\y . LC .. y))

       | {M1::(term (arrow B1 B2))[g, x:term A]}
         [g, x:term A] (lapp (M1 .. x) (M2 .. x)) =>
          let [g] LC1 .. = cong [g] <g, x . M1 .. x> c in
          let [g] LC2 .. = cong [g] <g, x . M2 .. x> c in
              [g] (c_app (LC1 ..) (LC2 ..));
```

Figure 2.6: Proof that congruence is preserved under a function (cong.bel)

Chapter 3

Background

Beluga data representation language is strongly inspired by contextual modal type theory [Nanevski et al., 2008]. The later is itself an extension of the logical framework LF [Harper et al., 1993]. This chapter presents a short overview of these two systems.

3.1 LF

LF is a dependently typed lambda calculus. There is very little more to that. In our presentation, we impose a syntactic restriction to allow only normal forms to be represented. As such, terms are split between normal and neutral terms to prevent the presence of redexes.

Contrarily to simple types, dependent types are not valid "by construction", so kinds are introduced to allows us to qualify on the validity of types. Contexts and substitutions complete the theory but are kept separated from the user-level language for the moment, as in Twelf.

3.1.1 Syntax

Kinds	K	$::=$	$\mathsf{type} \mid \Pi x{:}A.K$
Types	A, B	$::=$	$\Pi x{:}A.B \mid P$
Atomic Types	P	$::=$	$\mathsf{a} \mid P\ M$
Normal terms	M, N	$::=$	$\lambda x.M \mid R$
Neutral terms	R	$::=$	$\mathsf{c} \mid x \mid R\ M$
Contexts	Ψ, Φ	$::=$	$\cdot \mid \Psi, x{:}A$
Substitutions	σ	$::=$	$\cdot \mid \sigma, M/x$

3.1.2 Judgements

The type system we introduce here is a bidirectional one: neutral terms (resp. atomic types) can synthesize a type (resp. kind) while normal terms and types in general must be checked. We have two judgements for types and two for terms.

Types		Terms	
$\Psi \vdash A$	$\Leftarrow \mathsf{type}$	$\Psi \vdash M$	$\Leftarrow A$
$\Psi \vdash P$	$\Rightarrow K$	$\Psi \vdash R$	$\Rightarrow A$

The theory is completed by a meta-level judgement that checks that a context is well-formed and one that checks that a substitution has domain Φ and range Ψ (i.e. terms in σ can only refer to bound variables declared in Ψ).

Contexts	Substitutions
$\vdash \Psi\ \mathsf{ctx}$	$\Psi \vdash \sigma \Leftarrow \Phi$

3.1.3 Typing rules

The typing rules for LF are given in figure 3.1. These rules suppose the presence of a signature Σ where type level and object level constants are declared. In practice, these constants are declared in sequence so a declaration can refer to

the previous ones.

Because the only well-formed terms are normal ones, the substitution operation we use must not introduce redexes. In order to do so, it needs to keep normalizing whenever it creates one by substituting in a term. This operation is called hereditary substitution and is described in [Nanevski et al., 2008]. We annotate the substitution with the type of the term we are substituting to be able to reason about termination of the operation (as a matter of fact, we also annotate with the syntactic category category of the term we are substituting into, to keep up with the definition in [Nanevski et al., 2008]). Indeed, it is not obvious why hereditary substitution should terminate, as one could imagine a substitution that would keep on creating new redexes, forever. However, by having the type A around, the creation of a new redex would tell us two things: A is actually a function type and the type of the new term we are substituting is a smaller type than A, namely the input type of the function. Knowing this, it is easy to see why hereditary substitution will eventually terminate because all types are finite.

Because we only characterize normal forms, type equality can be seen as syntactic equality or equality modulo α-renaming depending on whether bound variables are implemented using DeBruijn indices or names.

One should note that in such a system, some terms, like `(\x . x x)`, are not even typable. In fact, LF is strongly normalizing [Harper et al., 1993]. While this is an interesting property to have for a data layer, a strongly normalizing language could not be Turing complete. This is the reason behind the less restricted form of recursion in Beluga's computational layer [Pientka and Dunfield, 2008].

Having type checking rules for substitutions might seems like overkill, because we only work with single substitutions (i.e. $[M/x]^*_A$) so far. In the next section however, they become first class objects, so this is why we reason about substitutions of greater length, which we also call simultaneous substitutions. In fact, such a simultaneous substitution is already present in the second rule for type checking substitution. As the objects of the substitution are defined in context

Ψ and the type A is defined in context Φ, we must first apply the substitution σ to A when type checking M in the premise of the rule.

3.2 Contextual modal type theory

Contextual modal type theory is basically LF with meta-variables [1] added in. These are a special class of variables whose (dependent) type also contains the context it depends on. We read $u{::}A[\Psi]$ as u has type A in context Ψ. We separate these declarations from the ordinary bound variables ones by putting them in a new (meta) context Δ.

Even if it would be possible to use only one context for both type of variables, they are often treated differently. Ordinary bound variables are used to represent abstractions in the object language while meta-variables are used to fill holes (missing arguments) in type reconstruction and are eventually instantiated (to bound variables or more complex terms). Since [Nanevski et al., 2008] already uses separate contexts, we keep up with the practice and avoid the need for an operation that would filter either one type of variable from a single context. We hope this will allow for a cleaner theory.

The syntactic additions are given next. Because contextual variables are defined in a separate context that does not depend on Ψ, they come with a substitution in order to make sense in the current (Ψ) context. We also add the notion of (simultaneous) substitution for meta-variables.

Neutral terms	R	$::=$	$\ldots \mid u[\sigma]$
Meta-contexts	Δ	$::=$	$\cdot \mid \Delta, u{::}A[\Phi]$
Contextual substitutions	ρ	$::=$	$\cdot \mid \rho, \hat{\Psi}.M/u$

The type checking judgements stay mostly the same: we have to thread through a Δ context. As before, we also need to add two new judgements for meta-contexts and contextual substitutions.

[1] Also called eigenvariables, existential variables or logic variables.

Types

$$\frac{\Psi \vdash A \Leftarrow \mathsf{type} \quad \Psi, x{:}A \vdash B \Leftarrow \mathsf{type}}{\Psi \vdash \Pi x{:}A.B \Leftarrow \mathsf{type}} \qquad \frac{\Psi \vdash P \Rightarrow \mathsf{type}}{\Psi \vdash P \Leftarrow \mathsf{type}}$$

Atomic types

$$\frac{\Sigma(\mathsf{a}) = K}{\Psi \vdash \mathsf{a} \Rightarrow K} \qquad \frac{\Psi \vdash P \Rightarrow \Pi x{:}A.K \quad \Psi \vdash M \Leftarrow A}{\Psi \vdash P\ M \Rightarrow [M/x]^k_A K}$$

Normal Terms

$$\frac{\Sigma(\mathsf{a}) = K}{\Psi \vdash \mathsf{a} \Rightarrow K} \qquad \frac{\Psi \vdash P \Rightarrow \Pi x{:}A.B \quad \Psi \vdash M \Leftarrow A}{\Psi \vdash P\ M \Rightarrow [M/x]^a_A B}$$

$$\frac{\Psi, x{:}A \vdash M \Leftarrow B}{\Psi \vdash \lambda x.M \Leftarrow \Pi x{:}A.B} \qquad \frac{\Psi \vdash R \Rightarrow P' \quad P =_\alpha P'}{\Psi \vdash R \Leftarrow P}$$

Neutral Terms

$$\frac{\Sigma(\mathsf{c}) = A}{\Psi \vdash \mathsf{c} \Rightarrow A} \qquad \frac{\Psi(x) = A}{\Psi \vdash x \Rightarrow A} \qquad \frac{\Psi \vdash R \Rightarrow \Pi x{:}A.B \quad \Psi \vdash M \Leftarrow A}{\Psi \vdash R\ M \Rightarrow [M/x]^a_A B}$$

Contexts

$$\frac{}{\vdash \cdot\ \mathsf{ctx}} \qquad \frac{\vdash \Psi\ \mathsf{ctx} \quad \Psi \vdash A \Leftarrow \mathsf{type}}{\vdash \Psi, x{:}A\ \mathsf{ctx}}$$

Substitutions

$$\frac{}{\Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Psi \vdash \sigma \Leftarrow \Phi \quad \Psi \vdash M \Leftarrow [\sigma]A}{\Psi \vdash \sigma, M/x \Leftarrow \Phi, x{:}A}$$

Figure 3.1: Typing/kinding rules for LF

Meta-variables

$$\frac{\Delta(u) = A[\Phi] \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash u[\sigma] \Rightarrow [\sigma]A}$$

Meta-contexts

$$\frac{}{\vdash \cdot \ \mathsf{mctx}} \qquad \frac{\vdash \Delta \ \mathsf{mctx} \quad \Delta; \Psi \vdash A \Leftarrow \mathsf{type}}{\vdash \Delta, u{::}A[\Psi] \ \mathsf{mctx}}$$

Contextual substitutions

$$\frac{}{\Delta' \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Delta' \vdash \rho \Leftarrow \Delta \quad \Delta'; [\![\rho]\!]\Psi \vdash M \Leftarrow [\![\rho]\!]A}{\Delta' \vdash \rho, \hat{\Psi}.M/u \Leftarrow \Delta, u{::}A[\Psi]}$$

Figure 3.2: Additional typing rules for CMTT

Meta-contexts	Contextual Substitutions
$\vdash \Delta \ \mathsf{mctx}$	$\Delta' \vdash \rho \Leftarrow \Delta$

The new rules for contextual variables, meta-contexts and contextual substitutions are given in figure 3.2. Note that because a contextual substitution element will make sense in Δ', we have to apply the rest of the (contextual) substitution to the context Ψ and type A when we check that it is well-typed. We also have to make sure that the bound variables that can appear in such a substitution element are the same as the ones in the context Ψ of the element's type. For this purpose, we keep around a list of names and we write it as $\hat{\Psi}$ (the notation is taken from [Nanevski et al., 2008]). This issue goes away when we use DeBruijn indices instead of names.

For reference purpose, we give the substitution rules for meta-variables in figure 3.3. These rules are simple recursions on the various syntactic forms and, as for bound variable substitutions, they are hereditary and will keep substituting whenever redexes are created. Again, we annotate the substitution with the type of the term (and the syntactical category of the term we are substituting

$$
\begin{array}{lll}
[\![\hat{\Psi}.M/u]\!]^n_{A[\Psi]}\lambda x.N & = \lambda x.[\![\hat{\Psi}.M/u]\!]^n_{A[\Psi]}N & \\
[\![\hat{\Psi}.M/u]\!]^r_{A[\Psi]}\mathsf{c} & = \mathsf{c} & \\
[\![\hat{\Psi}.M/u]\!]^r_{A[\Psi]}x & = x & \\
[\![\hat{\Psi}.M/u]\!]^r_{A[\Psi]}R\ N & = R'\ [\![\hat{\Psi}.M/u]\!]^n_{A[\Psi]}N & \text{if } [\![\hat{\Psi}.M/u]\!]^r_{A[\Psi]}R = R' \\
[\![\hat{\Psi}.M/u]\!]^r_{A[\Psi]}R\ N & = [N'/x]^n_{B[\Psi]}M' & \text{if } [\![\hat{\Psi}.M/u]\!]^r_{A[\Psi]}R = \lambda x.M' \text{ and} \\
 & & [\![\hat{\Psi}.M/u]\!]^n_{A[\Psi]}N = N' \\
[\![\hat{\Psi}.M/u]\!]^r_{A[\Psi]}u[\sigma] & = [[\![\hat{\Psi}.M/u]\!]^s_{A[\Psi]}\sigma]M & \\
[\![\hat{\Psi}.M/u]\!]^r_{A[\Psi]}v[\sigma] & = v[[\![\hat{\Psi}.M/u]\!]^s_{A[\Psi]}\sigma] &
\end{array}
$$

Figure 3.3: Contextual substitution

into), to reason about termination of the operation.

So far, we introduced the dependently typed lambda calculus that will serve as Beluga's object layer. We added in meta-variables and substitutions (contextual and bound variable) to provide the technology we will need to perform type reconstruction over Beluga's object layer. The next chapter will present an efficient way to implement the concepts introduced here.

Chapter 4

Beluga with explicit substitutions

In this chapter, we present the syntax and typing rules for an efficient implementation of Beluga. First, we delay the application of substitutions. Indeed, in the presence of dependent types, we might need to substitute terms into the type we are checking against. Delaying the substitution to the moment when we actually compare two base types allows us to save multiple pass over the term, an operation that can potentially be costly in the case of larger examples. We also use a spine formulation instead of the $(R\,M)$ application. Such a formulation gives us direct access to the head of a term and speed up unification (by failing early) and weak head normalization, an operation needed by delayed substitutions. This notation also simplifies the reconstruction phase when we have to check for pattern spines.

Finally, variables are implemented using DeBruijn [de Bruijn, 1972] indexing.

4.1 Syntax

As mentioned earlier, Beluga's object language is LF (dependently typed lambda calculus), augmented with meta-variables. Those meta-variables, however, are only accessible to the user in the computational layer. In the context of this thesis, their purpose will be to fill in implicit arguments during reconstruction. Because the computation layer also features explicit contexts and context variables, the substitutions of our data layer will have to reflect this. For example, in Beluga's setting, the contexts from the substitutions judgement $\Psi \vdash \sigma \Leftarrow \Phi$ could rely on context variables and the judgement will have to behave in a correct manner if we were to substitute a concrete context into it.

Moreover, pattern matching on terms with contexts variables brings the need for a new kind of contextual variable. Indeed, we will not be able to match against "any bound variable that might be contained in the context we will eventually substitute for context variable ψ" because meta-variables fail to capture the atomic nature of this matching and we do not know the exact name of the bound variables we should match against until we substitute an actual concrete context. These new variables will be called parameter variables and are written as $p[\sigma]$. Aside from their different behavior during matching, the behave similarly to meta-variables.

Therefore, the language we present here can be though of as "contextual modal type theory augmented with context variables and parameter variables". The addition is of interest mostly because we envision the data layer in the larger Beluga context, but they will not be necessary to the reconstruction process we describe in Chapter 5.

Finally, working with delayed substitutions will also bring the need for closures (a term together with a substitution). Because it is not always possible to keep those at the judgement level, we introduce syntactic categories for closures on types, terms and spines.

Because the type checking algorithm is bidirectional, we do not require typing annotation in λ-abstractions.

Kinds	K	$::=$	$\mathsf{type} \mid \Pi A.K$
Types	A, B	$::=$	$P \mid \Pi A.B \mid \mathsf{clo}(A, \sigma)$
Atomic types	P	$::=$	$\mathsf{a} \cdot S$
Normal terms	M, N	$::=$	$\lambda.M \mid H \cdot S \mid \mathsf{clo}(M, \sigma)$
Head	H	$::=$	$\mathsf{c} \mid x \mid u[\sigma] \mid p[\sigma]$
Spine	S	$::=$	$\mathsf{nil} \mid M\ S \mid \mathsf{clo}(S, \sigma)$
Substitutions	σ	$::=$	$\uparrow^{c,k} \mid \sigma, M \mid \sigma, H \mid \sigma, \mathsf{Undef}$
Context shifts	c	$::=$	$-\psi \mid \psi \mid 0$
Contexts	Ψ, Φ	$::=$	$\cdot \mid \psi \mid \Psi, A$
Meta-contexts	Δ	$::=$	$\cdot \mid \Delta, A[\Psi]$

As stated before, the bound variables (x) are represented as integers. For example, the λ-abstraction $\lambda x.\lambda y.yxx$ is represented as $\lambda.\lambda.011$ internally. For the same reason, we do without the names for all λ and Π binders. Similarly, when reasoning in a context Ψ, bound variable k will refer to the k'th element of Ψ (if it does not appear under a binder).

Syntax for substitutions is a variation on the explicit substitution defined in [Abadi et al., 1990]. This choice was made for efficiency reasons. Aside from being more compact, the "shift" representation will speed up common operations like composition and inversion. The shift operator is written $\uparrow^{c,k}$. Its argument(s) describe the context we are shifting over. For example, for a context Φ of length $n+k$ and a context Ψ, a prefix of Φ of length n, both containing no context variable, we would have something like this:

$$\Delta; \Phi \vdash \uparrow^{0,k} \Leftarrow \Psi$$

From this formulation, a natural question would be whether or not k can be negative, if we were to switch the domain and range, for example. In this case, the (inverse) substitution would not be defined. To avoid any misunderstanding,

we treat this as a special case: we use Undef's and restrict k to be positive. We use Undef's because it is sometimes meaningful to reason about a substitution that is not completely defined, in the case of inverse substitution, for example. However, trying to apply the undefined part of such a substitution would lead to an error (i.e. no rule covers that, see appendix A.1). Because contexts can also contain variables, the c argument will tell if we shift over a context variable or not. This is because these variables eventually get replaced and we have to adjust the substitution accordingly to the shape of the context that get substituted for the variable. For example, given $[\![\Psi/\psi]\!](\uparrow^{\psi,2})$ with Ψ of length 3 and not containing any context variable, the result would need to be $\uparrow^{0,5}$ to keep the theory coherent. Finally, shift substitutions can then be extended with normal terms or heads. Reasoning with DeBruijn indices, these extensions are understood to be positional. Therefore, the substitution σ, M will replace bound variable of index 0 with the term M.

We introduce right away an abbreviation for a frequently seen substitution:

Definition 1 (identity) $\mathsf{id} = \uparrow^{0,0}$

4.2 Judgements

We have the usual type checking (resp. type synthesizing) judgement for normal terms (resp. heads and spines). To that we add a judgement that checks that a substitution σ has domain Ψ and range Φ. The theory is completed by judgements for checking that a type is well-kinded and that kinds and contexts are well-formed. To keep the presentation simple, the details on the latter three are postponed to the appendix.

Having context variables around brings the question of whether we characterize them as we characterize terms with types. This is indeed the case, and when we substitute a context, we first check that it respects a given schema. This matter is primarily a computation level issue — and outside the scope of this thesis — so for the remainder of this thesis, we will only assume that contexts

$$
\begin{array}{lcll}
[\![\Psi/\psi]\!] \uparrow^{c,k} & = & \uparrow^{c,k} & \text{if } c \neq \pm\psi \\
[\![\cdot /\psi]\!] \uparrow^{\pm\psi,k} & = & \uparrow^{0,k} & \\
[\![\psi'/\psi]\!] \uparrow^{\psi,k} & = & \uparrow^{\psi',k} & \\
[\![\psi'/\psi]\!] \uparrow^{-\psi,k} & = & \uparrow^{-\psi',k} & \\
[\![(\Psi, A)/\psi]\!] \uparrow^{\psi,k} & = & [\![\Psi/\psi]\!] \uparrow^{\psi,k+1} & \\
[\![(\Psi, A)/\psi]\!] \uparrow^{-\psi,k} & = & ([\![\Psi/\psi]\!] \uparrow^{-\psi,k}), \mathsf{Undef} & \\
[\![\Psi/\psi]\!](\sigma, M) & = & [\![\Psi/\psi]\!]\sigma, [\![\Psi/\psi]\!]M & \\
[\![\Psi/\psi]\!](\sigma, H) & = & [\![\Psi/\psi]\!]\sigma, [\![\Psi/\psi]\!]H & \\
[\![\Psi/\psi]\!](\sigma, \mathsf{Undef}) & = & [\![\Psi/\psi]\!]\sigma, \mathsf{Undef} &
\end{array}
$$

Figure 4.1: Context variable substitution in substitution objects

are of a correct form whenever we replace a context variable. More details can be found in [Dunfield and Pientka, 2009].

In practice, whenever we check a LF signature, we do it in sequence and check that each constant refers to a well-formed kind or a well-kinded type. That means that the corresponding judgements are the ones bootstraping the whole process.

$$
\begin{array}{lll}
\Delta; \Psi & \vdash & (M, \sigma_1) \Leftarrow (A, \sigma_2) \\
\Delta; \Psi & \vdash & H \Rightarrow (A, \sigma) \\
\Delta; \Psi & \vdash & (S, \sigma_1) : (A, \sigma_2) \Rightarrow (P, \sigma) \\
\Delta; \Psi & \vdash & \sigma \Leftarrow \Phi
\end{array}
$$

4.3 Substitutions

We present right away some extension and results required by the new definition of the $\uparrow^{c,k}$ substitution, because substitutions play such a fundamental role in the typing rules.

4.3.1 Extension of context variable substitution

Since we keep track of context variables in our substitutions, we had to extend the notion of context variable substitution defined in [Pientka, 2008].

Coherently with our restriction that k must be positive, the substitution in a negative context shift leads to the unrolling of an Undef instead of a decrementing of k. Another remark is that we have to push the substitution into normal terms and heads since they can contain nested (bound variable) substitutions. The rules for context variable substitutions into normal terms and heads are simple recursion on their structure until we encounter a (bound variable) substitution, so it is omitted here. The rest of the rules, presented in figure 4.1, are what one would expect.

The following result states that this extension of context variable substitution does not change anything to the well-formedness of bound variables substitutions. Again, to stay completely formal, we have to make sure that the context we are substituting is of the correct form (schema).

Lemma 2 (Stability of substitution under context variable substitution)

Given a context Ψ'' *of the correct form to substitute for* ψ

If $\quad \Delta; \Psi' \vdash \sigma \Leftarrow \Psi$

then $\quad [\![\Psi''/\psi]\!]\Delta; [\![\Psi''/\psi]\!]\Psi' \vdash [\![\Psi''/\psi]\!]\sigma \Leftarrow [\![\Psi''/\psi]\!]\Psi$

Proof: By induction on the length of the derivation.

The interesting cases are the ones with the negative context shift. The substitution could possibly result in unrolling a series of `Undef`'s that we will want to check against some types. It is standard practice [Pierce, 1997] to understand bottom ($\perp$, the type of Undef) as a subtype of any type.

The full proof is given in appendix A.5.

4.3.2 Composition

Composition is a very common operation on substitutions. The typing rules we give next will depend on it. We write the composition of σ_1 and σ_2 as:

$$\sigma_1 \circ \sigma_2 = \sigma$$

In the typing rules, we often use the dot1 abbreviation. This abbreviation encompasses a simple intuition: whenever we substitute under a binder, we want to avoid capture and have no effect on the newly introduced variable. These are the reasons behind the $\uparrow^{0,1}$ and $\ldots, 1$ parts in the definition.

Definition 3 (dot1 σ)
If $\Psi' \vdash \sigma \Leftarrow \Psi$ *then* $\Psi', \mathit{clo}(A, \sigma) \vdash (\sigma \circ \uparrow^{0,1}), 1 \Leftarrow \Psi, x{:}A$ *and we write* dot1 σ *for* $(\sigma \circ \uparrow^{0,1}), 1$.

Back to composition, one must first remark that, because of the way substitutions are checked, they are always defined in the scope of *at most* one context variable. So if σ_1 and σ_2 would both refer to a context variable, it has to be the same for σ to be defined.
When we define composition, we maintain the following invariant:

Lemma 4 (Invariant of composition) *If* $\Psi \vdash \sigma_1 \Leftarrow \Psi_1$ *and* $\Psi_2 \vdash \sigma_2 \Leftarrow \Psi$ *then* $\Psi_2 \vdash \sigma_1 \circ \sigma_2 \Leftarrow \Psi_1$.

Proof: By induction on (see appendix A.5.1)

1. the structure of σ_1
2. the structure of σ_2
3. the value of the shift indices

The definition of composition in figure 4.2 is quite straightforward, if not for the abundance of base cases. We remark that composition is not defined for all possible syntactic cases. For example, in

$$\uparrow^{\psi,0} \circ (\sigma_2, H)$$

it would not make sense to substitute a head for the "first elements" of the range of $\uparrow^{\psi,0}$ since this range can only be the context variable ψ.

$$
\begin{array}{lcl}
\mathsf{id} \circ\ \sigma_2 & = & \sigma_2 \\
\\
\uparrow^{0,k} \circ\ \uparrow^{0,k'} & = & \uparrow^{0,k+k'} \\
\uparrow^{\psi,k} \circ\ \uparrow^{0,k'} & = & \uparrow^{\psi,k+k'} \\
\uparrow^{-\psi,k} \circ\ \uparrow^{0,k'} & = & \uparrow^{-\psi,k+k'} \\
\uparrow^{-\psi,0} \circ\ \uparrow^{\psi,k'} & = & \uparrow^{0,k'} \\
\uparrow^{\psi,0} \circ\ \uparrow^{-\psi,k'} & = & \uparrow^{0,k'} \\
\\
\uparrow^{c,k+1} \circ (\sigma_2, H) & = & \uparrow^{c,k} \circ \sigma_2 \\
\uparrow^{c,k+1} \circ (\sigma_2, \mathsf{Undef}) & = & \uparrow^{c,k} \circ \sigma_2 \\
\uparrow^{c,k+1} \circ (\sigma_2, M) & = & \uparrow^{c,k} \circ \sigma_2 \\
\\
(\sigma_1, M) \circ\ \sigma_2 & = & (\sigma_1 \circ\ \sigma_2, [\sigma_2]M) \\
(\sigma_1, H) \circ\ \sigma_2 & = & (\sigma_1 \circ\ \sigma_2, [\sigma_2]H) \\
(\sigma_1, \mathsf{Undef}) \circ\ \sigma_2 & = & (\sigma_1 \circ\ \sigma_2, \mathsf{Undef})
\end{array}
$$

Figure 4.2: Composition of substitutions

Lemma 5 (Associativity of composition) *When these compositions are defined, they are equal:*
$\sigma_1 \circ\ (\sigma_2 \circ\ \sigma_3) = (\sigma_1 \circ\ \sigma_2) \circ\ \sigma_3$

Proof: Induction on the structure of σ_1, then σ_2 (see appendix A.5.1).

4.3.3 Inversion

Another common operation on substitution is inversion. It is needed, amongst other things, in unification, which we rely on in the reconstruction algorithm we present in the next chapter.

Such an inversion however, is only fully defined for substitutions that map each variable of their domain to a different variable of their range. These substitutions are called *pattern substitutions* [Miller, 1991]. Indeed, if a substitution maps many different variables to the same image i, there won't be a unique choice for what we want to substitute for i in the inverse substitution.

To cover the greatest number of cases, we will assume a pre-phase of η-contraction on the substitution elements. This operation is introduced fully in the next

$$
\begin{array}{lcl}
(\sigma, x)_{\triangleleft,k} & = & (\sigma)_{\triangleleft,k+1} \\
(\sigma, \mathsf{Undef})_{\triangleleft,k} & = & (\sigma)_{\triangleleft,k+1} \\
(\uparrow^{c,k})_{\triangleleft,k'} & = & (\uparrow^{-c,k'})_{\triangleright,k} \\
(\sigma)_{\triangleright,0} & = & \sigma \\
(\sigma)_{\triangleright,k} & = & \begin{cases} (\sigma, i)_{\triangleright,k-1} & \text{if } \sigma_0 = \uparrow^{c,k'}, \ldots, x_{i+1}, k, x_{i-1}, \ldots \\ (\sigma, \mathsf{Undef})_{\triangleright,k-1} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 4.3: Inversion of substitutions

chapter, but intuitively, it contracts terms of the form

$$\lambda. \ldots . \lambda. f \cdot 1\ 2\ \ldots\ n$$

into the unique variable f, where n is the length of the λ prefix. That way, the substitution

$$\uparrow^{0,0}; \lambda x. f \cdot x$$

will have an inverse. We also allow the substitution to contain Undef's for reversibility of the operation. The rules in figure 4.3 give the right inverse.

Definition 6 (inverse) $\sigma^{-1} = (\sigma)_{\triangleleft,0}$

Inversion of a substitution happens in two phases. The first phase (we identify it with $\triangleleft$) counts the number of variables appearing before we hit the base ($\uparrow^{c,k}$) substitution. We then invert the variable count with the shift index k. The second ($\triangleright$) phase then creates a series of k terms, either relating them to their preimage under the original substitution σ_0 or setting them to Undef.
In practice, the operation of looking into σ_0 can be done in constant time by building up a global array in the first ($\triangleleft$) phase of inversion.

Lemma 7 *If* $\Psi \vdash \sigma \Leftarrow \Phi$ *then* $\Phi \vdash \sigma^{-1} \Leftarrow \Psi$

Proof: By induction on the size of Φ then on k (see appendix A.5.2).

$$\frac{\Delta; \Psi, \mathsf{clo}(A, \sigma_2) \vdash (M, \mathsf{dot1}\ \sigma_1) \Leftarrow (B, \mathsf{dot1}\ \sigma_2)}{\Delta; \Psi \vdash (\lambda.M, \sigma_1) \Leftarrow (\Pi A.B, \sigma_2)}$$

$$\frac{\Delta; \Psi \vdash H \Rightarrow (A, \sigma) \quad \Delta; \Psi \vdash (S, \sigma_1) : \mathsf{whnf}(A, \sigma) \Rightarrow (P', \sigma_1') \quad (P', \sigma_1') \simeq (P, \sigma_2)}{\Delta; \Psi \vdash (H \cdot S, \sigma_1) \Leftarrow (P, \sigma_2)}$$

Figure 4.4: Typing rules for normal terms

Lemma 8 *If σ is a pattern substitution, we have:*

1. $\sigma \circ \sigma^{-1} = \mathsf{id}$
2. $(\sigma^{-1})^{-1} = \sigma$

Proof: Straightforward from the definition, using an observation on pattern substitutions (see appendix A.5.2).

Lemma 9 *Composition and inversion terminate.*

Proof: By structural induction on σ_1 and σ_2 and σ respectively.
Composition iterates over the structure of σ_1 and then, over the structure of σ_2, both of which are finite. Therefore, composition terminates.
Inversion iterates over the structure of σ and then, over the shift index k, both of which are finite and therefore, inversion terminates. □

4.4 Typing rules

With the substitution properties established, we are now ready to present the typing rules for Beluga's object layer. We present first the rules for well-typed normal terms.
A notable feature of these rules is the use of closures to delay a substitution. The rules assume that (M, σ_1) and (A, σ_2) are in weak head normal form (defined in

$$\frac{\Sigma(\mathsf{c}) = A}{\Delta; \Psi \vdash \mathsf{c} \Rightarrow (A, \mathsf{id})} \qquad \frac{\Psi(x) = (A, \uparrow^{0,x})}{\Delta; \Psi \vdash x \Rightarrow (A, \uparrow^{0,x})} \qquad \frac{\Delta(u) = A[\Phi] \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash u[\sigma] \Rightarrow (A, \sigma)}$$

Figure 4.5: Typing rules for heads

appendix A.1), meaning that closures are pushed inside the term "far enough to allow us to see its shape". Concretely, whnf is a linear operation on the size of σ that checks if there are redexes at the outermost level and resolve them by composing delayed substitutions. As stated before, this makes for a more efficient implementation because the hereditary substitution potentially need to traverse the whole term. For clarity purpose, we only mention the operation when it's called in a way that is not obvious from the setting.

One might also note that neutral terms (i.e. terms that are not λ-abstractions) will always be of atomic type, so heads will always appear with a complete spine.

We use $\simeq$ to denote convertibility between two terms. This convertibility is basically η-convertibility under their respective substitutions, computed in a lazy way (in the implementation, we also have to ignore names and other informations kept around for error-printing purpose).

The rules for heads are quite straightforward. One must note that, since we work with indices, $\Psi(x)$ means "take the x-th element from context Ψ. Because the resulting type A does not depend on the newer declarations, we must shift it by the index of the variable x (i.e. the number of newer declarations). This is however not the case for meta-variables, as the substitution σ is a bound variable substitution and not a meta-variable substitution. Constants are internally referred to with unique identifiers.

Figure 4.7 presents the various base cases for the shift construct. One notable case is the `unroll` rule that creates a new bound variable in the substitution while incrementing the shift index. The restriction of this rule to cases without context shift is actually an important one. Otherwise, we would allow substitu-

$$\frac{}{\Delta; \Psi \vdash (\mathsf{nil}, \sigma_1) : (P, \sigma_2) \Rightarrow (P, \sigma_2)}$$

$$\frac{\Delta; \Psi \vdash (M, \sigma_1) \Leftarrow (A, \sigma_2) \quad \Delta; \Psi \vdash (S, \sigma_1) : \mathsf{whnf}(B, (\sigma_2; \mathsf{clo}(M, \sigma_1))) \Rightarrow (P, \sigma)}{\Delta; \Psi \vdash (M\ S, \sigma_1) : (\Pi A.B, \sigma_2) \Rightarrow (P, \sigma)}$$

Figure 4.6: Typing rules for spines

$$\frac{}{\cdot \vdash \uparrow^{0,0} \Leftarrow \cdot} \quad \frac{}{\psi \vdash \uparrow^{0,0} \Leftarrow \psi} \quad \frac{}{\psi \vdash \uparrow^{\psi,0} \Leftarrow \cdot} \quad \frac{}{\cdot \vdash \uparrow^{-\psi,0} \Leftarrow \psi}$$

$$\frac{\Psi \vdash \uparrow^{c,k} \Leftarrow \cdot}{\Psi, A \vdash \uparrow^{c,k+1} \Leftarrow \cdot} \quad \frac{\Psi \vdash \uparrow^{c,k} \Leftarrow \psi}{\Psi, A \vdash \uparrow^{c,k+1} \Leftarrow \psi} \quad \frac{\Psi' \vdash \uparrow^{0,k+1}, k+1 \Leftarrow \Psi, A \quad k \geq 0}{\Psi' \vdash \uparrow^{0,k} \Leftarrow \Psi, A}\ \texttt{unroll}$$

$$\frac{\Psi' \vdash \sigma \Leftarrow \Psi \quad \Psi' \vdash H \Rightarrow (A', \sigma') \quad (A', \sigma') \simeq (A, \sigma)}{\Psi' \vdash \sigma, H \Leftarrow \Psi, A} \quad \frac{\Psi' \vdash \sigma \Leftarrow \Psi}{\Psi' \vdash \sigma, \mathsf{Undef} \Leftarrow \Psi, A}$$

$$\frac{\Psi' \vdash \sigma \Leftarrow \Psi \quad \Psi' \vdash (M, \mathsf{id}) \Leftarrow (A, \sigma)}{\Psi' \vdash \sigma, M \Leftarrow \Psi, A}$$

Figure 4.7: Typing rules for substitutions

tions of this form (variable names written in for explicitness)

$$\psi, x_k{:}A_k, \ldots, x_1{:}A_1 \vdash \uparrow^{\psi,0} \Leftarrow x_k{:}A_k, \ldots, x_1{:}A_1$$

While this seems sensible at first — each variable x_i get shifted by 0 and thus maps to x_i — substituting a context for ψ could introduce new declarations to the *right* of the k declarations already there. In the absence of the domain/range, it would be impossible to infer k when substituting (for a context variable) into the substitution, and therefore, keep the shift index meaningful. The solution is to force the substitution to always be unrolled — $(\uparrow^{\psi,k}, k, \ldots, 1)$ instead of $\uparrow^{\psi,0}$ in this case — at the cost of space and some efficiency.
As expected, the following holds:

Lemma 10 $\Delta; \Psi \vdash \mathsf{id} \Leftarrow \Psi$

Proof: By induction on the length of Ψ.

Theorem 11 *Type checking for object-level Beluga terminates.*

Proof: By induction on the structure of the various syntactic objects using lemma 9 and the termination of weak head normalization (based on the termination of hereditary substitution).
Type-checking for object level Beluga is syntax directed and relies on operations of convertibility (which is also syntax directed) and weak head normalization (which is linear on the size of the substitution). We could use the `unroll` (and the equivalent rule in convertibility) to expand a substitution indefinitely, but we only do it when no other case applies. From these facts, it is easy to see that type-checking for object level Beluga does terminates. □

Chapter 5

Type reconstruction

In this chapter, we present an algorithm to perform type reconstruction on Beluga's user-level syntax. Since the internal syntax uses a DeBruijn index representation for variables, a first phase is to translate bound variables from names to indices. On the other hand, the algorithm uses a named representation for free variables and meta-variables are implemented via references. They are replaced by bound variables in the very last phase of reconstruction.

We keep up with the practice introduced in the last chapter of only characterizing fully applied neutral terms. As we do not want to impose the burden of writing η-expanded forms on the user, our algorithm will have to reconstruct these, if needed. This is useful in practice as it allows the user to write terms more compactly when the explicit form of a function is not wanted. Also, we always create meta-variables of atomic type; this can be achieved with lowering. This way, we will not have to re-normalize when we substitute a term for a meta-variable that is associated with a pattern substitution.

We follow the same presentation as before, introducing the syntax, judgements and rules, together with supporting judgements for η-conversion and lowering. We then present various supporting lemmas and conclude the chapter with the soundness statement for our algorithm.

5.1 Syntax

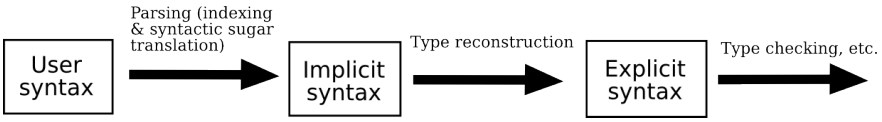
The user-level syntax for the object layer, as found in the signature of a Beluga program, is omitted as it is not very interesting. For readability, bound variables are referred by name and non-dependent products are written using the arrow (`->`). For example, an explicit version of the composition operator from Chapter 2:

```
@ : {A:obj} {B:obj} {C:obj} mor B C -> mor A B -> mor A C.
```

would be parsed as

$$@ : \Pi \texttt{obj}.\Pi \texttt{obj}.\Pi \texttt{obj}.\Pi \texttt{mor}\ 2\ 1.\Pi \texttt{mor}\ 4\ 3.\texttt{mor}\ 5\ 3$$

This shows the necessity of having a user syntax even if the translation to implicit syntax is quite straightforward.
The following picture shows the interpretation process:



The syntax we present here is the output of the indexing phase, so the binders are already written without names and we omit details about syntactic sugar. We call this one the *implicit syntax*. One thing to note is that we allow the user to write "holes", written "_", in the place of a normal term. This allows the user to benefit from the reconstruction by omitting some information when it can be reconstructed.

Implicit syntax

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

Next, we present the *internal* or *explicit* syntax for Beluga's object layer. This syntax adds in meta-variables and free variables. In our setting, meta-variables will always be of base type, so they will not appear with a spine. Υ is an unordered context that describes meta-variables. Free variables are very similar to bound variables, except for the fact that they will be referred by name and will appear in the unordered context Φ. Those contexts are unordered because, even if free variables and meta-variable will be introduced in an ordered manner, we will use unification to instantiate some of the meta-variables, and thus the ordering might need to change. Implicit arguments and free variables are eventually made explicit at the constant level in the explicit syntax (we reconstruct one constant type/kind at a time). For this reason the explicit signature features a number i attached to each constant that will give us the number of its arguments that are implicit.

Explicit syntax

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

5.2 Judgements

For the sake of clarity, we introduce the reconstruction judgements with omitted Δ contexts. Context variables and meta-variables cannot happen in the signature, but when we will reconstruct object level terms embedded into a Beluga function, they could be present. This is the reason why the shift substitution ($\uparrow$) still mentions contexts variables in the syntax. However, having bound meta-variables (as added in the computation layer) would also mean that the meta-variables we instantiate for fill in missing arguments could depend on variables in Δ. Thus, their type should be generalized to something of the form $A[\Psi; \Delta]$ and these meta-meta-variables (or meta^2 variables) would live on a higher level than the bound meta-variables accessible to the user. Because we concentrate on the object layer, we make the simplification that Δ will always be empty and therefore, our meta-variables will only have to depend on the ordinary bound variable context Ψ. Therefore, the rules for meta-variables as

stated in the previous chapter stay valid.

For our presentation needs, we only mention reconstruction rules for normal terms and spines. Other rules follow a similar pattern and are given in appendix B.3. Taking the judgement for normal term as an example, we read it like this: in contexts $\Upsilon_1; \Phi_1; \Psi$, implicit term m checks against explicit type (A, σ) and reconstructs to M, updating contexts Υ_1 and Φ_1 in the process and generating the contextual (i.e. meta-variables) substitution ρ. The judgement for spines follows the same pattern.

The third judgement is called when we encounter a constant that has implicit arguments. It will accomplish the same thing as the judgement for spines, but will also introduce i meta-variables before doing so, to create an explicit spine of the correct form. These meta-variables will have the possibility to be instantiated later to make sure the whole term is well-typed and the ones left at the end of reconstruction will then be abstracted at the constant level as implicit arguments in their turn.

The fourth judgement will be used when we need to infer a type for a free variable when we first encounter it. Because we can only reconstruct the type of a free variable if s is a pattern spine (i.e. a spine composed of distinct bound variables), the judgement makes that assumption. To cover more cases, it is enough for s to be η-convertible to a pattern spine.

$$
\begin{array}{llcl}
\Upsilon_1; \Phi_1; \Psi & \vdash & m & \Leftarrow (A, \sigma) \,/_\rho\, (\Upsilon_2; \Phi_2; M) \\
\Upsilon_1; \Phi_1; \Psi & \vdash & s : (A, \sigma_1) & \Leftarrow (P, \sigma_2) \,/_\rho\, (\Upsilon_2; \Phi_2; S) \\
\Upsilon_1; \Phi_1; \Psi & \vdash^i & s : (A, \sigma_1) & \Leftarrow (P, \sigma_2) \,/_\rho\, (\Upsilon_2; \Phi_2; S) \\
\Upsilon_1; \Phi_1; \Psi & \vdash & s & \Leftarrow (P, \sigma) \,/\, ((A, \sigma); S)
\end{array}
$$

Throughout reconstruction, we maintain the invariant that contexts, contextual substitutions and types are well-formed:

$$
\begin{array}{llll}
 & \vdash_{\Phi_1} & \Upsilon_1 & \mathsf{mctx} \\
\Upsilon_1 & \vdash & \Phi_1 & \mathsf{fctx} \\
\Upsilon_1; \Phi_1 & \vdash & \Psi & \mathsf{ctx} \\
\Upsilon_1; \Phi_1; \Psi & \vdash & (A, \sigma) & \Leftarrow \mathsf{type} \\
\Upsilon_1; \Phi_1; \Psi & \vdash & (P, \sigma) & \Leftarrow \mathsf{type}
\end{array}
$$

$$
\begin{array}{llll}
 & \vdash_{\Phi_2} & \Upsilon_2 & \mathsf{mctx} \\
\Upsilon_2 & \vdash & \Phi_2 & \mathsf{fctx} \\
\Upsilon_2 & \vdash_{\Phi_2} & \rho & \Leftarrow \Upsilon_1
\end{array}
$$

Proof: By induction on the reconstruction derivation (appendix B.4.1).

An interesting fact to note here is that ρ's are ultimately generated by unification and could introduce circular dependencies amongst meta-variables. This is the main reason behind the unordering of the Υ and Φ contexts. Consequently, contextual substitutions ρ are not defined incrementally (see appendix): each of their element must make sense in the substitution's range Υ_2. That way, it is possible to define a typing judgement for these substitutions, it will still make sense to apply them — they will produce well-typed objects — and we are assured that the operation will terminate, even in the presence of circular dependencies.

5.3 η-conversion

Our reconstruction algorithm allows the user to write terms in either η-expanded or η-contracted form. For example, for a bound variable x of functional type, a user could write only x instead of

$$\lambda y_n. \ldots . \lambda y_1. x \cdot y_n \ \ldots \ y_1$$

$$\frac{\Psi, \mathsf{clo}(A, \sigma) \vdash^{i+1} \mathsf{expand}\ y\ :\ (B, \mathsf{dot1}\ \sigma)\ /\ M}{\Psi \vdash^{i} \mathsf{expand}\ y\ :\ (\Pi A.B, \sigma)\ /\ \lambda.M}$$

$$\frac{\Psi \vdash y \Rightarrow (A, \uparrow^{0,y}) \quad \Psi \vdash^{i} \mathsf{expand}\ (A, \uparrow^{0,y}) \Rightarrow (P, \sigma)\ /\ S}{\Psi \vdash^{i} \mathsf{expand}\ y\ :\ (P, \sigma)\ /\ y \cdot S}$$

$$\frac{\Psi \vdash^{0} \mathsf{expand}\ i\ :\ (A, \sigma_1)\ /\ M \quad \Psi \vdash^{i-1} \mathsf{expand}\ (B, (\sigma_1; \mathsf{clo}(M, \mathsf{id}))) \Rightarrow (P, \sigma_2)\ /\ S}{\Psi \vdash^{i} \mathsf{expand}\ (\Pi A.B, \sigma_1) \Rightarrow (P, \sigma_2)\ /\ M\ S}$$

$$\frac{}{\Psi \vdash^{0} \mathsf{expand}\ (P, \sigma) \Rightarrow (P, \sigma)\ /\ \mathsf{nil}}$$

Figure 5.1: Rules for η-expansion of terms

avoiding to write redundant and cumbersome code.

Another use for η-equivalent forms is when we have to infer the type of a free variable (fig. 5.6). Because we only infer free variable types from a pattern spine, it is worth it to check if a variable was written in its η-expanded form, to cover more cases.

The judgements for η-expansion and contraction are:

$$\begin{aligned} \Upsilon; \Phi; \Psi \quad &\vdash^{i} \mathsf{expand}\ x\ :\ (A, \sigma)\ /\ M \\ \Upsilon; \Phi; \Psi \quad &\vdash^{i} \mathsf{expand}\ (A, \sigma_1) \Rightarrow (P, \sigma_2)\ /\ S \end{aligned}$$

$$\begin{aligned} &\mathsf{contract}_i\ m\ /\ x \\ &\mathsf{contract}_i\ s \end{aligned}$$

The idea behind the judgements for expansion is to count the length of the Π prefix and then create λ prefix and a spine of the correct form and length. We keep up with the explicit substitution notation and, for clarity purpose, the rules do not feature Υ and Φ since they will be constant. The rules are given in figure 5.1.

η-contraction works the other way, counting the length of the λ prefix and recursively applying contraction to the spine before checking that it is of the correct form/length. If so, contraction returns the head (bound variable) of

$$\frac{\mathsf{contract}_{i+1}\ m\ /\ x}{\mathsf{contract}_i\ \lambda.m\ /\ x} \qquad \frac{\mathsf{contract}_i\ s}{\mathsf{contract}_i\ x \cdot s\ /\ x}$$

$$\frac{}{\mathsf{contract}_0\ \mathsf{nil}} \qquad \frac{\mathsf{contract}_0\ m\ /\ i \quad \mathsf{contract}_{i-1}\ s}{\mathsf{contract}_i\ (m\ s)}$$

Figure 5.2: Rules for η-contraction of terms

the λ-abstraction body. One notes that, contrarily to expansion, contraction is not type-directed. This is because we do not have access to the bound variable before we actually do the contraction. This has the consequence of "accepting" a term of the form

$$\lambda y_2.\lambda y_1.x \cdot y_2\ y_1\ \mathsf{nil}$$

for a variable x of type $\Pi A.P$. However, the correctness of the contraction can be verified afterwards, by checking that the lambda prefix length is less or equal to the length of the Π prefix of the variable's type. This is easily done in the implementation (i.e. by doing a second pass), but to avoid putting this additional burden on the presentation, we will not mention it further. The rules for η-contraction are given in figure 5.2.

One important property of η-expansion is that it produces well-typed terms. The proof is quite straightforward and is found in the appendix.

Lemma 12 (Expansion produces well-typed term)

1. *If* $(\Upsilon;\Phi) \mid \Psi \vdash^i \mathsf{expand}\ x\ :\ (A,\sigma)\ /\ M$
   then $\Upsilon;\Phi;\Psi \vdash (M,\mathsf{id}) \Leftarrow (A,\sigma)$.

2. *If* $(\Upsilon;\Phi) \mid \Psi \vdash^i \mathsf{expand}\ (A,\sigma_1) \Rightarrow (P,\sigma_2)\ /\ S$
   then $\Upsilon;\Phi;\Psi \vdash (S,\mathsf{id}) : (A,\sigma_1) \Rightarrow (P,\sigma_2)$.

Proof: Induction on the expansion derivation (see appendix B.4.2).

Another important property is that, whenever we have an η-expanded form, we can substitute it for one occurrence of the original variable and the type

checking judgement will still stand. We prove it for normal terms and spines, but it is true for all judgements.
Note that the theorem is stated using eager substitution rules, so the contexts in these rules are not exactly the same that the ones we use in the η-expansion judgements because the later could contain closures. The former are therefore normalized version of the output contexts and we use them as such in the proof, even if we do not introduce the additional annotation in the statement.

Lemma 13 (η-expansion stable under substitution)
If $\Psi \vdash^i \mathsf{expand}\ x\ :\ (B, \sigma)\ /\ N$ *with* $\Psi(x) = (B, \uparrow^{0,x})$ *and*

1. $\Upsilon; \Phi; \Psi \vdash M \Leftarrow A$
2. $\Upsilon; \Phi; \Psi \vdash S : A \Rightarrow P$

then

1. $\Upsilon; \Phi; \Psi \vdash M' \Leftarrow A$
2. $\Upsilon; \Phi; \Psi \vdash M \Leftarrow A'$
3. $\Upsilon; \Phi; \Psi \vdash S : A' \Rightarrow P$
4. $\Upsilon; \Phi; \Psi \vdash S' : A \Rightarrow P$

Where M', A' *and* S' *are the original terms where one occurrence of* x *has been replaced by* N*.*

Proof: By induction on the typing derivation (appendix B.4.2)

Corollary 14 *Result 13 stands for the substitution of any number of occurrences of* x*.*

Proof: By multiple application of lemma 13.

$$\frac{u \notin \Upsilon}{\mathsf{lower}(\Upsilon; \Phi; \Psi \vdash (P, \sigma)) = (u[\mathsf{id}]; u{::}\mathsf{clo}(P, \sigma)[\Psi])}$$

$$\frac{\mathsf{lower}(\Upsilon; \Phi; \Psi, \mathsf{clo}(A, \sigma) \vdash (B, \mathsf{dot1}\ \sigma)) = (M; u{::}P[\Psi'])}{\mathsf{lower}(\Upsilon; \Phi; \Psi \vdash (\Pi A.B, \sigma)) = (\lambda.M; u{::}P[\Psi'])}$$

Figure 5.3: Rules for lowering

5.4 Lowering

To simplify the unification process, we would like to only compare meta-variables of atomic type. Indeed, comparing two meta-variables applied to spines of possibly different length would be awkward. This will also have the advantage of avoiding to re-normalize whenever we instantiate a meta-variable, as we know it will not create redexes if the meta-variable was associated with a pattern substitution.

To achieve this, we will always create meta-variables of an atomic type. We call lowering the type directed process in which we create a new meta-variable (of atomic type) under a λ-prefix that will match the Π-prefix of the given type. We write it:

$$\mathsf{lower}(\Upsilon; \Phi; \Psi \vdash (A, \sigma)) = (M, u{::}P[\Psi'])$$

The rules are straightforward and given in figure 5.3.

Lemma 15 (Lowering)
If $\mathsf{lower}(\Upsilon; \Phi; \Psi \vdash (A, \sigma)) = (M; u{::}P[\Psi'])$
then $\Upsilon, u{::}P[\Psi']; \Phi; \Psi \vdash (M, \mathsf{id}) \Leftarrow (A, \sigma)$.

Proof: By structural induction on A (appendix B.4.2).

5.5 Reconstruction rules

The reconstruction rules for normal terms feature multiple things. Compared to the rules written in the previous chapter, the rule for constants will also

feature an additional element i that gives the number of implicit arguments that are to be reconstructed. The user can also ask explicitely for a term to be reconstructed by writing an underscore (_), which is also translated to a meta-variable in the internal syntax.

When we encounter a new free variable, we reconstruct its type by examining the attached spine. The check that s is a pattern spine is to be understood modulo η-conversion. We then prune the reconstructed type with the empty substitution to make sure it does not depend on any bound variable in Ψ. After that, we update the free variable context Φ. When the same free variable is encountered later, it is treated in a way similar to a bound variable. We do not need to shift the type we get from the free variable context Φ according to the (bound variable) context Ψ because we made sure this type was closed before inserting it in Φ.

Finally, the rule for checking a neutral term against a functional type will simply create the appropriate λ-prefix and spine in a process akin to η-expansion, to make sure a neutral term is always of atomic type.

The rules for λ-abstractions and bound variables are pretty much what one would expect.

The reconstruction rules for spines are straightforward: we unify the types in the base case and we do a recursion otherwise.

Synthesizing a type from a pattern spine is a little more complex. Although we only return the expected type in the base case, the recursive case has more steps to it. First, we contract the current spine element to a bound variable. If this step fails, then we are not in the presence of a pattern spine and we cannot reconstruct a type. This could be done in a previous separate check, but the principle would be the same (and efficiency would suffer by having to traverse the spine twice). As stated before, given how our contraction is not type directed, we could make a wrong term right in this process. This can be avoided by doing a second, type directed contraction, once we get the type of the contracted variable. We then expand the variable back, to ensure that all variables of functional type will be fully η-expanded in the internal syntax. One

$$\frac{\Upsilon_1; \Phi_1; \Psi, \mathsf{clo}(A, \sigma) \vdash m \Leftarrow (B, \mathsf{dot1}\ \sigma)\ /_\rho\ (\Upsilon_2; \Phi_2; M)}{\Upsilon_1; \Phi_1; \Psi \vdash \lambda.m \Leftarrow (\Pi A.B, \sigma)\ /_\rho\ (\Upsilon_2; \Phi_2; \lambda.M)}$$

$$\frac{\begin{array}{c} h' = [\uparrow^{0,n}]h \quad s' = [\uparrow^{0,n}]s \quad \sigma_{i-1} = \mathsf{dot1}\ \sigma_i \quad A'_i = \mathsf{clo}(A_i, \sigma_i) \\ \Upsilon_1; \Phi_1; \Psi, A'_n, \ldots, A'_1 \vdash h' \cdot (s' @ ((n \cdot \mathsf{nil}) \ldots (1 \cdot \mathsf{nil})\mathsf{nil})) \Leftarrow (P, \sigma_0)\ /_\rho\ (\Upsilon_2; \Phi_2; R) \end{array}}{\Upsilon_1; \Phi_1; \Psi \vdash h \cdot s \Leftarrow (\Pi A_n \ldots A_1.P, \sigma_n)\ /_\rho\ (\Upsilon_2; \Phi_2; \lambda. \ldots \lambda.R)}$$

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

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

$$\frac{\begin{array}{c} X \notin \Phi_1 \quad s \text{ is a pattern spine} \\ \Upsilon_1; \Phi_1; \Psi \vdash s \Leftarrow (P, \sigma)\ /\ ((A, \sigma_1); S) \quad \Upsilon_1; \Phi_1; \Psi \vdash (A, \sigma_1) \mid [\cdot]^{-1} \Rightarrow (\Upsilon_2; \rho) \end{array}}{\Upsilon_1; \Phi_1; \Psi \vdash X \cdot s \Leftarrow (P, \sigma)\ /_\rho\ (\Upsilon_2; [\![\rho]\!]\Phi_1, X : [\![\rho]\!]\mathsf{clo}(A, \sigma_1); [\![\rho]\!](X \cdot S))}$$

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

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

Figure 5.4: Reconstruction for normal terms

$$\frac{\Upsilon_1; \Phi_1; \Psi \vdash (\mathsf{a} \cdot S', \sigma_1) \doteq (\mathsf{a} \cdot S, \sigma_2)\ /\ (\rho; \Upsilon_2)}{\Upsilon_1; \Phi_1; \Psi \vdash \mathsf{nil} : (\mathsf{a} \cdot S', \sigma_1) \Leftarrow (\mathsf{a} \cdot S, \sigma_2)\ /_\rho\ (\Upsilon_2; [\![\rho]\!]\Phi_1; \mathsf{nil})}$$

$$\frac{\begin{array}{c} \Upsilon_1; \Phi_1; \Psi \vdash m \Leftarrow (A, \sigma_1)\ /_{\rho_1}\ (\Upsilon_2; \Phi_2; M) \\ \Upsilon_2; \Phi_2; [\![\rho_1]\!]\Psi \vdash s : ([\![\rho_1]\!]B, ([\![\rho_1]\!]\sigma_1; M)) \Leftarrow [\![\rho_1]\!](P, \sigma_2)\ /_{\rho_2}\ (\Upsilon_3; \Phi_3; S) \end{array}}{\Upsilon_1; \Phi_1; \Psi \vdash m\ s : (\Pi A.B, \sigma_1) \Leftarrow (P, \sigma_2)\ /_{\rho_1 \circ \rho_2}\ (\Upsilon_3; \Phi_3; [\![\rho_2]\!]M\ S)}$$

Figure 5.5: Reconstruction for spines

$$\frac{}{\Upsilon_1; \Phi_1; \Psi \vdash \mathsf{nil} \Leftarrow (P, \sigma) \mathbin{/} ((P, \sigma); \mathsf{nil})}$$

$$\frac{\begin{array}{l}\mathsf{contract}_0\ m \mathbin{/} x \quad \Psi(x) = (A, \uparrow^{0,x}) \quad \Psi \vdash^0 \mathsf{expand}\ x\ :\ (A, \uparrow^{0,x}) \mathbin{/} M \\ \Upsilon_1; \Phi_1; \Psi \vdash s \Leftarrow (P, \sigma) \mathbin{/} ((B, \sigma''); S) \quad \sigma' = \uparrow^{0,x+1}, 1, x, \ldots, 1\end{array}}{\Upsilon_1; \Phi_1; \Psi \vdash m\ s \Leftarrow (P, \sigma) \mathbin{/} ((\Pi\mathsf{clo}(A, \uparrow^{0,x}).\mathsf{clo}(B, \sigma'' \circ\ \uparrow^{0,1} \circ \sigma'), \mathsf{id}); M\ S)}$$

Figure 5.6: Synthesize type from pattern spine

must note that contraction and expansion are not exactly dual here, even in the presence of well-typed terms. In fact, for a variable x of type $\Pi A_k. \ldots . \Pi A_1. P$, we allow the user to write anything between x and $\lambda \ldots \lambda . x$ where the λ prefix is of length less or equal to k.

The type we return in the end is a Π abstraction over the type of x. While we know that x will be in Ψ, we have no guarantee that x will of index 1. To make sure of that we first shift (B, σ'') by 1, to avoid capture by the Π binder. We then adjust the x variable to be of index 1 (it will be the only one), leaving all the other unchanged. This is the idea behind the seemingly complicated σ'.

The last set of rules is for the reconstruction of missing arguments. We simply recurse over the argument i to fill in the right number of holes. In the process, we use the lowering judgement to make sure meta-variables are of base type, so the holes are filled with either a meta-variable or an equivalent form with a λ prefix and longer substitution.

We also note that the substitution we get from the recursive call will refer to u, so we have to remove this part from substitution we return to preserve the reconstruction invariant.

We are now ready to state the soundness result for our reconstruction algorithm.

Theorem 16 (Soundness of reconstruction)

1. *If* $\Upsilon_1; \Phi_1; \Psi \vdash m \Leftarrow (A, \sigma) \mathbin{/}_\rho (\Upsilon_2; \Phi_2; M)$
   then $\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash M \Leftarrow [\![\rho]\!][\sigma]A$.

2. *If* $\Upsilon_1; \Phi_1; \Psi \vdash^i s : (A, \sigma_1) \Leftarrow (P, \sigma_2) \mathbin{/}_\rho (\Upsilon_2; \Phi_2; S)$

$$\frac{\Upsilon_1; \Phi_1; \Psi \vdash s : (A, \sigma_1) \Leftarrow (P, \sigma_2) \;/_\rho\; (\Upsilon_2; \Phi_2; S)}{\Upsilon_1; \Phi_1; \Psi \vdash^0 s : (A, \sigma_1) \Leftarrow (P, \sigma_2) \;/_\rho\; (\Upsilon_2; \Phi_2; S)}$$

$$\frac{\begin{array}{l}\mathsf{lower}(\Upsilon_1; \Phi; \Psi \vdash (A, \sigma_1)) = (M, u{::}Q[\Psi']) \\ \Upsilon_1, u{::}Q[\Psi']; \Phi_1; \Psi \vdash^{i-1} s : (B, (\sigma_1; M)) \Leftarrow (P, \sigma_2) \;/_\rho\; (\Upsilon_2; \Phi_2; S)\end{array}}{\Upsilon_1; \Phi_1; \Psi \vdash^i s : (\Pi A.B, \sigma_1) \Leftarrow (P, \sigma_2) \;/_{\rho \setminus (R/u)}\; (\Upsilon_2; \Phi_2; [\![\rho]\!]M\ S)}$$

Figure 5.7: Reconstruction of missing arguments

then $\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash S : [\![\rho]\!][\sigma_1]A \Rightarrow [\![\rho]\!][\sigma_2]P$.

3. *If* $\Upsilon_1; \Phi_1; \Psi \vdash s : (A, \sigma_1) \Leftarrow (P, \sigma_2) \;/_\rho\; (\Upsilon_2; \Phi_2; S)$
   then $\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash S : [\![\rho]\!][\sigma_1]A \Rightarrow [\![\rho]\!][\sigma_2]P$.

4. *If* $\Upsilon_1; \Phi_1; \Psi \vdash s \Leftarrow (P, \sigma_2) \;/\; ((A, \sigma_1); S)$
   and s *is a pattern spine*
   then $\Upsilon_1; \Phi_1; \Psi \vdash S : [\sigma_1]A \Rightarrow [\sigma_2]P$
   and $\Upsilon_1; \Phi_1; \Psi \vdash [\sigma_1]A \Leftarrow \mathsf{type}$.

Moreover, in the first 3 cases, we have:
$[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ *and* $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ *and* $\Upsilon_2 \vdash \Phi_2\ \mathsf{fctx}$ *and* $\vdash_{\Phi_2} \Upsilon_2\ \mathsf{mctx}$

Proof: By structural induction on the reconstruction judgment.
The underlying idea is that a reconstructed term will always be type correct under the updated contexts. The result is written in terms of eager substitution because our algorithm introduces η-expanded form. That would prevent us from comparing two types for equality or even convertibility. Fortunately, the problem goes away when the substitutions are applied either because hereditary substutition collapses η-expanded form (in the case of non nil spine) or simply because the η-expansion produces well-typed terms (lemma 12).
The full proof can be found in appendix B.4.3, together with the typing rules for eager substitution form. The theorem holds for types and kinds as well, but this part is omitted because it follows a pattern very similar to what is already there and brings no additional understanding.

Another note is that the contexts we get from reconstruction are again not exactly the same that we use in the judgements in eager substitution form. The later are normalized version (we use them as such in the proof).
The theorem in itself is quite simple: it goes over the various reconstruction judgements and says that the reconstructed terms will be well-typed. In addition, when we generate the type for a free variable from a pattern spine, this type will be well-kinded. Finally, the meta-variable substitution ρ that we generate will preserve the well-formedness of contexts. The subset relation between $[\![\rho]\!]\Phi_1$ and Φ_2 exist because new free variables could be collected in the process.

5.6 Abstraction

Once a term has been reconstructed, we have to get rid of the context it depends on. This is done in a phase we call abstraction. The idea is to unroll the Υ and Φ contexts as a Π prefix for the reconstructed terms and kinds (signatures consist of constants annotated with their type/kind).
The first issue with abstraction is that Υ and Φ are not ordered. In fact, our implementation does not deal with these contexts explicitely. Instead we use side effects and global data. With that in mind, the first step of abstraction is to traverse a term to first collect all the variables of Υ and Φ. We use a simple algorithm, similar to a depth-first search, that will only collect a variable if we already have all it's dependencies. We keep track of variables that we encountered but did not collect yet to avoid running into infinite loop in case of circular dependencies. If we cannot find such an ordering, we fail.
We then do a second pass in which we transform free variables and meta-variables to bound variables, indexing them according to their position in the contexts. In the case of meta-variables, we also have to transform the substitution to a spine that will be attached to the bound variable. This is just a matter of unrolling the substitution if needs be, η-expanding bound variables in the process. This step needs to be type directed to know when to stop unrolling a shift substitution.

Once this is done, the rest is just a matter of transforming the (bound variable) contexts Υ and Ψ into a Π prefix for a given type or kind. We will also keep the number of implicit arguments (i.e. the number of variables in the two contexts) with the constant's type/kind.

Chapter 6

Related work

To allow its users to use them efficiently, most dependently typed systems include some form of type reconstruction. Understanding this problem is an important step towards incorporating the technology into mainstream languages. However, because it is often seen as syntactic sugar, the theoretical implication of type reconstruction are sometimes overlooked. In this chapter, we explore how type reconstruction is implemented in various other proposal out there.

6.1 Cayenne

The main design choice behind Cayenne [Augustsson, 1998] is the mixing of dependent types with full recursion. This allows for a "simpler" language (terms and types are not distinct), but leads to undecidability of type checking. The prototype has some kind of type reconstruction which is described as a "syntactic device without any deep semantic properties". While the syntax tried to stay close to LEGO's, the reconstruction algorithm is described as "quite weak" and needing to move to "a more powerful method that introduces meta-variables [...] and unification" [Augustsson, 1998]. However, the undecidability of type checking prevents reconstruction from having any guarantee on which term can be reconstructed. The language does not support HOAS either.

6.2 LEGO family

As mentioned before, early ideas on type reconstruction came from Pollack's work on LEGO [Pollack, 1990]. This work introduces the notion of placeholders, or meta-variables. These placeholders then get instantiated in a process called "mixed prefix unification" [Miller, 1992] that takes into account the bound variables into which scope the meta-variable appear, not unlike the CMTT approach of writing a meta-variable together with a closure. He then describes an implicit function space that is used to infer the placeholders automatically. However, it differs from our approach in that the user is always allowed to supply implicit arguments explicitly by a syntactic annotation.

Another difference is that the implicit function declaration needs to be annotated with its input type, where we do infer the type of free variables. However, LEGO's polymorphism allows meta-variables to stand in for types, and this achieves a result that is not far from our use of free variables in terms of syntactic redundancy.

A soudness claim is made about the reconstruction process in LEGO but the algorithm itself is not given.

Epigram's implicit syntax is strongly inspired by LEGO's. In [McBride, 2005], it is claimed to be different, in that it has a *separate* implicit function space, yet they impose a requirement to segregate this function space when doing reconstruction. It is not clear what this implicit function space achieves exactly.

A more general remark is that even if Epigram is presented as a functional programming language, it still provides its user with the weaker structural recursion [McBride, 2000] and doesn't support reasonning over HOAS encoding, thus is less expressive than full Beluga.

In turn, Agda's [Norell, 2007] implicit syntax is mostly based on Epigram's, with a few syntactic variations as far as dependent types and reconstruction are concerned. As its predecessors, it is polymorphic and has a limited form of recursion to keep type checking decidable.

6.3 Twelf family

Being the first implementation of the logical framework LF, Twelf approach to type reconstruction — sketched in [Pfenning, 1991] — was the main inspiration of our work. Like in our approach, Twelf allows its user to write constants which type (or kind) contains free variables and, from that point on, reconstruct the implicit quantifier and fills in missing arguments, with meta-variables, in the later use of the constant. It does not allow the user to write implicit arguments explicitly.

Twelf implementation differs from our approach in that they allow meta-variables to stand for types, instead of just terms. Because of that, the actual algorithm is decoupled in two phases: the first one determines the simply typed form of the terms and, in a second pass, unification is used to instantiate meta-variables and ensure that a term is dependently well-typed. While our implementation might seem weaker than Twelf on that point, we implemented all the examples (that did not use definitions or constraint solvers) of the Twelf library and the net result was that we had to provide annotations in just a few cases. For example, where Twelf had

```
({x: _} conv (M x) (M' x))
```

we had to write

```
({x: term _} conv (M x) (M' x))
```

which was not that much of a burden, since this does not make these annotations heavier than the ones in a simply typed systems. Some other differences are:

- We only characterize β-normal forms while Twelf allows the user to write terms containing redexes.
- We do not yet support arbitrary annotations.

- We do not support definitions yet.

Delphin [Poswolsky, 2008] is similar to Beluga in spirit in that it proposes a dependently typed language suitable for programming. It also uses LF as an object layer, but does not re-implement reconstruction. Instead, it only reuses the (mostly undocumented) reconstruction algorithm from Twelf, directly.

Celf [Schack-Nielsen and Schürmann, 2008] is an implementation of the the concurrent logical framework CLF [Watkins et al., 2002] which is itself a conservative extension of LF. As such, their prototype is similar in functionality to ours in that it can read Twelf signatures, supports HOAS and performs reconstruction over dependent types. However, Celf's focus is different from Beluga's in that it is primarily meant to model concurrent systems. At the moment, it does not support much meta-level functionalities such as coverage and termination checking or writing proofs in functional style. The high level ideas behind their reconstruction are similar to ours but they do not provide a detailed description of their algorithm.

SASyLF [Aldrich et al., 2008] is another LF implementation which purpose is to bridge the gap between paper proofs and proof assistants. Because of that, its syntax is very different from Twelf's and closer to what a proof would look like on paper. They also use a functional style instead of Twelf's logic style to write their proofs, thus being more similar to Beluga in that regard. The prototype provides functionalities like holes and meta-variables but HOAS is not fully supported. Since LF is used for data representation, there must be some kind of (dependent) type reconstruction. However, it is not clear if and how it is actually done as they give no technical description of their system. They also make hypothetical judgements more evident by working with explicit contexts. Their Java implementation being a completely new implementation of LF, it would be interesting to compare to other proposals out there, but because their syntax is so different, it is difficult to do in practise as the Twelf library of examples is not easily (and not yet) translated in SASyLF.

6.4 Nuprl

Nuprl is yet another dependently typed lambda-calculus. Although the original system description predates Pollack's work, newer version could have integrated his ideas if they had seen fit. The system has meta-variables and allows the user to omit some parts and use tactics to reconstruct some terms. However, because type checking and type inference in Nuprl is "highly heuristic" [Felty et al., 1998] and in general undecidable [Kreitz, 2002], there cannot be many guarantees on whether and when a term can be reconstructed.

6.5 DML/ATS

DML's [Xi, 1998] approach to integrating dependent types in functional programing is opposite to Cayenne's in that the developers were quite conservative in choosing to index their types only with natural numbers. This made for a well-behaved system lacking in expressivity. DML's successor ATS [Xi, 2004] allows dependent types to be indexed by any data of simple type. While the indexing domain is bigger than in DML, it is still quite limited compared to ours in terms of expressivity. Also, this limitation has the effect of keeping the types quite small in size and thus, the advantages of type reconstruction would not be as significant in their setting. They do not seem to have any reconstruction for now.

6.6 Coq

Coq is a proof assistant based on the calculus of inductive constructions [Paulin-Mohring, 1993]. As such, it features dependent types and polymorphism. The approach they take to the reconstruction of dependent types is similar to ours in that they reconstruct the language to an explicit one. Similarly to LEGO's, their reconstruction algorithm works in two passes: one to infer placeholders (meta-variables) and one to instanciate them. It is shown to be sound in [Saïbi,

1999].

Their approach differs from ours however in the way they think about their meta-variables. While they identify the need to know the context in which a meta-variable is introduced, they do not formalize it in the way CMTT does. This has the result of making their algorithms (unification, type inference) a little less straightforward.

Another difference is how they treat free variables. The system they present [Team, 2009] can infer automatically which arguments are reconstructible. The task of defining the behaviour of different kinds of implicit variables is left to the user. For example, argument N of the constant `cons`

```
cons:  N:nat nat -> list N -> list (s N).
```

can be inferred from the second argument, so Coq will identify it as implicit. However, the decision to infer N or not when `cons` is used will depend on which option were set by the user. Also, since Coq allows constants to be partially applied, `cons` could be understood as the constant itself or the constant applied to the implicit argument N. The burden of solving all these technical questions is left to the user. Another example would be how to order the implicit arguments. In our setting, this task is solved by abstraction while in Coq, the user always has to at least name the "free variables", so the system knows what their order should be.

Chapter 7

Conclusion

The goal of this thesis was to give a formal description on how to implement the data layer of a dependently typed programming language. To motivate our work, we first presented an example that showed how one could use dependent types in a programming language (Chapter 2). Then we presented some theoretical background (Chapter 3). The first contribution of this thesis was to describe how to adapt ideas on spine calculus, explicit substitutions and De Bruijn indexing to efficiently implement a dependently typed functional language that supports reasonning over HOAS. The second contribution was to present an algorithm, together with proofs of correctness, to reconstruct explicit dependent types from a lightweight, user level, language.

In the course of our presentation, we compared our system to various other proposals to show that our work is part of a genuine interest for dependent types, while showing that we pushed the understanding of these one step further.

We tested our implementation on Twelf library of examples (the ones that did not use definitions or constraint solvers). Dimitri Kirchner implemented timing functions and our implementation showed similar timing results as Twelf for the examples we tested it on. The Beluga prototype is available for download at

`http://complogic.cs.mcgill.ca/beluga/`

7.1 Future work

The focus of this thesis was on Beluga's data layer. There is some work left to be done in order to have a fully working prototype, most of which is related to the computational layer.

Computational level reconstruction Type reconstruction issues arise in the computational layer when we pattern match over data with the "case" construct. We identify three of them:

1. **Synthesize branch type** In a dependently typed setting, the different branch patterns could have different types. This is the reason in [Pientka and Dunfield, 2008] for branch annotation. However, it is often the case that we can reconstruct these annotation.

2. **Infer type for pattern variables** Most of the time, the types of (free) pattern variables can be reconstructed too.

3. **Reconstruct missing arguments in patterns** As before, we reconstruct missing arguments for the data we examine. Arguments are filled with meta^2 variables because, as opposed to arguments in the signatures, they could depend on bound meta-variables present in the Δ context.

While most of these issues are addressed in the prototype, the theory backing them is still missing.

Schema checking Beluga features variables that stand for context. Those variables are characterized by schemas that specify what are the possible shapes for the context elements. The computational layer will have to feature a mechanism for checking a context against a given schema.

Coverage checking In the presence of dependent types, checking that a Beluga function covers all of its domain is not trivial. There is still some work to do in that area [Dunfield and Pientka, 2009].

Error messages While this is not a theoretical issue, in order to be useful our prototype will have to include good error messages. We are confident that our theory will provide the adequate framework to do so, but for the moment, the message themselves are quite minimalist.

Bibliography

[Abadi et al., 1990] Abadi, M., Cardelli, L., Curien, P.-L., and Lèvy, J.-J. (1990). Explicit substitutions. In *17th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM Press.

[Aldrich et al., 2008] Aldrich, J., Simmons, R. J., and Shin, K. (2008). SASyLF: An Educational Proof Assistant for Language Theory. In *International Workshop on Functional and Declarative Programming in Education (FDPE '08)*, pages 31–40. ACM Press.

[Altenkirch et al., 2005] Altenkirch, T., McBride, C., and McKinna, J. (2005). Why dependent types matter. Manuscript, available online.

[Augustsson, 1998] Augustsson, L. (1998). Cayenne—a language with dependent types. In *3rd International Conference on Functional Programming (ICFP '98)*, pages 239–250. ACM Press.

[Cervesato and Pfenning, 2003] Cervesato, I. and Pfenning, F. (2003). A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688.

[de Bruijn, 1972] de Bruijn, N. G. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392.

[Dunfield and Pientka, 2009] Dunfield, J. and Pientka, B. (2009). Case analysis of higher-order data. In *International Workshop on Logical Frameworks and*

Meta-Languages: Theory and Practice (LFMTP'08), volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 69–84. Elsevier.

[Felty et al., 1998] Felty, A. P., Howe, D. J., and Stomp, F. A. (1998). Protocol verification in Nuprl. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, volume 1427, pages 428–439. Springer.

[Harper et al., 1993] Harper, R., Honsell, F., and Plotkin, G. (1993). A framework for defining logics. *Journal of the ACM*, 40(1):143–184.

[Kreitz, 2002] Kreitz, C. (2002). *The Nuprl Proof Development System, Version 5: Reference Manual and User's Guide*. Available online.

[Licata and Harper, 2009] Licata, D. R. and Harper, R. (2009). Positively dependent types. In *PLPV '09: Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, pages 3–14, New York, NY, USA. ACM Press.

[Luther, 2001] Luther, M. (2001). More on implicit syntax. In Gore, R., Leitsch, A., and Nipkow, T., editors, *Proceedings of the First International Joint Conference on Automated Reasoning, Siena, Italy*, Lecture Notes in Artificial Intelligence (LNAI) 2083, pages 386–400. Springer.

[McBride, 2000] McBride, C. (2000). *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, University of Edinburgh. Technical Report ECS-LFCS-00-419.

[McBride, 2005] McBride, C. (2005). The epigram prototype: a nod and two winks. Manuscript, available online.

[Miller, 1991] Miller, D. (1991). A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536.

[Miller, 1992] Miller, D. (1992). Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358.

[Nanevski et al., 2008] Nanevski, A., Pfenning, F., and Pientka, B. (2008). Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49.

[Necula, 1997] Necula, G. C. (1997). Proof-carrying code. In *24th Annual Symposium on Principles of Programming Languages(POPL'97)*, pages 106–119, Paris, France. ACM Press.

[Norell, 2007] Norell, U. (2007). *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.

[Paulin-Mohring, 1993] Paulin-Mohring, C. (1993). Inductive Definitions in the System Coq – Rules and Properties. In Bezem, M. and Groote, J. F., editors, *Proceedings of the 1st International Conference on Typed Lambda Calculi and Applications (TLCA '93)*, volume 664, pages 328–345. Springer.

[Pfenning, 1991] Pfenning, F. (1991). Logic programming in the LF logical framework. In Huet, G. and Plotkin, G., editors, *Logical Frameworks*, pages 149–181. Cambridge University Press.

[Pfenning and Schürmann, 1999] Pfenning, F. and Schürmann, C. (1999). System description: Twelf — a meta-logical framework for deductive systems. In Ganzinger, H., editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer.

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

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

[Pierce, 1997] Pierce, B. C. (1997). Bounded quantification with bottom. Technical Report TR492, Computer Science Department, Indiana University.

[Pollack, 1990] Pollack, R. (1990). Implicit syntax. In Huet, G. and Plotkin, G., editors, *LF '90: Informal Proceedings of the 1st Workshop on Logical Frameworks*, pages 421–434, Antibes.

[Poswolsky, 2008] Poswolsky, A. B. (2008). *Functional Programming with Logical Frameworks*. PhD thesis, Department of Computer Science, Yale University.

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

[Saïbi, 1999] Saïbi, A. (1999). *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories.* PhD thesis, Université Paris 6.

[Schack-Nielsen and Schürmann, 2008] Schack-Nielsen, A. and Schürmann, C. (2008). Celf – A Logical Framework for Deductive and Concurrent Systems (System Description). In *4th International Joint Conference on Automated Reasoning (IJCAR '08)*, pages 320–331. Springer.

[Team, 2009] Team, T. C. D. (2009). *The Coq Proof Assistant Reference Manual Version 8.2*. Available online.

[Watkins et al., 2002] Watkins, K., Cervesato, I., Pfenning, F., and Walker, D. (2002). A Concurrent Logical Framework I: Judgments and Properties. Tech-

nical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University.

[Xi, 1998] Xi, H. (1998). *Dependent types in practical programming*. PhD thesis, Carnegie Mellon University.

[Xi, 2004] Xi, H. (2004). Applied type system. In *TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer.

Appendix A

Object level Beluga

A.1 Weak head normal form

$$
\begin{array}{ll}
\mathsf{whnf}(\lambda.M, \sigma) & = (\lambda.M, \sigma) \\
\mathsf{whnf}(\mathsf{clo}(M, \sigma_1), \sigma_2) & = \mathsf{whnf}(M, \sigma_1 \circ\ \sigma_2) \\
\mathsf{whnf}(u[\sigma_1], \sigma_2) & = (u[\sigma_1 \circ\ \sigma_2], \mathsf{id}) \\
\mathsf{whnf}(p[\sigma_1], \sigma_2) & = (p[\sigma_1 \circ\ \sigma_2], \mathsf{id}) \\
\mathsf{whnf}(\mathsf{c} \cdot S, \sigma) & = (\mathsf{c} \cdot \mathsf{clo}(S, \sigma), \mathsf{id}) \\
\mathsf{whnf}(X \cdot S, \sigma) & = (X \cdot \mathsf{clo}(S, \sigma), \mathsf{id}) \\
\mathsf{whnf}(x \cdot S, \sigma) & = \begin{cases} \mathsf{whnfRedex}((M, \mathsf{id}), (S, \sigma)) & \text{if } [\sigma]x = M \\ (H \cdot \mathsf{clo}(S, \sigma), \mathsf{id}) & \text{if } [\sigma]x = H \end{cases} \\
\mathsf{whnfRedex}(H \cdot S, \sigma_1)(\mathsf{nil}, \sigma_2) & = \mathsf{whnf}(H \cdot S, \sigma_1) \\
\mathsf{whnfRedex}(\lambda.M, \sigma_1)(N\ S, \sigma_2) & = \mathsf{whnfRedex}(M, (\sigma_1, \mathsf{clo}(N, \sigma_2)))(S, \sigma_2) \\
\mathsf{whnfRedex}(M, \sigma_1)(\mathsf{clo}(S, \sigma_2'), \sigma_2) & = \mathsf{whnfRedex}(M, \sigma_1)(S, \sigma_2' \circ\ \sigma_2) \\
\mathsf{whnfRedex}(\mathsf{clo}(M, \sigma_1'), \sigma_1)(S, \sigma_2) & = \mathsf{whnfRedex}(M, \sigma_1' \circ\ \sigma_1)(S, \sigma_2)
\end{array}
$$

We describe here the weak head normalization. This operation is used in the context where we apply substitutions lazily. Instead of applying a substitution on a whole term, weak head normalization only applies it on the outmost syntactic level and pushes the substitution down one level by the mean of closures. This way, we are able to see the shape of a term and the operation is efficient (linear on the size of the substitution instead of the size of the term). Weak

head normalization uses the notion of composition defined earlier in Chapter 4. We mention here the weak head normal form of a term with a free variable (see Chapter 5), as it will be needed later in reconstruction.

A.2 Convertibility

Next we give the rules for convertibility, needed for type checking. These rules assume that we are working with weak head normal forms, so closures are not mentioned explicitly and bound variables are convertible only if they are equal. We use the `dot1` notation defined in Chapter 4.

Kinds

$$\frac{(A_1, \sigma_1) \simeq (A_2, \sigma_2) \quad (K_1, \mathsf{dot1}\ \sigma_1) \simeq (K_2, \mathsf{dot1}\ \sigma_2)}{(\Pi A_1.K_1, \sigma_1) \simeq (\Pi A_2.K_2, \sigma_2)} \qquad \frac{}{(\mathsf{type}, \sigma_1) \simeq (\mathsf{type}, \sigma_2)}$$

Types

$$\frac{(A_1, \sigma_1) \simeq (A_2, \sigma_2) \quad (B_1, \mathsf{dot1}\ \sigma_1) \simeq (B_2, \mathsf{dot1}\ \sigma_2)}{(\Pi A_1.B_1, \sigma_1) \simeq (\Pi A_2.B_2, \sigma_2)} \qquad \frac{(S_1, \sigma_1) \simeq (S_2, \sigma_2)}{(\mathsf{a} \cdot S_1, \sigma_1) \simeq (\mathsf{a} \cdot S_2, \sigma_2)}$$

Normal terms

$$\frac{(M_1, \mathsf{dot1}\ \sigma_1) \simeq (M_2, \mathsf{dot1}\ \sigma_2)}{(\lambda.M_1, \sigma_1) \simeq (\lambda.M_2, \sigma_2)} \qquad \frac{(H_1, \sigma_1) \simeq (H_2, \sigma_2) \quad (S_1, \sigma_1) \simeq (S_2, \sigma_2)}{(H_1 \cdot S_1, \sigma_1) \simeq (H_2 \cdot S_2, \sigma_2)}$$

Heads

$$\frac{}{(\mathsf{c}, \sigma_1) \simeq (\mathsf{c}, \sigma_2)} \qquad \frac{}{(x, \sigma_1) \simeq (x, \sigma_2)} \qquad \frac{\sigma_1' \circ\ \sigma_1 \simeq \sigma_2' \circ\ \sigma_2}{(u[\sigma_1'], \sigma_1) \simeq (u[\sigma_2'], \sigma_2)}$$

Spines

$$\frac{}{(\mathsf{nil}, \sigma_1) \simeq (\mathsf{nil}, \sigma_2)} \qquad \frac{(M_1, \sigma_1) \simeq (M_2, \sigma_2) \quad (S_1, \sigma_1) \simeq (S_2, \sigma_2)}{(M_1\ S_1, \sigma_1) \simeq (M_2\ S_2, \sigma_2)}$$

Substitutions

$$\frac{\sigma_1 \simeq \sigma_2}{(\sigma_1; M) \simeq (\sigma_2, M)} \qquad \frac{\sigma_1 \simeq \sigma_2}{(\sigma_1, H) \simeq (\sigma_2, H)} \qquad \frac{\sigma_1 \simeq \sigma_2}{(\sigma_1, \mathsf{Undef}) \simeq (\sigma_2, \mathsf{Undef})}$$

$$\frac{}{\uparrow^{c,k} \simeq \uparrow^{c,k}} \qquad \frac{\uparrow^{c,k+1}, k+1 \simeq \sigma}{\uparrow^{c,k} \simeq \sigma} \qquad \frac{\uparrow^{c,k+1}, k+1 \simeq \sigma}{\sigma \simeq \uparrow^{c,k}}$$

Lemma 17

If $(O_1, \sigma_1) \simeq (O_2, \sigma_2)$ *then* $[\sigma_1]O_1 = [\sigma_2]O_2$ *for* $O \in \{K, A, M, H, S\}$

Proof: By induction on the convertibility derivation.

Most of the cases are trivial and follow the simple pattern "inversion – induction hypothesis – structural equality under a substitution".

Base cases happen when we compare constants, nil spines, base kind type and bound variables. For the first three, they are only convertible if they are equal, according to the rules, so the conclusion holds. In the bound variable case, we must note that we are working with weak head normal forms, so the respective substitutions are *already* applied, thus the conclusion holds.
For two meta-variables to be convertible, their respective substitutions must be equal, so the conclusion holds by the fact that substitution is a deterministic operation. □

A.3 Judgements

$$\begin{aligned}
\Delta; \Psi &\vdash K \ \mathsf{kind} \\
\Delta; \Psi &\vdash (A, \sigma) \Leftarrow \mathsf{type} \\
\Delta; \Psi &\vdash (S, \sigma_1) : (K, \sigma_2) \Rightarrow \mathsf{type}
\end{aligned}$$

$$\begin{aligned}
\Delta &\vdash \Psi \Leftarrow W \\
&\vdash \Delta \ \mathsf{mctx}
\end{aligned}$$

A.4 Typing rules

Kinds

$$\frac{}{\Delta; \Psi \vdash \mathsf{type}\ \mathsf{kind}} \qquad \frac{\Delta; \Psi \vdash (A, \mathsf{id}) \Leftarrow \mathsf{type} \quad \Delta; \Psi, A \vdash K\ \mathsf{kind}}{\Delta; \Psi \vdash \Pi A.K\ \mathsf{kind}}$$

Types

$$\frac{\Sigma(a) = K \quad \Delta; \Psi \vdash (S, \sigma_1) : (K, \mathsf{id}) \Rightarrow \mathsf{type}}{\Delta; \Psi \vdash (a \cdot S, \sigma_1) \Leftarrow \mathsf{type}}$$

$$\frac{\Delta; \Psi \vdash (A, \sigma) \Leftarrow \mathsf{type} \quad \Delta; \Psi, \mathsf{clo}(A, \sigma) \vdash (B, \mathsf{dot1}\ \sigma) \Leftarrow \mathsf{type}}{\Delta; \Psi \vdash (\Pi A.B, \sigma) \Leftarrow \mathsf{type}}$$

Type Spines

$$\frac{}{\Delta; \Psi \vdash (\mathsf{nil}, \sigma_1) : (\mathsf{type}, \sigma_2) \Rightarrow \mathsf{type}}$$

$$\frac{\Delta; \Psi \vdash (M, \sigma_1) \Leftarrow (A, \sigma_2) \quad \Delta; \Psi \vdash (S, \sigma_1) : \mathsf{whnf}(K, (\sigma_2, \mathsf{clo}(M, \sigma_1))) \Rightarrow \mathsf{type}}{\Delta; \Psi \vdash (M\ S, \sigma_1) : (\Pi A.K, \sigma_2) \Rightarrow \mathsf{type}}$$

Contexts

$$\frac{}{\Delta \vdash \cdot\ \mathsf{ctx}} \qquad \frac{\Delta \vdash \Psi\ \mathsf{ctx} \quad \Delta; \Psi \vdash A \Leftarrow \mathsf{type}}{\Delta \vdash \Psi, A\ \mathsf{ctx}}$$

Meta Contexts

$$\frac{}{\vdash \cdot\ \mathsf{mctx}} \qquad \frac{\vdash \Delta\ \mathsf{mctx} \quad \Delta \vdash \Psi\ \mathsf{ctx} \quad \Delta; \Psi \vdash (A, \mathsf{id}) \Leftarrow \mathsf{type}}{\vdash \Delta, A[\Psi]\ \mathsf{mctx}}$$

A.5 Proofs

Lemma 2 (Stability of substitution under context variable substitution)

Assuming stability of other type checking judgements and convertibility under context variable substitution, and given a context Ψ'' of

the correct form to substitute for ψ, we have:

If $\quad \Delta; \Psi' \vdash \sigma \Leftarrow \Psi$

then $\quad [\![\Psi''/\psi]\!]\Delta; [\![\Psi''/\psi]\!]\Psi' \vdash [\![\Psi''/\psi]\!]\sigma \Leftarrow [\![\Psi''/\psi]\!]\Psi$

Proof: By induction on the length of the first derivation. We prove the case where ψ happens at least in one of Ψ', σ and Ψ (other cases are trivially true). Note that we omit Δ contexts for clarity and we write "cvar" as an abbreviation for "context variable"

Case $\cdot \vdash \uparrow^{0,0} \Leftarrow \cdot$

$[\![\Psi''/\psi]\!]\cdot \vdash [\![\Psi''/\psi]\!] \uparrow^{0,0} \Leftarrow [\![\Psi''/\psi]\!]\cdot$ — def. of cvar substitution

Case $\psi \vdash \uparrow^{0,0} \Leftarrow \psi$

$[\![\Psi''/\psi]\!]\psi \vdash \uparrow^{0,0} \Leftarrow [\![\Psi''/\psi]\!]\psi$ — lemma 10

$[\![\Psi''/\psi]\!]\psi \vdash [\![\Psi''/\psi]\!] \uparrow^{0,0} \Leftarrow [\![\Psi''/\psi]\!]\psi$ — def. of cvar substitution

Case $\psi \vdash \uparrow^{\psi,0} \Leftarrow \cdot$

Subcase $\Psi'' = \cdot, A_k, \ldots, A_1$

$\Psi'' \vdash \uparrow^{0,k} \Leftarrow \cdot$ — typing rule

$[\![\Psi''/\psi]\!]\psi \vdash [\![\Psi''/\psi]\!] \uparrow^{\psi,0} \Leftarrow \cdot$ — def. of cvar substitution

Subcase $\Psi'' = \psi', A_k, \ldots, A_1$

$\psi', A_k, \ldots, A_1 \vdash \uparrow^{\psi',k} \Leftarrow \cdot$ — typing rule

$[\![\Psi''/\psi]\!]\psi \vdash [\![\Psi''/\psi]\!] \uparrow^{\psi,0} \Leftarrow \cdot$ — def. of cvar substitution

Case $\cdot \vdash \uparrow^{-\psi,0} \Leftarrow \psi$

Subcase $\Psi'' = \cdot, A_k, \ldots, A_1$

$\cdot \vdash \uparrow^{0,0}, \underbrace{\mathsf{Undef}, \ldots, \mathsf{Undef}}_{k} \Leftarrow \cdot, A_k, \ldots, A_1$ — typing rule

$\cdot \vdash ([\![\cdot/\psi]\!] \uparrow^{-\psi,0}), \underbrace{\mathsf{Undef}, \ldots, \mathsf{Undef}}_{k} \Leftarrow \cdot, A_k, \ldots, A_1$ — def. of cvar substitution

$\cdot \vdash [\![\Psi''/\psi]\!] \uparrow^{-\psi,0} \Leftarrow \Psi''$ — def. of cvar substitution

$[\![\Psi''/\psi]\!]\cdot \vdash [\![\Psi''/\psi]\!] \uparrow^{-\psi,0} \Leftarrow [\![\Psi''/\psi]\!]\psi$ — def. of cvar substitution

Subcase $\Psi'' = \psi', A_k, \ldots, A_1$

$\cdot \vdash \uparrow^{-\psi',0}, \underbrace{Undef, \ldots, \mathsf{Undef}}_{k} \Leftarrow \psi', A_k, \ldots, A_1$ typing rule

$\cdot \vdash (\llbracket \psi'/\psi \rrbracket \uparrow^{-\psi,0}), \underbrace{\mathsf{Undef}, \ldots, \mathsf{Undef}}_{k} \Leftarrow \psi', A_k, \ldots, A_1$ def. of cvar substitution

$\cdot \vdash \llbracket \Psi''/\psi \rrbracket \uparrow^{-\psi,0} \Leftarrow \Psi''$ def. of cvar substitution

$\llbracket \Psi''/\psi \rrbracket \cdot \vdash \llbracket \Psi''/\psi \rrbracket \uparrow^{-\psi,0} \Leftarrow \llbracket \Psi''/\psi \rrbracket \psi$ def. of cvar substitution

Case $\Psi, A \vdash \uparrow^{\psi,k+1} \Leftarrow \cdot$

$\Psi \vdash \uparrow^{\psi,k} \Leftarrow \cdot$ inversion

$\llbracket \Psi''/\psi \rrbracket \Psi \vdash \llbracket \Psi''/\psi \rrbracket \uparrow^{\psi,k} \Leftarrow \llbracket \Psi''/\psi \rrbracket \cdot$ i.h.

let $\Psi'' = \psi', A_l, \ldots, A_1$

$\llbracket \Psi''/\psi \rrbracket \Psi \vdash \uparrow^{\psi',k+l} \Leftarrow \llbracket \Psi''/\psi \rrbracket \cdot$ def. of cvar substitution

$\llbracket \Psi''/\psi \rrbracket (\Psi, A) \vdash \uparrow^{\psi',k+l+1} \Leftarrow \llbracket \Psi''/\psi \rrbracket \cdot$ typing rule

$\llbracket \Psi''/\psi \rrbracket (\Psi, A) \vdash \llbracket \Psi''/\psi \rrbracket \uparrow^{\psi,k+1} \Leftarrow \llbracket \Psi''/\psi \rrbracket \cdot$ def. of cvar substitution

Case $\Psi, A \vdash \uparrow^{c,k+1} \Leftarrow \psi$

$\Psi \vdash \uparrow^{c,k} \Leftarrow \psi$ inversion

$\llbracket \Psi''/\psi \rrbracket \Psi \vdash \llbracket \Psi''/\psi \rrbracket \uparrow^{c,k} \Leftarrow \llbracket \Psi''/\psi \rrbracket \psi$ i.h.

Subcase $c = 0$

$\llbracket \Psi''/\psi \rrbracket \Psi \vdash \uparrow^{0,k} \Leftarrow \llbracket \Psi''/\psi \rrbracket \psi$ def. of cvar substitution

$\llbracket \Psi''/\psi \rrbracket (\Psi, A) \vdash \uparrow^{0,k+1} \Leftarrow \llbracket \Psi''/\psi \rrbracket \psi$ typing rule

$\llbracket \Psi''/\psi \rrbracket (\Psi, A) \vdash \llbracket \Psi''/\psi \rrbracket \uparrow^{0,k+1} \Leftarrow \llbracket \Psi''/\psi \rrbracket \psi$ def. of cvar substitution

Subcase $c = -\psi$

Subsubcase $\Psi'' = \cdot, A_l, \ldots, A_1$

$\Psi \vdash \uparrow^{0,k}, \underbrace{\mathsf{Undef}, \ldots, \mathsf{Undef}}_{l} \Leftarrow \cdot, A_l, \ldots, A_1$

def. of cvar substitution (Ψ cannot contain a context variable)

$\Psi \vdash \uparrow^{0,k} \Leftarrow \cdot$ typing rule

$\Psi, A \vdash \uparrow^{0,k+1} \Leftarrow \cdot$ typing rule

$\Psi, A \vdash \uparrow^{0,k+1}, \underbrace{\mathsf{Undef}, \ldots, \mathsf{Undef}}_{l} \Leftarrow \cdot, A_l, \ldots, A_1$ typing rule

$[\![\Psi''/\psi]\!](\Psi, A) \vdash [\![\Psi''/\psi]\!] \uparrow^{-\psi,k+1} \Leftarrow [\![\Psi''/\psi]\!]\psi$ def. of cvar substitution

Subsubcase $\Psi'' = \psi', A_l, \ldots, A_1$

$\Psi \vdash \uparrow^{-\psi',k}, \underbrace{\mathsf{Undef}, \ldots, \mathsf{Undef}}_{l} \Leftarrow \psi', A_l, \ldots, A_1$

def. of cvar substitution (Ψ cannot contain a context variable)

$\Psi \vdash \uparrow^{-\psi',k} \Leftarrow \psi'$ typing rule

$\Psi, A \vdash \uparrow^{-\psi',k+1} \Leftarrow \psi'$ typing rule

$\Psi, A \vdash \uparrow^{-\psi',k+1}, \underbrace{\mathsf{Undef}, \ldots, \mathsf{Undef}}_{l} \Leftarrow \psi', A_l, \ldots, A_1$ typing rule

$[\![\Psi''/\psi]\!](\Psi, A) \vdash [\![\Psi''/\psi]\!] \uparrow^{-\psi,k+1} \Leftarrow [\![\Psi''/\psi]\!]\psi$ def. of cvar substitution

Case $\Psi' \vdash \uparrow^{0,k} \Leftarrow \Psi$

$k \geq 0$ inversion

$\Psi' \vdash \uparrow^{0,k+1}, k+1 \Leftarrow \Psi$ inversion

$[\![\Psi''/\psi]\!]\Psi' \vdash [\![\Psi''/\psi]\!] \uparrow^{0,k+1}, k+1 \Leftarrow [\![\Psi''/\psi]\!]\Psi$ i.h.

$[\![\Psi''/\psi]\!]\Psi' \vdash \uparrow^{0,k+1}, k+1 \Leftarrow [\![\Psi''/\psi]\!]\Psi$ def. of cvar substitution

$[\![\Psi''/\psi]\!]\Psi' \vdash \uparrow^{0,k} \Leftarrow [\![\Psi''/\psi]\!]\Psi$ typing rule

$[\![\Psi''/\psi]\!]\Psi' \vdash [\![\Psi''/\psi]\!] \uparrow^{0,k} \Leftarrow [\![\Psi''/\psi]\!]\Psi$ def. of cvar substitution

Case $\Psi' \vdash \sigma, H \Leftarrow \Psi, A$

$\Psi' \vdash \sigma \Leftarrow \Psi$ inversion

$[\![\Psi''/\psi]\!]\Psi' \vdash [\![\Psi''/\psi]\!]\sigma \Leftarrow [\![\Psi''/\psi]\!]\Psi$ i.h.

$\Psi' \vdash H \Rightarrow (A', \sigma')$ inversion

$[\![\Psi''/\psi]\!]\Psi' \vdash [\![\Psi''/\psi]\!]H \Rightarrow [\![\Psi''/\psi]\!](A', \sigma')$ assumption

$(A', \sigma') \simeq (A, \sigma)$ inversion

$[\![\Psi''/\psi]\!](A', \sigma') \simeq [\![\Psi''/\psi]\!](A, \sigma)$ assumption

$[\![\Psi''/\psi]\!]\Psi' \vdash [\![\Psi''/\psi]\!](\sigma, H) \Leftarrow [\![\Psi''/\psi]\!](\Psi, A)$ typing rule

Case $\Psi' \vdash \sigma, \mathsf{Undef} \Leftarrow \Psi, A$

$\Psi' \vdash \sigma \Leftarrow \Psi$	inversion
$[\![\Psi''/\psi]\!]\Psi' \vdash [\![\Psi''/\psi]\!]\sigma \Leftarrow [\![\Psi''/\psi]\!]\Psi$	i.h.
$[\![\Psi''/\psi]\!]\Psi' \vdash [\![\Psi''/\psi]\!](\sigma, \mathsf{Undef}) \Leftarrow [\![\Psi''/\psi]\!]\Psi$	def. of cvar substitution

Case $\Psi' \vdash \sigma, M \Leftarrow \Psi, A$

$\Psi' \vdash (M, \mathsf{id}) \Leftarrow (A, \sigma)$	inversion
$[\![\Psi''/\psi]\!]\Psi' \vdash [\![\Psi''/\psi]\!](M, \mathsf{id}) \Leftarrow [\![\Psi''/\psi]\!](A, \sigma)$	assumption
$\Psi' \vdash \sigma \Leftarrow \Psi$	inversion
$[\![\Psi''/\psi]\!]\Psi' \vdash [\![\Psi''/\psi]\!]\sigma \Leftarrow [\![\Psi''/\psi]\!]\Psi$	i.h.
$[\![\Psi''/\psi]\!]\Psi' \vdash [\![\Psi''/\psi]\!](\sigma, M) \Leftarrow [\![\Psi''/\psi]\!](\Psi, A)$	typing rule

□

A.5.1 Composition

We prove some properties about the composition operation defined in Chapter 4.

Lemma 4 (Invariant of composition)

Assuming composition $\sigma_1 \circ \sigma_2$ *is defined, we have:*
If $\Psi \vdash \sigma_1 \Leftarrow \Psi_1$ *and* $\Psi_2 \vdash \sigma_2 \Leftarrow \Psi$
then $\Psi_2 \vdash \sigma_1 \circ \sigma_2 \Leftarrow \Psi_1$

Proof: Following the definition for composition, we proceed by induction on

1. the structure of σ_1
2. the structure of σ_2

Case $\Psi \vdash \sigma_1, M \Leftarrow \Psi_1, A$

$\Psi \vdash \sigma_1 \Leftarrow \Psi_1$	inversion
$\Psi_2 \vdash \sigma_1 \circ \sigma_2 \Leftarrow \Psi_1$	i.h.
$\Psi \vdash (M, \mathsf{id}) \Leftarrow (A, \sigma_1)$	inversion

$\Psi_2 \vdash ([\sigma_2]M, \mathsf{id}) \Leftarrow (A, \sigma_1 \circ \sigma_2)$ subs. lemma

$\Psi_2 \vdash (\sigma_1 \circ \sigma_2, [\sigma_2]M) \Leftarrow \Psi_1, A$ typing rule

$\Psi_2 \vdash (\sigma_1, M) \circ \sigma_2 \Leftarrow \Psi_1, A$ $\circ$ rule

Cases for Heads and Undef's are proven in a similar way.

Case $\Psi, A \vdash \uparrow^{c_1,k_1} \Leftarrow \Psi_1$ and $\Psi_2 \vdash (\sigma_2, M) \Leftarrow \Psi, A$

$\Psi \vdash \uparrow^{c_1,k_1-1} \Leftarrow \Psi_1$ inversion

$\Psi_2 \vdash \sigma_2 \Leftarrow \Psi$ inversion

$\Psi_2 \vdash \uparrow^{c_1,k_1-1} \circ \sigma_2 \Leftarrow \Psi_1$ i.h.

$\Psi_2 \vdash \uparrow^{c_1,k_1} \circ (\sigma_2, M) \Leftarrow \Psi_1$ $\circ$ rule

Cases for Heads and Undef's are proven in a similar way.

Case $\Psi_1, A_{k_1}, \ldots, A_1 \vdash \uparrow^{0,k_1} \Leftarrow \Psi_1$

and $\Psi_1, A_{k_1}, \ldots, A_1, B_{k_2}, \ldots, B_1 \vdash \uparrow^{0,k_2} \Leftarrow \Psi_1, A_{k_1}, \ldots, A_1$

$\Psi_1, A_{k_1}, \ldots, A_1, B_{k_2}, \ldots, B_1 \vdash \uparrow^{0,k_1+k_2} \Leftarrow \Psi_1$ typing rules

Case $\psi, A_{k_1}, \ldots, A_1 \vdash \uparrow^{\psi,k_1} \Leftarrow \cdot$

and $\psi, A_{k_1}, \ldots, A_1, B_{k_2}, \ldots, B_1 \vdash \uparrow^{0,k_2} \Leftarrow \psi, A_{k_1}, \ldots, A_1$

$\psi, A_{k_1}, \ldots, A_1, B_{k_2}, \ldots, B_1 \vdash \uparrow^{\psi,k_1+k_2} \Leftarrow \cdot$ typing rules

Case $A_{k_1}, \ldots, A_1 \vdash \uparrow^{-\psi,k_1} \Leftarrow \psi$

and $A_{k_1}, \ldots, A_1, B_{k_2}, \ldots, B_1 \vdash \uparrow^{0,k_2} \Leftarrow A_{k_1}, \ldots, A_1$

$A_{k_1}, \ldots, A_1, B_{k_2}, \ldots, B_1 \vdash \uparrow^{-\psi,k_1+k_2}$ typing rules

Case $\cdot \vdash \uparrow^{-\psi,0} \Leftarrow \psi$ and $\psi, A_{k_2}, \ldots, A_1 \vdash \uparrow^{\psi,k_2} \Leftarrow \cdot$

$\psi, A_{k_2}, \ldots, A_1 \vdash \uparrow^{0,k_2} \Leftarrow \psi$ typing rule

Case $\psi \vdash \uparrow^{\psi,0} \Leftarrow \cdot$ and $\Psi_2 \vdash \uparrow^{-\psi,k_2} \Leftarrow \psi$

$\Psi_2 \vdash \uparrow^{0,k_2} \Leftarrow \cdot$ typing rule

$\Psi_2 \vdash \uparrow^{\psi,0} \circ \uparrow^{-\psi,k_2} \Leftarrow .$ $\circ$ rule

Case $\Psi_1 \vdash \mathsf{id} \Leftarrow \Psi_1$ and $\Psi_2 \vdash \sigma_2 \Leftarrow \Psi_1$

$\Psi_2 \vdash \mathsf{id} \circ \sigma_2 \Leftarrow \Psi_1$ $\circ$ rule

□

Lemma 5 (Associativity of composition)

When these compositions are defined, they are equal:

$\sigma_1 \circ (\sigma_2 \circ \sigma_3) = (\sigma_1 \circ \sigma_2) \circ \sigma_3$

Proof: Induction on the structure of σ_1, then σ_2.

We suppose that σ_1, σ_2 and σ_3 are not the identity (id) substitution, in which case the theorem becomes trivial by the fact that id is also the right identity.

Case $(\sigma_1, M) \circ (\sigma_2 \circ \sigma_3)$

$= (\sigma_1 \circ (\sigma_2 \circ \sigma_3), [\sigma_2 \circ \sigma_3]M)$ def. of $\circ$

$= ((\sigma_1 \circ \sigma_2) \circ \sigma_3, [\sigma_2 \circ \sigma_3]M)$ i.h.

$= ((\sigma_1 \circ \sigma_2) \circ \sigma_3, [\sigma_3][\sigma_2]M)$ prop. of $\circ$

$= (\sigma_1 \circ \sigma_2, [\sigma_2]M) \circ \sigma_3$ def. of $\circ$

$= ((\sigma_1, M) \circ \sigma_2) \circ \sigma_3$ def. of $\circ$

Case $(\sigma_1, H) \circ (\sigma_2 \circ \sigma_3)$

Case $(\sigma_1, \mathsf{Undef}) \circ (\sigma_2 \circ \sigma_3)$

similar to previous case

Case $\sigma_1 = \uparrow^{c_1,k_1}$ and $(\sigma_2 \circ \sigma_3) = (\sigma', M')$

$\sigma_1 = \uparrow^{c,k+1}$ def. of $\circ$ $(\sigma_1 \neq \mathsf{id})$

$\sigma_2 = (\sigma_2', M)$ def. of $\circ$ $(\sigma_2 \neq \mathsf{id})$

$\uparrow^{c,k+1} \circ ((\sigma_2', M) \circ \sigma_3)$

$= \uparrow^{c,k+1} \circ (\sigma_2' \circ \sigma_3, [\sigma_3]M)$ def. of $\circ$

$= \uparrow^{c,k} \circ (\sigma_2' \circ\ \sigma_3)$ def. of $\circ$

$= (\uparrow^{c,k} \circ \sigma_2') \circ\ \sigma_3$ i.h.

$= (\uparrow^{c,k+1} \circ (\sigma_2', M)) \circ\ \sigma_3$ def. of $\circ$

Case $\uparrow^{c,k+1} \circ ((\sigma_2, H) \circ\ \sigma_3)$

Case $\uparrow^{c,k+1} \circ ((\sigma_2, \mathsf{Undef}) \circ\ \sigma_3)$

similar to previous case

Case $\sigma_1 = \uparrow^{c_1,k_1}$ and $(\sigma_2 \circ\ \sigma_3) = \uparrow^{c,k}$

$\sigma_2 = \uparrow^{c_2,k_2}$ def. of $\circ$

let $\sigma_3 = \uparrow^{c_3,k_3}, \ldots)$

Subcase suffix of σ_3 is of length $k_2 + x$

$\sigma_1 \circ\ (\sigma_2 \circ\ \sigma_3)$

$= \uparrow^{c_1,k_1} \circ (\uparrow^{c_2,0} \circ\ \uparrow^{c_3,k_3}, \underbrace{\ldots}_{x})$ def. of $\circ$

$= \uparrow^{c_1,k_1} \circ (\mathsf{id} \circ\ \uparrow^{c_3,k_3}, \ldots)$ $c_2 = 0$ (else $\circ$ is undefined)

$= \uparrow^{c_1,k_1} \circ\ \uparrow^{c_3,k_3}, \ldots$ def. of $\circ$

$= \uparrow^{c_1,k_1+k_2} \circ\ \uparrow^{c_3,k_3}, \underbrace{\ldots}_{k_2+x}$ def. of $\circ$

$= (\uparrow^{c_1,k_1} \circ\ \uparrow^{c_2,k_2}) \circ\ \uparrow^{c_3,k_3}, \ldots$ def. of $\circ$

Subcase suffix of σ_3 is of length $k_2 - x$

$\sigma_1 \circ\ (\sigma_2 \circ\ \sigma_3)$

$= \uparrow^{c_1,k_1} \circ (\uparrow^{c_2,x} \circ\ \uparrow^{c_3,k_3})$ def. of $\circ$

$= \uparrow^{c_1,k_1} \circ (\uparrow^{c_2+c_3,x+k_3})$ def. of $\circ$ [1]

$= \uparrow^{c_1+(c_2+c_3),k_1+(x+k_3)}$ def. of $\circ$

$= \uparrow^{(c_1+c_2)+c_3,(k_1+x)+k_3}$ assumption

$= \uparrow^{c_1+c_2,k_1+x} \circ\ \uparrow^{c_3,k_3}$ def. of $\circ$

[1] $c_1 + c_2$ defined as $\begin{cases} c + 0 = c \\ 0 + c = c \\ c + (-c) = 0 \\ \text{undefined otherwise} \end{cases}$

$= \uparrow^{c_1+c_2,k_1+k_2} \circ \uparrow^{c_3,k_3}, \ldots$ def. of $\circ$

$= (\uparrow^{c_1,k_1} \circ \uparrow^{c_2,k_2}) \circ \uparrow^{c_3,k_3}, \ldots$ def. of $\circ$

$\square$

A.5.2 Inversion

We prove some properties about the inversion operation defined in Chapter 4.

Lemma 7 (Inversion produces well-formed substitution)

If $\Psi \vdash \sigma \Leftarrow \Phi$ *then* $\Phi \vdash \sigma^{-1} \Leftarrow \Psi$

Proof: By induction on the size of Φ then on k.

Case $\cdot \vdash \uparrow^{0,0} \Leftarrow \cdot$

$\cdot \vdash \uparrow^{-0,0} \Leftarrow \cdot$ typing rule

Case $\psi \vdash \uparrow^{0,0} \Leftarrow \psi$

$\psi \vdash \uparrow^{-0,0} \Leftarrow \psi$ typing rule

Case $\psi \vdash \uparrow^{\psi,0} \Leftarrow \cdot$

$\cdot \vdash \uparrow^{-\psi,0} \Leftarrow \psi$ typing rule

Case $\cdot \vdash \uparrow^{-\psi,0} \Leftarrow \psi$

$\psi \vdash \uparrow^{\psi,0} \Leftarrow \cdot$ typing rule

Case $\Psi, A \vdash \uparrow^{c,k+1} \Leftarrow \cdot$

$\Psi \vdash \uparrow^{c,k} \Leftarrow \cdot$ inversion

$\cdot \vdash \uparrow^{-c,0}, \underbrace{\mathsf{Undef}, \ldots, \mathsf{Undef}}_{k} \Leftarrow \Psi$ i.h.

$\cdot \vdash \uparrow^{-c,0}, \underbrace{\mathsf{Undef}, \ldots, \mathsf{Undef}}_{k+1} \Leftarrow \Psi, A$ typing rule

$\cdot \vdash (\uparrow^{c,k+1})^{-1} \Leftarrow \Psi, A$ inversion rules

Case $\Psi, A \vdash \uparrow^{c,k+1} \Leftarrow \psi$

$\Psi \vdash \uparrow^{c,k} \Leftarrow \psi$ inversion

$\psi \vdash \uparrow^{-c,0}, \underbrace{\mathsf{Undef}, \ldots, \mathsf{Undef}}_{k} \Leftarrow \Psi$ i.h.

$\psi \vdash \uparrow^{-c,0}, \underbrace{\mathsf{Undef}, \ldots, \mathsf{Undef}}_{k+1} \Leftarrow \Psi, A$ typing rule

$\psi \vdash (\uparrow^{c,k+1})^{-1} \Leftarrow \Psi, A$ inversion rules

Case $\Psi' \vdash \uparrow^{0,k} \Leftarrow \Psi, A$

We first observe that $\Psi' = \Psi, A, \Psi''$ and $|\Psi''| = k$

$\Psi, A \vdash \uparrow^{0,0} \Leftarrow \Psi, A$ lemma 10

$\Psi, A \vdash \uparrow^{0,0}, \underbrace{\mathsf{Undef}, \ldots, \mathsf{Undef}}_{k} \Leftarrow \Psi'$ typing rule

$\Psi, A \vdash (\uparrow^{0,k})^{-1} \Leftarrow \Psi'$ inversion rule

Case $\Psi' \vdash \sigma, H \Leftarrow \Psi, A$

$\Psi' \vdash \sigma \Leftarrow \Psi$ inversion

let $\sigma = \uparrow^{c,k}, H_{k'}, \ldots, H_1$

$\sigma^{-1} = \uparrow^{-c,k'}, b_k, \ldots, b_1$ inversion rules

$\Psi \vdash \uparrow^{-c,k'}, b_k, \ldots, b_1 \Leftarrow \Psi'$ i.h.

$\Psi \vdash \uparrow^{-c,k''} \Leftarrow \cdot$ typing rules

$\Psi, A \vdash \uparrow^{-c,k''+1} \Leftarrow \cdot$ typing rule

$\Psi, A \vdash \uparrow^{-c,k'+1}, b_k, \ldots, b_1 \Leftarrow \Psi'$ typing rule

$\Psi, A \vdash (\sigma, H)^{-1} \Leftarrow \Psi'$ inversion rules

Inverse is not defined if σ is of the form (σ, Undef) or (σ, M). □

Lemma 8 (Inversion properties)

If σ is a pattern substitution, we have:

1. $\sigma \circ \sigma^{-1} = \mathsf{id}$

2. $(\sigma^{-1})^{-1} = \sigma$

Proof: [1]

Let $\sigma = \uparrow^{c,k}, a_{k'}, \ldots, a_1 \qquad \sigma^{-1} = \uparrow^{-c,k'}, b_k, \ldots, b_1$ where $k, k' \geq 0$

$$\begin{aligned}
\sigma \circ \sigma^{-1} &= (\uparrow^{c,k} \circ (\uparrow^{-c,k'}, b_k, \ldots, b_1)), [\sigma^{-1}]a_{k'}, \ldots, [\sigma^{-1}]a_1 \\
&= (\uparrow^{c,0} \circ \uparrow^{-c,k'}), [\sigma^{-1}]a_{k'}, \ldots, [\sigma^{-1}]a_1 \\
&= \uparrow^{0,k'}, [\sigma^{-1}]a_{k'}, \ldots, [\sigma^{-1}]a_1 \\
&= \uparrow^{0,k'}, k', \ldots, 1 \\
&= \uparrow^{0,0}
\end{aligned}$$

The requirement that σ is a pattern substitution is critical here. Indeed, the equation $[\sigma^{-1}]a_i = i$ is only true if $a_i \neq \mathsf{Undef}$ and $[\sigma^{-1}]a_i$ is unique. The later is always defined because $a_i \leq k$ (else σ is not a pattern substitution) and therefore variable a_i is always part of σ^{-1}'s "tail" (the part of the substitution that is not a shift) and maps to i. Also note that the $k', \ldots, 1$ part will always be ordered because we understand these substitutions in the context where we work with DeBruijn indexing. Therefore, a_i do not replace just some variable in σ's domain, but precisely the variable of index i. Other steps hold by composition rules and the last one by typing rule. □

Proof: [2]

Trivial. The reader can easily verify that applying the inversion operation twice will always return the original substitution. □

Appendix B

Reconstruction

B.1 Contextual substitution

$$
\begin{array}{ll}
[\![R/u]\!]\mathsf{type} & = \mathsf{type} \\
[\![R/u]\!]\Pi A.K & = \Pi([\![R/u]\!]A).([\![R/u]\!]K) \\
 & \\
[\![R/u]\!]\mathsf{a} \cdot S & = \mathsf{a} \cdot [\![R/u]\!]S \\
[\![R/u]\!]\Pi A.B & = \Pi([\![R/u]\!]A).([\![R/u]\!]B) \\
 & \\
[\![R/u]\!]\lambda.M & = \lambda.[\![R/u]\!]M \\
[\![R/u]\!]H \cdot S & = H \cdot [\![R/u]\!]S \\
[\![R/u]\!]u[\sigma] & = \mathsf{clo}(R, [\![R/u]\!]\sigma) \\
[\![R/u]\!]v[\sigma] & = v[[\![R/u]\!]\sigma] \\
 & \\
[\![R/u]\!]\mathsf{nil} & = \mathsf{nil} \\
[\![R/u]\!]M\ S & = [\![R/u]\!]M\ [\![R/u]\!]S \\
 & \\
[\![R/u]\!]\uparrow^{c,k} & = \uparrow^{c,k} \\
[\![R/u]\!](\sigma, M) & = ([\![R/u]\!]\sigma), ([\![R/u]\!]M) \\
[\![R/u]\!](\sigma, x) & = ([\![R/u]\!]\sigma), x \\
 & \\
[\![R/u]\!]\cdot & = \cdot \\
[\![R/u]\!](\Psi, A) & = ([\![R/u]\!]\Psi), ([\![R/u]\!]A) \\
[\![R/u]\!](\Phi, X{:}A) & = ([\![R/u]\!]\Phi), X{:}([\![R/u]\!]A)
\end{array}
$$

We list here the rules for contextual substitution. Note that since reconstruction

works with meta-variables of atomic type, this operation will never introduce redexes. As such, termination is guaranteed by the fact that this operation is a simple syntax directed recursion and we will not need to index it with the type of the term we are substituting, as we did in Chapter 3.

B.2 Additional reconstruction judgements

$\Upsilon_1;\Phi_1;\Psi$	$\vdash$	k	$\mathsf{kind}\ /_\rho\ (\Upsilon_2;\Phi_2;K)$	k reconstructs to K
$\Upsilon_1;\Phi_1;\Psi$	$\vdash$	a	$\Leftarrow \mathsf{type}\ /_\rho\ (\Upsilon_2;\Phi_2;A)$	a reconstructs to A
$\Upsilon_1;\Phi_1;\Psi$	$\vdash$	$s : (K,\sigma)$	$\Leftarrow \mathsf{type}\ /_\rho\ (\Upsilon_2;\Phi_2;S)$	s reconstructs to S
$\Upsilon_1;\Phi_1;\Psi$	$\vdash^i$	$s : (K,\sigma)$	$\Leftarrow \mathsf{type}\ /_\rho\ (\Upsilon_2;\Phi_2;S)$	introduce i arguments and reconstruct s to S
$\Upsilon_1;\Phi_1;\Psi$	$\vdash$	s	$\Leftarrow \mathsf{type}\ /\ (K;S)$	s reconstructs to S and synthesize kind K

B.3 Additional reconstruction rules

Kinds

$$\frac{}{\Upsilon_1;\Phi_1;\Psi \vdash \mathsf{type}\ \mathsf{kind}\ /_{\mathsf{id}(\Upsilon_1)}\ (\Upsilon_1;\Phi_1;\mathsf{type})}$$

$$\frac{\Upsilon_1;\Phi_1;\Psi \vdash a \Leftarrow \mathsf{type}\ /_{\rho_1}\ (\Upsilon_2;\Phi_2;A) \quad \Upsilon_2;\Phi_2;[\![\rho_1]\!]\Psi, A \vdash k\ \mathsf{kind}\ /_{\rho_2}\ (\Upsilon_3;\Phi_3;K)}{\Upsilon_1;\Phi_1;\Psi \vdash \Pi a.k\ \mathsf{kind}\ /_{\rho_1\circ\,\rho_2}\ (\Upsilon_3;\Phi_3;\Pi([\![\rho_2]\!]A).K)}$$

Types

$$\frac{\Sigma(\mathsf{a}) = (K,i) \quad \Upsilon_1;\Phi_1;\Psi \vdash^i s : (K,\mathsf{id}) \Leftarrow \mathsf{type}\ /_\rho\ (\Upsilon_2;\Phi_2;S)}{\Upsilon_1;\Phi_1;\Psi \vdash \mathsf{a}\cdot s \Leftarrow \mathsf{type}\ /_\rho\ (\Upsilon_2;\Phi_2;\mathsf{a}\cdot S)}$$

$$\frac{\Upsilon_1;\Phi_1;\Psi \vdash a \Leftarrow \mathsf{type}\ /_{\rho_1}\ (\Upsilon_2;\Phi_2;A) \quad \Upsilon_2;\Phi_2;[\![\rho_1]\!]\Psi, A \vdash b \Leftarrow \mathsf{type}\ /_{\rho_2}\ (\Upsilon_3;\Phi_3;B)}{\Upsilon_1;\Phi_1;\Psi \vdash \Pi a.b \Leftarrow \mathsf{type}\ /_{\rho_1\circ\,\rho_2}\ (\Upsilon_3;\Phi_3;\Pi([\![\rho_2]\!]A).B)}$$

Type spines

$$\frac{}{\Upsilon_1; \Phi_1; \Psi \vdash \mathsf{nil} : (\mathsf{type}, \sigma) \Leftarrow \mathsf{type} \,/_{\mathsf{id}(\Upsilon_1)}\, (\Upsilon_1; \Phi_1; \mathsf{nil})}$$

$$\frac{\begin{array}{l}\Upsilon_1; \Phi_1; \Psi \vdash m \Leftarrow (A, \sigma) \,/_{\rho_1}\, (\Upsilon_2; \Phi_2; M) \\ \Upsilon_2; \Phi_2; [\![\rho_1]\!]\Psi \vdash s : ([\![\rho_1]\!]K, ([\![\rho_1]\!]\sigma, M)) \Leftarrow \mathsf{type} \,/_{\rho_2}\, (\Upsilon_3; \Phi_3; S)\end{array}}{\Upsilon_1; \Phi_1; \Psi \vdash m\ s : (\Pi A.K, \sigma) \Leftarrow \mathsf{type} \,/_{\rho_1 \circ\, \rho_2}\, (\Upsilon_3; \Phi_3; [\![\rho_2]\!]M\ S)}$$

Type spines (with missing arguments)

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

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

Type spines (synthesizing)

$$\frac{}{\Upsilon_1; \Phi_1; \Psi \vdash \mathsf{nil} \Leftarrow \mathsf{type} \,/\, (\mathsf{type}; \mathsf{nil})}$$

$$\frac{\begin{array}{l}\mathsf{contract}_0\ m \,/\, x \quad \Psi(x) = (A, \uparrow^{0,x}) \quad \Psi \vdash^0 \mathsf{expand}\ x\ :\ (A, \uparrow^{0,x}) \,/\, M \\ \Upsilon_1; \Phi_1; \Psi \vdash s \Leftarrow \mathsf{type} \,/\, (K; S) \quad \sigma' = \uparrow^{0,x+1}, 1, x, \ldots, 1\end{array}}{\Upsilon_1; \Phi_1; \Psi \vdash m\ s \Leftarrow \mathsf{type} \,/\, (\Pi\mathsf{clo}(A, \uparrow^{0,x}).\mathsf{clo}(K, \uparrow^{0,1} \circ \sigma'); M\ S)}$$

B.4 Proofs

B.4.1 Invariant of reconstruction

Judgements for contexts

$$
\begin{array}{rlll}
 & \vdash_\Phi \ \Upsilon & \mathsf{mctx} & \Upsilon \text{ is a well-formed meta context} \\
\Upsilon & \vdash \ \Phi & \mathsf{fctx} & \Phi \text{ is a well-formed free variable context} \\
\Upsilon; \Phi & \vdash \ \Psi & \mathsf{ctx} & \Psi \text{ is a well-formed bound variable context} \\
\Upsilon' & \vdash_\Phi \ \rho & \Leftarrow \Upsilon & \rho \text{ is a contextual substitution with domain } \Upsilon \text{ and range } \Upsilon'
\end{array}
$$

Rules for contexts and contextual substitutions

Meta contexts

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

Free variable contexts

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

Bound variable contexts

$$\frac{}{\Upsilon_1; \Phi_1 \vdash \cdot \ \mathsf{ctx}} \qquad \frac{\Upsilon; \Phi \vdash \Psi \ \mathsf{ctx} \quad \Upsilon; \Phi; \Psi \vdash (A, \mathsf{id}) \Leftarrow \mathsf{type}}{\Upsilon; \Phi \vdash \Psi, A \ \mathsf{ctx}}$$

Contextual substitutions

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

Invariant

With the preceding definitions in mind, the invariant of reconstruction can be summarized as follow: the algorithm assumes well-kinded types and well-formed contexts and produces well-formed contexts and contextual substitution.

$$
\begin{array}{llll}
\cdot & \vdash_{\Phi_1} & \Upsilon_1 & \mathsf{mctx} \\
\Upsilon_1 & \vdash & \Phi_1 & \mathsf{fctx} \\
\Upsilon_1; \Phi_1 & \vdash & \Psi & \mathsf{ctx} \\
\Upsilon_1; \Phi_1; \Psi & \vdash & (A, \sigma) & \Leftarrow \mathsf{type} \\
\Upsilon_1; \Phi_1; \Psi & \vdash & (P, \sigma) & \Leftarrow \mathsf{type}
\end{array}
$$

$$
\begin{array}{llll}
 & \vdash_{\Phi_2} & \Upsilon_2 & \mathsf{mctx} \\
\Upsilon_2 & \vdash & \Phi_2 & \mathsf{fctx} \\
\Upsilon_2 & \vdash_{\Phi_2} & \rho & \Leftarrow \Upsilon_1
\end{array}
$$

Proof: By induction on the reconstruction derivation.

In most cases, the invariant holds simply by the induction hypotheses and the fact that we manipulate the objects correctly.
The free variable case holds because a similar invariant holds for pruning.
The nil spine case holds because a similar invariant holds for unification.
The placeholder (_) case holds based on the definition of $\mathsf{id}(\Upsilon)$.
The interesting cases left are given next (we prove only the second part of the invariant).

Case $\Upsilon_1; \Phi_1; \Psi \vdash^i s : (\Pi A.B, \sigma_1) \Leftarrow (P, \sigma_2) \; /_{\rho\backslash(R/u)} \; (\Upsilon_2; \Phi_2; [\![\rho]\!]M \; S)$

$\mathsf{lower}(\Upsilon_1; \Phi; \Psi \vdash (A, \sigma_1)) = (M, u{::}Q[\Psi'])$ inversion

$\Upsilon_1, u{::}Q[\Psi']; \Phi_1; \Psi \vdash^{i-1} s : (B, (\sigma_1; M)) \Leftarrow (P, \sigma_2) \; /_\rho \; (\Upsilon_2; \Phi_2; S)$ inversion

$\vdash_{\Phi_2} \Upsilon_2 \; \mathsf{mctx}$ and $\Upsilon_2 \vdash \Phi_2 \; \mathsf{fctx}$ i.h.

$\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1, u{::}Q[\Psi']$ i.h.

for all$(R/v) \in \rho \quad (\Upsilon_1, u{::}Q[\Psi'])(v) = P[\Psi] \quad \Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash (R, \mathsf{id}) \Leftarrow [\![\rho]\!](P, \mathsf{id})$ inversion

let $\rho' = \rho\backslash(R/u)$

for all$(R/v) \in \rho' \quad \Upsilon_1(v) = P[\Psi] \quad \Upsilon_2; \Phi_2; [\![\rho']\!]\Psi \vdash (R, \mathsf{id}) \Leftarrow [\![\rho']\!](P, \mathsf{id})$

Ψ and Υ_1 do not refer to u

$\Upsilon_2 \vdash_{\Phi_2} \rho' \Leftarrow \Upsilon_1$ typing rule

Case $\Upsilon_1; \Phi_1; \Psi \vdash m\ s : (\Pi A.B, \sigma_1) \Leftarrow (P, \sigma_2)\ /_{\rho_1 \circ\ \rho_2}\ (\Upsilon_3; \Phi_3; [\![\rho_2]\!]M\ S)$

$\Upsilon_1; \Phi_1; \Psi \vdash m \Leftarrow (A, \sigma_1)\ /_{\rho_1}\ (\Upsilon_2; \Phi_2; M)$ inversion

$\vdash_{\Phi_2} \Upsilon_2\ \mathsf{mctx}$ and $\Upsilon_2 \vdash \Phi_2\ \mathsf{fctx}$ and $\Upsilon_2 \vdash_{\Phi_2} \rho_1 \Leftarrow \Upsilon_1$ i.h.

$\Upsilon_2; \Phi_2; [\![\rho_1]\!]\Psi \vdash s : ([\![\rho_1]\!]B, ([\![\rho_1]\!]\sigma_1; M)) \Leftarrow [\![\rho_1]\!](P, \sigma_2)\ /_{\rho_2}\ (\Upsilon_3; \Phi_3; S)$ inversion

$\vdash_{\Phi_3} \Upsilon_3\ \mathsf{mctx}$ and $\Upsilon_3 \vdash \Phi_3\ \mathsf{fctx}$ and $\Upsilon_3 \vdash_{\Phi_3} \rho_2 \Leftarrow \Upsilon_2$ i.h.

for all$(R/u) \in \rho_1 \quad \Upsilon_1(u) = P[\Psi] \quad \Upsilon_2; \Phi_2; [\![\rho_1]\!]\Psi \vdash (R, \mathsf{id}) \Leftarrow [\![\rho_1]\!](P, \mathsf{id})$

inversion

for all$([\![\rho_2]\!]R/u) \in [\![\rho_2]\!]\rho_1 \quad \Upsilon_1(u) = P[\Psi] \quad \Upsilon_3; [\![\rho_2]\!]\Phi_2; [\![\rho_2]\!][\![\rho_1]\!]\Psi \vdash ([\![\rho_2]\!]R, \mathsf{id}) \Leftarrow [\![\rho_2]\!][\![\rho_1]\!](P, \mathsf{id})$

stability of type checking under contextual substitution

for all$([\![\rho_2]\!]R/u) \in [\![\rho_2]\!]\rho_1 \quad \Upsilon_1(u) = P[\Psi] \quad \Upsilon_3; \Phi_3; [\![\rho_2]\!][\![\rho_1]\!]\Psi \vdash ([\![\rho_2]\!]R, \mathsf{id}) \Leftarrow [\![\rho_2]\!][\![\rho_1]\!](P, \mathsf{id})$ $[\![\rho_2]\!]\Phi_2 \subseteq \Phi_3$

$\Upsilon_3 \vdash_{\Phi_3} [\![\rho_2]\!]\rho_1 \Leftarrow \Upsilon_1$ typing rule

□

B.4.2 Lemmas

Lemma 18 $\mathsf{dot1}\ \mathsf{id} = \mathsf{id}$

Proof: By using the definition for id, $\mathsf{dot1}$ and using the `unroll` rule.

$$\begin{aligned}
\mathsf{dot1}\ \mathsf{id} &= (\mathsf{id} \circ \uparrow^{0,1}), 1 && \text{def. of } \mathsf{dot1} \\
&= \uparrow^{0,1}, 1 && \text{def. of } \circ \\
&= \uparrow^{0,0} && \text{typing rule} \\
&= \mathsf{id} && \text{def. of } \mathsf{id}
\end{aligned}$$

□

Lemma 15 (Lowering)

If $\mathsf{lower}(\Upsilon; \Phi; \Psi \vdash (A, \sigma)) = (M; u{::}P[\Psi'])$

then $\Upsilon, u{::}P[\Psi']; \Phi; \Psi \vdash (M, \mathsf{id}) \Leftarrow (A, \sigma)$.

Proof: By structural induction on A.

Case $\mathsf{lower}(\Upsilon; \Phi; \Psi \vdash (P, \sigma)) = (u[\mathsf{id}]; u{::}\mathsf{clo}(P, \sigma)[\Psi])$

$(\Upsilon, u{::}\mathsf{clo}(P, \sigma)[\Psi])(u) = \mathsf{clo}(P, \sigma)[\Psi]$	def.
$\Upsilon, u{::}\mathsf{clo}(P, \sigma)[\Psi]; \Phi; \Psi \vdash \mathsf{id} \Leftarrow \Psi$	typing rule
$\sigma \circ \mathsf{id} = \sigma$	prop. of id
$\mathsf{whnf}(\mathsf{clo}(P, \sigma), \mathsf{id}) = \mathsf{whnf}(P, \sigma \circ \mathsf{id})$	def. of whnf
$(\mathsf{clo}(P, \sigma), \mathsf{id}) \simeq (P, \sigma)$	def. of $\simeq$
$\Upsilon, u{::}\mathsf{clo}(P, \sigma)[\Psi]; \Phi; \Psi \vdash u[\mathsf{id}] \Leftarrow (P, \sigma)$	typing rule

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

$\mathsf{lower}(\Upsilon; \Phi; \Psi, \mathsf{clo}(A, \sigma) \vdash (B, \mathsf{dot1}\ \sigma)) = (M; u{::}P[\Psi'])$	inversion
$\Upsilon, u{::}P[\Psi']; \Phi; \Psi, \mathsf{clo}(A, \sigma) \vdash (M, \mathsf{id}) \Leftarrow (B, \mathsf{dot1}\ \sigma)$	i.h.
$\Upsilon, u{::}P[\Psi']; \Phi; \Psi, \mathsf{clo}(A, \sigma) \vdash (M, \mathsf{dot1}\ \mathsf{id}) \Leftarrow (B, \mathsf{dot1}\ \sigma)$	lemma 18
$\Upsilon, u{::}P[\Psi']; \Phi; \Psi \vdash (M, \mathsf{id}) \Leftarrow (\Pi A.B, \sigma)$	typing rule

□

Lemma 12 (η-expansion produces well-typed terms)

1. *If* $\Psi \vdash^i \mathsf{expand}\ x\ :\ (A, \sigma)\ /\ M$ *then* $\Psi \vdash (M, \mathsf{id}) \Leftarrow (A, \sigma)$.

2. *If* $\Psi \vdash^i \mathsf{expand}\ (A, \sigma_1) \Rightarrow (P, \sigma_2)\ /\ S$ *then* $\Psi \vdash (S, \mathsf{id})\ :\ (A, \sigma_1) \Rightarrow (P, \sigma_2)$

Proof: By induction on the expansion derivation.

Case $\Psi \vdash^i \mathsf{expand}\ y\ :\ (\Pi A.B, \sigma)\ /\ \lambda.M$

$\Psi, \mathsf{clo}(A, \sigma) \vdash^{i+1} \mathsf{expand}\ y\ :\ (B, \mathsf{dot1}\ \sigma)\ /\ M$	inversion
$\Psi, \mathsf{clo}(A, \sigma) \vdash (M, \mathsf{id}) \Leftarrow (B, \mathsf{dot1}\ \sigma)$	i.h.1
$\Psi, \mathsf{clo}(A, \sigma) \vdash (M, \mathsf{dot1}\ \mathsf{id}) \Leftarrow (B, \mathsf{dot1}\ \sigma)$	lemma 18
$\Psi \vdash (\lambda.M, \mathsf{id}) \Leftarrow (\Pi A.B, \sigma)$	typing rule

Case $\Psi \vdash^i \mathsf{expand}\ y\ :\ (P,\sigma)\ /\ y \cdot S$

$\Psi \vdash^i \mathsf{expand}\ S \Rightarrow (A, \uparrow^{0,y})\ /\ (P,\sigma)$	inversion
$\Psi \vdash (S, \mathsf{id}) : (A, \uparrow^{0,y}) \Rightarrow (P,\sigma)$	i.h.2
$\Psi \vdash y \Rightarrow (A, \uparrow^{0,y})$	assumption
$(P,\sigma) \simeq (P,\sigma)$	prop. of $\simeq$
$\Psi \vdash (y \cdot S, \mathsf{id}) \Leftarrow (P,\sigma)$	typing rule

Case $\Psi \vdash^i \mathsf{expand}\ M\ S \Rightarrow (\Pi A.B, \sigma_1)\ /\ (P,\sigma_2)$

$\Psi \vdash^0 \mathsf{expand}\ i\ :\ (A,\sigma_1)\ /\ M$	inversion
$\Psi \vdash (M, \mathsf{id}) \Leftarrow (A,\sigma_1)$	i.h.1
$\Psi \vdash^{i-1} \mathsf{expand}\ (B, (\sigma_1; \mathsf{clo}(M, \mathsf{id}))) \Rightarrow (P,\sigma_2)\ /\ S$	inversion
$\Psi \vdash (S, \mathsf{id}) : (B, (\sigma_1; \mathsf{clo}(M, \mathsf{id}))) \Rightarrow (P,\sigma_2)$	i.h.2
$\Psi \vdash (M\ S, \mathsf{id}) : (\Pi A.B, \sigma_1) \Rightarrow (P,\sigma_2)$	typing rule

Case $\Psi \vdash^0 \mathsf{expand}\ \mathsf{nil} \Rightarrow (P,\sigma)\ /\ (P,\sigma)$

$\Psi \vdash (\mathsf{nil}, \mathsf{id}) : (P,\sigma) \Rightarrow (P,\sigma)$	typing rule

□

Lemma 13 (η-expansion stable under substitution)

If $\Psi \vdash^i \mathsf{expand}\ x\ :\ (B,\sigma)\ /\ N$ *with* $\Psi(x) = (B, \uparrow^{0,x})$ *and*

1. $\Upsilon; \Phi; \Psi \vdash M \Leftarrow A$
2. $\Upsilon; \Phi; \Psi \vdash S : A \Rightarrow P$

then

1. $\Upsilon; \Phi; \Psi \vdash M' \Leftarrow A$
2. $\Upsilon; \Phi; \Psi \vdash M \Leftarrow A'$
3. $\Upsilon; \Phi; \Psi \vdash S : A' \Rightarrow P$
4. $\Upsilon; \Phi; \Psi \vdash S' : A \Rightarrow P$

Normal terms

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

$$\frac{\Sigma(\mathsf{c}) = A \quad \Phi;\Psi \vdash S : A \Rightarrow P}{\Upsilon;\Phi;\Psi \vdash \mathsf{c} \cdot S \Leftarrow P} \quad \frac{\Psi(x) = (A, \uparrow^{0,x}) \quad \Upsilon;\Phi;\Psi \vdash S : [\uparrow^{0,x}]^a_{\Psi_1} A \Rightarrow P}{\Upsilon;\Phi;\Psi \vdash x \cdot S \Leftarrow P}$$

$$\frac{\Phi(X) = A \quad \Upsilon;\Phi;\Psi \vdash S : A \Rightarrow P}{\Upsilon;\Phi;\Psi \vdash X \cdot S \Leftarrow P}$$

Spines

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

Figure B.1: Typing rules for LF objects

Where M', A' and S' are the original terms where one occurrence of x has been replaced by N.

Proof: By induction on the typing derivation.

Most cases hold by induction hypothesis.
In the bound variable case, we have two choices: either this is not the one occurrence that is replaced, in which case the result holds by assumption. If the current occurrence is the one that has been replaced, we note that, because bound variables always appear with a complete spine and hereditary substitution keeps substituting when it encounters a redex, we will have

$$[N/x]x \cdot S = x \cdot S$$

in which case the results also hold by assumption. □

B.4.3 Soundness of reconstruction

Before stating the soundness theorem, we will need a lemma on the invariant of type checking with explicit substitutions. The lemma is stated for normal

terms and spines, but it scales to types and kinds as well.

Note that the theorem's conclusion is stated using eager substitution rules (see figure B.1), so the contexts in these rules are not exactly the same as in the premiss because the later could contain closures. The former are therefore normalized version of the output contexts and we use them as such in the proof, even if we do not introduce the additional annotation in the statement.

Lemma 19 (Invariant of type checking)

1. *If* $\Upsilon; \Phi; \Psi \vdash (M, \sigma_1) \Leftarrow (A, \sigma_2)$
   then $\Upsilon; \Phi; \Psi \vdash [\sigma_1]M \Leftarrow [\sigma_2]A$

2. *If* $\Upsilon; \Phi; \Psi \vdash (S, \sigma_1) : (A, \sigma_2) \Rightarrow (P, \sigma)$
   then $\Upsilon; \Phi; \Psi \vdash [\sigma_1]S : [\sigma_2]A \Rightarrow [\sigma]P$

Proof: By induction on the typing derivation.

Case $\Upsilon; \Phi; \Psi \vdash (\lambda.M, \sigma_1) \Leftarrow (\Pi A.B, \sigma_2)$

$\Upsilon; \Phi; \Psi, \mathsf{clo}(A, \sigma_2) \vdash (M, \mathsf{dot1}\ \sigma_1) \Leftarrow (B, \mathsf{dot1}\ \sigma_2)$	inversion
$\Upsilon; \Phi; \Psi, [\sigma_2]A \vdash [\mathsf{dot1}\ \sigma_1]M \Leftarrow [\mathsf{dot1}\ \sigma_2]B$	i.h.1
$\Upsilon; \Phi; \Psi \vdash \lambda.[\mathsf{dot1}\ \sigma_1]M \Leftarrow \Pi[\sigma_2]A.[\mathsf{dot1}\ \sigma_2]B$	typing rule
$\Upsilon; \Phi; \Psi \vdash [\sigma_1]\lambda.M \Leftarrow [\sigma_2](\Pi A.B)$	def. of subs.

Case $\Upsilon; \Phi; \Psi \vdash (H \cdot S, \sigma_1) \Leftarrow (P, \sigma_2)$

$\Upsilon; \Phi; \Psi \vdash H \Rightarrow (A, \sigma)$	inversion
$\Upsilon; \Phi; \Psi \vdash H \Rightarrow [\sigma]A$	normalized contexts/signature
$\Upsilon; \Phi; \Psi \vdash [\sigma_1]H \Rightarrow [\sigma]A$	$\sigma_1 = \mathsf{id}$ (prop. of whnf)
$\Upsilon; \Phi; \Psi \vdash (S, \sigma_1) : \mathsf{whnf}(A, \sigma) \Rightarrow (P', \sigma_1')$	inversion
$\Upsilon; \Phi; \Psi \vdash [\sigma_1]S : [\sigma]A \Rightarrow [\sigma_1']P'$	i.h.2
$(P', \sigma_1') \simeq (P, \sigma_2)$	inversion
$[\sigma_1']P' = [\sigma_2]P$	lemma 17

$\Upsilon; \Phi; \Psi \vdash [\sigma_1](H \cdot S) \Leftarrow [\sigma_2]P$ typing rule

Case $\Upsilon; \Phi; \Psi \vdash (\mathsf{nil}, \sigma_1) : (P, \sigma_2) \Rightarrow (P, \sigma_2)$

$\Upsilon; \Phi; \Psi \vdash \mathsf{nil} : [\sigma_2]P \Rightarrow [\sigma_2]P$ typing rule

$\Upsilon; \Phi; \Psi \vdash [\sigma_1]\mathsf{nil} : [\sigma_2]P \Rightarrow [\sigma_2]P$ def. of subs.

Case $\Upsilon; \Phi; \Psi \vdash (M\ S, \sigma_1) : (\Pi A.B, \sigma_2) \Rightarrow (P, \sigma)$

$\Upsilon; \Phi; \Psi \vdash (M, \sigma_1) \Leftarrow (A, \sigma_2)$ inversion

$\Upsilon; \Phi; \Psi \vdash [\sigma_1]M \Leftarrow [\sigma_2]A$ i.h.1

$\Upsilon; \Phi; \Psi \vdash (S, \sigma_1) : \mathsf{whnf}(B, (\sigma_2; \mathsf{clo}(M, \sigma_1))) \Rightarrow (P, \sigma)$ inversion

$\Upsilon; \Phi; \Psi \vdash [\sigma_1]S : [\sigma_2, \mathsf{clo}(M, \sigma_1)]B \Rightarrow [\sigma]P$ i.h.2

$\Upsilon; \Phi; \Psi \vdash [\sigma_1]S : [\sigma_2, [\sigma_1]M]B \Rightarrow [\sigma]P$ prop. of hereditary substitution

$\Upsilon; \Phi; \Psi \vdash [\sigma_1]S : [\mathsf{id}, [\sigma_1]M][\mathsf{dot1}\ \sigma_2]B \Rightarrow [\sigma]P$ def. of $\mathsf{dot1}$ and $\circ$

$\Upsilon; \Phi; \Psi \vdash [\sigma_1](M\ S) : \Pi[\sigma_2]A.[\mathsf{dot1}\ \sigma_2]B \Rightarrow [\sigma]P$ typing rule

$\Upsilon; \Phi; \Psi \vdash [\sigma_1](M\ S) : [\sigma_2](\Pi A.B) \Rightarrow [\sigma]P$ def. of subs.

□

Theorem 16 (Soundness of reconstruction)

1. *If* $\Upsilon_1; \Phi_1; \Psi \vdash m \Leftarrow (A, \sigma) \,/_\rho\, (\Upsilon_2; \Phi_2; M)$
   then $\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash M \Leftarrow [\![\rho]\!][\sigma]A$.
2. *If* $\Upsilon_1; \Phi_1; \Psi \vdash^i s : (A, \sigma_1) \Leftarrow (P, \sigma_2) \,/_\rho\, (\Upsilon_2; \Phi_2; S)$
   then $\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash S : [\![\rho]\!][\sigma_1]A \Rightarrow [\![\rho]\!][\sigma_2]P$.
3. *If* $\Upsilon_1; \Phi_1; \Psi \vdash s : (A, \sigma_1) \Leftarrow (P, \sigma_2) \,/_\rho\, (\Upsilon_2; \Phi_2; S)$
   then $\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash S : [\![\rho]\!][\sigma_1]A \Rightarrow [\![\rho]\!][\sigma_2]P$.
4. *If* $\Upsilon_1; \Phi_1; \Psi \vdash s \Leftarrow (P, \sigma_2) \,/\, ((A, \sigma_1); S)$
   and s *is a pattern spine*
   then $\Upsilon_1; \Phi_1; \Psi \vdash S : [\sigma_1]A \Rightarrow [\sigma_2]P$
   and $\Upsilon_1; \Phi_1; \Psi \vdash [\sigma_1]A \Leftarrow \mathsf{type}$.

Moreover, in the first 3 cases, we have:

$[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ and $\Upsilon_2 \vdash \Phi_2$ fctx and $\vdash_{\Phi_2} \Upsilon_2$ mctx

Proof: By structural induction on the first derivation.

Again here, the conclusion's contexts are normalized version of the ones from the premiss. Also, the results about contexts will sometimes be mentioned simply as "context results" in the proof, without stating explicitly the four results.

Case $\Upsilon_1; \Phi_1; \Psi \vdash \lambda.m \Leftarrow (\Pi A.B, \sigma) \,/_\rho\, (\Upsilon_2; \Phi_2; \lambda.M)$

$\Upsilon_1; \Phi_1; \Psi, \mathsf{clo}(A, \sigma) \vdash m \Leftarrow (B, \mathsf{dot1}\ \sigma) \,/_\rho\, (\Upsilon_2; \Phi_2; M)$ inversion

$\Upsilon_2; \Phi_2; [\![\rho]\!](\Psi, [\sigma]A) \vdash M \Leftarrow [\![\rho]\!][\mathsf{dot1}\ \sigma]B$ i.h.1

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi, [\![\rho]\!]([\sigma]A) \vdash M \Leftarrow [\![\rho]\!][\mathsf{dot1}\ \sigma]B$ def. of contextual subs.

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash M \Leftarrow \Pi([\![\rho]\!][\sigma]A).([\![\rho]\!][\mathsf{dot1}\ \sigma]B)$ typing rule

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash M \Leftarrow [\![\rho]\!][\sigma](\Pi A.B)$ def. of contextual subs., def. of subs.

context results i.h.1

Case $\Upsilon_1; \Phi_1; \Psi \vdash h \cdot s \Leftarrow (\Pi A_n \ldots A_1.P, \sigma_n) \,/_\rho\, (\Upsilon_2; \Phi_2; \lambda. \ldots \lambda.R)$

$h' = [\uparrow^{0,n}]h$ inversion

$s' = [\uparrow^{0,n}]s$ inversion

$\sigma_{i-1} = \mathsf{dot1}\ \sigma_i$ inversion

$A'_i = \mathsf{clo}(A_i, \sigma_i)$ inversion

$\Upsilon_1; \Phi_1; \Psi, A'_n, \ldots, A'_1 \vdash h' \cdot (s'@((n \cdot \mathsf{nil}) \ldots (1 \cdot \mathsf{nil})\mathsf{nil})) \Leftarrow (P, \sigma_0) \,/_\rho\, (\Upsilon_2; \Phi_2; R)$ inversion

$\Upsilon_2; \Phi_2; [\![\rho]\!](\Psi, [\sigma_n]A_n, \ldots, [\sigma_1]A_1) \vdash R \Leftarrow [\![\rho]\!][\sigma_0]P$ i.h.1

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash \lambda. \ldots \lambda.R \Leftarrow [\![\rho]\!][\sigma_n](\Pi A_n \ldots A_1.P)$

typing rule (n times), using def. of subs.

context results i.h.1

Case $\Upsilon_1; \Phi_1; \Psi \vdash \mathsf{c} \cdot s \Leftarrow (P, \sigma) \,/_\rho\, (\Upsilon_2; \Phi_2; \mathsf{c} \cdot S)$

$\Sigma(\mathsf{c}) = (A, i)$ inversion

$\Upsilon_1; \Phi_1; \Psi \vdash^i s : (A, \mathsf{id}) \Leftarrow (P, \sigma) \,/_\rho\, (\Upsilon_2; \Phi_2; S)$ inversion

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash S : [\![\rho]\!]A \Rightarrow [\![\rho]\!][\sigma]P$ i.h.2

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash \mathsf{c} \cdot S \Leftarrow [\![\rho]\!][\sigma]P$ typing rule

context results i.h.2

Case $\Upsilon_1; \Phi_1; \Psi \vdash x \cdot s \Leftarrow (P, \sigma) \mathbin{/_\rho} (\Upsilon_2; \Phi_2; x \cdot S)$

$\Psi(x) = (A, \uparrow^{0,x})$ inversion

$[\![\rho]\!]\Psi(x) = [\![\rho]\!](A, \uparrow^{0,x})$ prop. of contextual subs.

$\Upsilon_1; \Phi_1; \Psi \vdash s : (\mathsf{clo}(A, \uparrow^{0,x}), \mathsf{id}) \Leftarrow (P, \sigma) \mathbin{/_\rho} (\Upsilon_2; \Phi_2; S)$ inversion

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash S : [\![\rho]\!][\uparrow^{0,x}]A \Rightarrow [\![\rho]\!][\sigma]P$ i.h.3

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash x \cdot S \Leftarrow [\![\rho]\!][\sigma]P$ typing rule

context results i.h.3

Case $\Upsilon_1; \Phi_1; \Psi \vdash X \cdot s \Leftarrow (P, \sigma) \mathbin{/_\rho} (\Upsilon_2; [\![\rho]\!]\Phi_1, X : [\![\rho]\!]\mathsf{clo}(A, \sigma_1); [\![\rho]\!]X \cdot S)$

$\Upsilon_1; \Phi_1; \Psi \vdash (A, \sigma_1) \mid [\cdot]^{-1} \Rightarrow (\Upsilon_2; \rho)$ inversion

$[\![\rho]\!]\Phi_1 \subseteq [\![\rho]\!]\Phi_1$ prop. of $\subseteq$

$\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ and $\Upsilon_2 \vdash [\![\rho]\!]\Phi_1$ fctx and $\vdash_{[\![\rho]\!]\Phi_1} \Upsilon_2$ mctx prop. of pruning

s is a pattern spine inversion

$\Upsilon_1; \Phi_1; \Psi \vdash s \Leftarrow (P, \sigma) \mathbin{/} ((A, \sigma_1); S)$ inversion

$\Upsilon_1; \Phi_1; \Psi \vdash S : [\sigma_1]A \Rightarrow [\sigma]P$ i.h.4

$\Upsilon_2; [\![\rho]\!]\Phi_1; [\![\rho]\!]\Psi \vdash [\![\rho]\!]S : [\![\rho]\!][\sigma_1]A \Rightarrow [\![\rho]\!][\sigma]P$ prop. of contextual subs.

$X \notin \Phi_1$ inversion

$(\Phi_1, X{:}\mathsf{clo}(A, \sigma_1))(X) = \mathsf{clo}(A, \sigma_1)$ by previous line (Φ is unordered)

$\Upsilon_1 \vdash \Phi_1$ fctx assumption

$\Upsilon_1; \Phi_1; \Psi \vdash [\sigma_1]A \Leftarrow \mathsf{type}$ i.h.4

$\Upsilon_2; [\![\rho]\!]\Phi_1; \cdot \vdash [\![\rho]\!][\sigma_1]A \Leftarrow \mathsf{type}$ prop. of pruning

$\Upsilon_2 \vdash ([\![\rho]\!]\Phi_1), X{:}\mathsf{clo}([\![\rho]\!]A, [\![\rho]\!]\sigma_1)$ fctx typing rule

$[\![\rho]\!](\Phi_1, X{:}[\sigma_1]A)(X) = [\![\rho]\!][\sigma_1]A$ $X \notin \Phi_1$

$\Upsilon_2; [\![\rho]\!](\Phi_1, X{:}[\sigma_1]A); [\![\rho]\!]\Psi \vdash [\![\rho]\!]S : [\![\rho]\!][\sigma_1]A \Rightarrow [\![\rho]\!][\sigma]P$ weakening

$\Upsilon_2; [\![\rho]\!](\Phi_1, X{:}[\sigma_1]A); [\![\rho]\!]\Psi \vdash [\![\rho]\!](X \cdot S) \Leftarrow [\![\rho]\!][\sigma]P$ typing rule

Case $\Upsilon_1; \Phi_1; \Psi \vdash X \cdot s \Leftarrow (P, \sigma) \mathbin{/_\rho} (\Upsilon_2; \Phi_2; X \cdot S)$

$\Upsilon_1; \Phi_1; \Psi \vdash s : (A, \mathsf{id}) \Leftarrow (P, \sigma) \mathbin{/_\rho} (\Upsilon_2; \Phi_2; S)$ inversion

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash S : [\![\rho]\!][\mathsf{id}]A \Rightarrow [\![\rho]\!][\sigma]P$ i.h.3

context results i.h.3

$\Phi_1(X) = A$ inversion

$[\![\rho]\!]\Phi_1(X) = [\![\rho]\!]A$ prop. of contextual subs.

$\Phi_2(X) = [\![\rho]\!]A$ $[\![\rho]\!]\Phi_1 \subseteq \Phi_2$

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash X \cdot S \Leftarrow [\![\rho]\!][\sigma]P$ typing rule

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

$(\Upsilon_1, u{::}([\sigma]P)[\Psi])(u) = ([\sigma]P)[\Psi]$

$\Upsilon_1, u{::}[\sigma]\mathsf{clo}(P, \sigma)[\Psi]; \Phi_1; \Psi \vdash \mathsf{id} \Leftarrow \Psi$ lemma 10

$[\mathsf{id}][\sigma]P = [\sigma]P$ $\circ$ rule

$\Upsilon_1, u{::}([\sigma]P)[\Psi]; \Phi_1; \Psi \vdash u[\mathsf{id}] \Leftarrow [\sigma]P$ typing rule

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

property of $\mathsf{id}(\Upsilon)$

context results property of $\mathsf{id}(\Upsilon)$

Case $\Upsilon_1; \Phi_1; \Psi \vdash \mathsf{nil} : (\mathsf{a} \cdot S', \sigma_1) \Leftarrow (\mathsf{a} \cdot S, \sigma_2) \mathbin{/_\rho} (\Upsilon_2; [\![\rho]\!]\Phi_1; \mathsf{nil})$

$\Upsilon_1; \Phi_1; \Psi \vdash (\mathsf{a} \cdot S', \sigma_1) \doteqdot (\mathsf{a} \cdot S, \sigma_2) \;/\; (\rho; \Upsilon_2)$ inversion

$\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1$ property of $\doteqdot$

$\vdash_{\Phi_2} \Upsilon_2\ \mathsf{mctx}$ property of $\doteqdot$

$[\![\rho]\!]\Phi_1 \subseteq [\![\rho]\!]\Phi_1$ prop. of $\subseteq$

$\Upsilon_2 \vdash [\![\rho]\!]\Phi_1\ \mathsf{fctx}$ prop. of contextual subs. and well-formedness of Φ_1

$\Upsilon_2; [\![\rho]\!]\Phi_1; [\![\rho]\!]\Psi \vdash \mathsf{nil} : [\![\rho]\!][\sigma_1](\mathsf{a} \cdot S') \Rightarrow [\![\rho]\!][\sigma_1](\mathsf{a} \cdot S')$ typing rule

$\Upsilon_2; [\![\rho]\!]\Phi_1; [\![\rho]\!]\Psi \vdash \mathsf{nil} : [\![\rho]\!][\sigma_1](\mathsf{a} \cdot S') \Rightarrow [\![\rho]\!][\sigma_2](\mathsf{a} \cdot S)$ property of $\doteqdot$

Case $\Upsilon_1; \Phi_1; \Psi \vdash m\ s : (\Pi A.B, \sigma_1) \Leftarrow (P, \sigma_2) \mathbin{/_{\rho_1 \circ\ \rho_2}} (\Upsilon_3; \Phi_3; [\![\rho_2]\!]M\ S)$

$\Upsilon_1; \Phi_1; \Psi \vdash m \Leftarrow (A, \sigma_1) \mathbin{/_{\rho_1}} (\Upsilon_2; \Phi_2; M)$ inversion

$\Upsilon_2; \Phi_2; [\![\rho_1]\!]\Psi \vdash M \Leftarrow [\![\rho_1]\!][\sigma_1]A$ i.h.3

$[\![\rho_1]\!]\Phi_1 \subseteq \Phi_2$, $\Upsilon_2 \vdash_{\Phi_2} \rho_1 \Leftarrow \Upsilon_1$, $\Upsilon_2 \vdash \Phi_2\ \mathsf{fctx}$ and $\vdash_{\Phi_2} \Upsilon_2\ \mathsf{mctx}$ i.h.3

$\Upsilon_2; \Phi_2; [\![\rho_1]\!]\Psi \vdash s : ([\![\rho_1]\!]B, ([\![\rho_1]\!]\sigma_1; M)) \Leftarrow [\![\rho_1]\!](P, \sigma_2) \,/_{\rho_2}\, (\Upsilon_3; \Phi_3; S)$ inversion

$\Upsilon_3; \Phi_3; [\![\rho_1 \circ\ \rho_2]\!]\Psi \vdash S : [\![\rho_2]\!][[\![\rho_1]\!]\sigma_1; M]([\![\rho_1]\!]B) \Rightarrow [\![\rho_1 \circ\ \rho_2]\!][\sigma_2]P$ i.h.3

$\Upsilon_3; \Phi_3; [\![\rho_1 \circ\ \rho_2]\!]\Psi \vdash S : [[\![\rho_1 \circ\ \rho_2]\!]\sigma_1; [\![\rho_2]\!]M]([\![\rho_1 \circ\ \rho_2]\!]B) \Rightarrow [\![\rho_1 \circ\ \rho_2]\!][\sigma_2]P$

def. of contextual subs.

$\Upsilon_3; \Phi_3; [\![\rho_1 \circ \rho_2]\!]\Psi \vdash S : [\mathsf{id}; [\![\rho_2]\!]M][\mathsf{dot1}\ [\![\rho_1 \circ \rho_2]\!]\sigma_1]([\![\rho_1 \circ \rho_2]\!]B) \Rightarrow [\![\rho_1 \circ \rho_2]\!][\sigma_2]P$

$\circ$ rules

$\Upsilon_3; \Phi_3; [\![\rho_1 \circ\ \rho_2]\!]\Psi \vdash S : [\mathsf{id}; [\![\rho_2]\!]M][\![\rho_1 \circ\ \rho_2]\!][\mathsf{dot1}\ \sigma_1]B \Rightarrow [\![\rho_1 \circ\ \rho_2]\!][\sigma_2]P$

property of contextual subs.

$[\![\rho_2]\!]\Phi_2 \subseteq \Phi_3$, $\Upsilon_3 \vdash_{\Phi_2} \rho_2 \Leftarrow \Upsilon_2$, $\Upsilon_3 \vdash \Phi_3\ \mathsf{fctx}$ and $\vdash_{\Phi_3} \Upsilon_3\ \mathsf{mctx}$ i.h.3

$\Upsilon_3; \Phi_3; [\![\rho_1 \circ\ \rho_2]\!]\Psi \vdash [\![\rho_2]\!]M \Leftarrow [\![\rho_1 \circ\ \rho_2]\!][\sigma_1]A$ property of contextual subs.

$\Upsilon_3; \Phi_3; [\![\rho_1 \circ\ \rho_2]\!]\Psi \vdash [\![\rho_2]\!]M\ S : \Pi([\![\rho_1 \circ\ \rho_2]\!]\sigma_1]A).([\![\rho_1 \circ\ \rho_2]\!][\mathsf{dot1}\ \sigma_1]B) \Rightarrow [\![\rho_1 \circ\ \rho_2]\!][\sigma_2]P$ typing rule

$\Upsilon_3; \Phi_3; [\![\rho_1 \circ\ \rho_2]\!]\Psi \vdash [\![\rho_2]\!]M\ S : [[\![\rho_1 \circ\ \rho_2]\!]\sigma_1](\Pi A.B) \Rightarrow [\![\rho_1 \circ\ \rho_2]\!][\sigma_2]P$

property of subs. and contextual subs.

$[\![\rho_1 \circ\ \rho_2]\!]\Phi_1 \subseteq \Phi_3$ and $\Upsilon_3 \vdash_{\Phi_3} \rho_1 \circ\ \rho_2 \Leftarrow \Upsilon_1$ property of $\circ$ on contextual subs.

Case $\Upsilon_1; \Phi_1; \Psi \vdash \mathsf{nil} \Leftarrow (P, \sigma)\ /\ ((P, \sigma); \mathsf{nil})$

$\Upsilon_1; \Phi_1; \Psi \vdash \mathsf{nil} : [\sigma]P \Rightarrow [\sigma]P$ typing rule

$\Upsilon_1; \Phi_1; \Psi \vdash (P, \sigma) \Leftarrow \mathsf{type}$ premiss of the reconstruction judgement

$\Upsilon_1; \Phi_1; \Psi \vdash [\sigma]P \Leftarrow \mathsf{type}$ lemma 19

Case $\Upsilon_1; \Phi_1; \Psi \vdash m\ s \Leftarrow (P, \sigma)\ /\ ((\Pi\mathsf{clo}(A, \uparrow^{0,x}).\mathsf{clo}(B, \sigma'' \circ\ \uparrow^{0,1} \circ \sigma'), \mathsf{id}); M\ S)$

$\mathsf{contract}_0\ m\ /\ x$ inversion

$\Psi(x) = (A, \uparrow^{0,x})$ inversion

$\Psi \vdash^0 \mathsf{expand}\ x\ :\ (A, \uparrow^{0,x})\ /\ M$ inversion

$\Psi \vdash (M, \mathsf{id}) \Leftarrow (A, \uparrow^{0,x})$ lemma 12

$\Psi \vdash [\mathsf{id}]M \Leftarrow [\uparrow^{0,x}]A$ lemma 19

$\sigma' = \uparrow^{0,x+1}, 1, x, \ldots, 1$ inversion

$\Upsilon_1; \Phi_1; \Psi \vdash s \Leftarrow (P, \sigma)\ /\ ((B, \sigma''); S)$ inversion

$\Upsilon_1; \Phi_1; \Psi \vdash S : [\sigma'']B \Rightarrow [\sigma]P$ i.h.4

$(\uparrow^{0,1} \circ \sigma' \circ\ (\mathsf{id}, x)) = \mathsf{id}$ def. of $\circ$ and typing rule

$\Upsilon_1; \Phi_1; \Psi \vdash S : [\uparrow^{0,1} \circ \sigma' \circ\ (\mathsf{id}, x)][\sigma'']B \Rightarrow [\sigma]P$ $\sigma \circ \mathsf{id} = \sigma$

$\Upsilon_1; \Phi_1; \Psi \vdash S : [\mathsf{id}, x][\sigma'' \circ\ \uparrow^{0,1} \circ \sigma']B \Rightarrow [\sigma]P$ associativity of $\circ$

$\Upsilon_1; \Phi_1; \Psi \vdash S : [\mathsf{id}; M][\sigma'' \circ\ \uparrow^{0,1} \circ \sigma']B \Rightarrow [\sigma]P$ lemma 13

$\Upsilon_1; \Phi_1; \Psi \vdash M\ S : \Pi[\uparrow^{0,x}]A.[\sigma'' \circ\ \uparrow^{0,1} \circ \sigma']B \Rightarrow [\sigma]P$ typing rule

$\Upsilon_1; \Phi_1; \Psi \vdash (A, \uparrow^{0,x}) \Leftarrow \mathsf{type}$ assumption

$\Upsilon_1; \Phi_1; \Psi \vdash [\uparrow^{0,x}]A \Leftarrow \mathsf{type}$ lemma 19

$\Upsilon_1; \Phi_1; \Psi \vdash [\sigma'']B \Leftarrow \mathsf{type}$ i.h.4

$\Upsilon_1; \Phi_1; \Psi, [\uparrow^{0,x}]A \vdash [\uparrow^{0,1}][\sigma'']B \Leftarrow \mathsf{type}$ weakening

$\Upsilon_1; \Phi_1; \Psi, [\uparrow^{0,x}]A \vdash [\sigma'][\uparrow^{0,1}][\sigma'']B \Leftarrow \mathsf{type}$ $\Psi(x) = \Psi(1)$

$\Upsilon_1; \Phi_1; \Psi \vdash \Pi[\uparrow^{0,x}]A.[\sigma'' \circ\ \uparrow^{0,1} \circ \sigma']B \Leftarrow \mathsf{type}$ typing rule

Case $\Upsilon_1; \Phi_1; \Psi \vdash^0 s : (A, \sigma_1) \Leftarrow (P, \sigma_2) \mathbin{/_\rho} (\Upsilon_2; \Phi_2; S)$

$\Upsilon_1; \Phi_1; \Psi \vdash s : (A, \sigma_1) \Leftarrow (P, \sigma_2) \mathbin{/_\rho} (\Upsilon_2; \Phi_2; S)$ inversion

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash S : [\![\rho]\!][\sigma_1]A \Rightarrow [\![\rho]\!][\sigma_2]P$ i.h.3

context results i.h.3

Case $\Upsilon_1; \Phi_1; \Psi \vdash^i s : (\Pi A.B, \sigma_1) \Leftarrow (P, \sigma_2) \mathbin{/_{\rho \setminus (R/u)}} (\Upsilon_2; \Phi_2; [\![\rho]\!]M\ S)$

$\Upsilon_1, u{::}Q[\Psi']; \Phi_1; \Psi \vdash^{i-1} s : (B, (\sigma_1, M)) \Leftarrow (P, \sigma_2) \mathbin{/_\rho} (\Upsilon_2; \Phi_2; S)$ inversion

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash S : [\![\rho]\!][\sigma_1, M]B \Rightarrow [\![\rho]\!][\sigma_2]P$ i.h.2

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash S : [\![\rho]\!][\mathsf{id}, M][\mathsf{dot1}\ \sigma_1]B \Rightarrow [\![\rho]\!][\sigma_2]P$ $\circ$ rules & def. of $\mathsf{dot1}$

$[\![\rho]\!]\Phi_1 \subseteq \Phi_2$ and $\Upsilon_2 \vdash_{\Phi_2} \rho \Leftarrow \Upsilon_1, u{::}Q[\Psi']$ and $\Upsilon_2 \vdash \Phi_2\ \mathsf{fctx}$

and $\vdash_{\Phi_2} \Upsilon_2\ \mathsf{mctx}$ i.h.2

$\mathsf{lower}(\Upsilon_1; \Phi; \Psi \vdash (A, \sigma_1)) = (M, u{::}Q[\Psi'])$ inversion

$\Upsilon_1, u{::}Q[\Psi']; \Phi_1; \Psi \vdash (M, \mathsf{id}) \Leftarrow (A, \sigma_1)$ lemma 15

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash [\![\rho]\!](M, \mathsf{id}) \Leftarrow [\![\rho]\!](A, \sigma_1)$ prop. of contextual subs.

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash [\![\rho]\!][\mathsf{id}]M \Leftarrow [\![\rho]\!][\sigma_1]A$ lemma 19

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash [\![\rho]\!]M\ S : [\![\rho]\!](\Pi[\sigma_1]A.[\mathsf{dot1}\ \sigma_1]B) \Rightarrow [\![\rho]\!][\sigma_2]P$ typing rule

$\Upsilon_2; \Phi_2; [\![\rho]\!]\Psi \vdash [\![\rho]\!]M\ S : [\![\rho]\!][\sigma_1](\Pi A.B) \Rightarrow [\![\rho]\!][\sigma_2]P$ property of subs.

let $\rho' = \rho \setminus (R/u)$

$\Upsilon_2; \Phi_2; [\![\rho']\!]\Psi \vdash [\![\rho]\!]M\ S : [\![\rho']\!][\sigma_1](\Pi A.B) \Rightarrow [\![\rho']\!][\sigma_2]P$ $u \notin \Upsilon_1$

$[\![\rho']\!]\Phi_1 \subseteq \Phi_2$ def. of contextual subs. ($u \notin \Phi_1$)

$\Upsilon_2 \vdash_{\Phi_2} \rho' \Leftarrow \Upsilon_1$ $u \notin \Upsilon_1$

$\square$