Bidirectional Elaboration of Dependently Typed Programs

Francisco Ferreira Brigitte Pientka
McGill University
{fferre8,bpientka}@cs.mcgill.ca

Abstract

Dependently typed programming languages allow programmers to express a rich set of invariants and verify them statically via type checking. To make programming with dependent types practical, dependently typed systems provide a compact language for programmers where one can omit some arguments, called implicit, which can be inferred. This source language is then usually elaborated into a core language where type checking and fundamental properties such as normalization are well understood. Unfortunately, this elaboration is rarely specified and in general is ill-understood. This makes it not only difficult for programmers to understand why a given program fails to type check, but also is one of the reasons that implementing dependently typed programming systems remains a black art known only to a few.

In this paper, we specify the design of a source language for a dependently typed programming language where we separate the language of programs from the language of types and terms occurring in types. We then give a bi-directional elaboration algorithm to translate source terms where implicit arguments can be omitted to a fully explicit core language and prove soundness of our elaboration. Our framework provides post-hoc explanation for elaboration found in the programming and proof environment, Beluga.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords dependent types, type reconstruction

1. Introduction

Dependently typed programming languages such as Agda [Norell 2007], Epigram [McBride and McKinna 2004] or Idris [Brady 2013] allow programmers to express a rich set of properties and statically verify them via type checking. To make programming with dependent types practical, all these systems provide a source language where programmers can omit (implicit) arguments which can be reasonably easy inferred and elaborate the source language into a well-understood core language, an idea going back to Pollack [1990]. However, this elaboration is rarely specified formally for dependently typed languages which support recursion and pattern matching. For Agda, Norrel [2007] for example describes a bi-directional type inference algorithm, but does not treat recursion and pattern matching. For Idris, Brady [2013] describes the elaboration between source and target, but no theoretical properties such as soundness are established. A notable exception is the work by Asperti et.al [2012] on describing a sound bi-directional elaboration algorithm for the Calculus of (Co)Inductive Constructions (CCIC) implemented in Matita.

In this paper, we investigate the design of a source language for dependently typed programming where we separate the language of programs from the language of types and terms occurring in types similar to indexed type systems (see [Zenger 1997; Xi and Pfenning 1999]); however, unlike these aforementioned systems, we allow pattern matching on index objects. As a consequence, we cannot simply erase our implicit arguments and obtain a program which is simply typed. Specifically, our source language is inspired by the Beluga language [Pientka 2008; Pientka and Dunfield 2010; Cave and Pientka 2012] where we specify formal systems in the logical framework LF [Harper et al. 1993] (our index language) and write recursive programs about LF objects using pattern matching. The main contribution of this paper the design of a source language for dependently typed programs where we omit implicit arguments together with a sound bi-directional elaboration algorithm from the source language to a fully explicit core language. Throughout our development, we will keep the index language abstract and state abstractly our requirements such as decidability of equality and typing. This will make our framework applicable to any language satisfying our stated requirements.

A central question when elaborating dependently typed languages is what arguments may the programmer omit. In dependently-typed systems such as Agda or Coq, the programmer declares constants of a given (closed) type and labels arguments that can be freely omitted when subsequently using the constant. Both, Coq and Agda, give the user the possibility to locally override the implicit arguments and provide instantiations explicitly.

In contrast, we follow here a simple, lightweight recipe which comes from the implementation of the logical framework *Elf* [Pfenning 1989] and its successor *Twelf* [Pfenning and Schürmann 1999]: *programmers may leave some index variables free when declaring a constant of a given type; elaboration of the type will abstract over these free variables at the outside; when subsequently using this constant, the user must omit passing arguments for those index variables which were left free in the original declaration.* Following this recipe, elaboration of terms and types in the logical framework has been described in Pientka [2013]. Here, we will consider a dependently typed functional programming language which supports pattern matching on index objects.

The key challenge in elaborating recursive programs which support case-analysis is that pattern matching in the dependently typed setting refines index arguments and hence refines types. In contrast to systems such as Coq and Agda, where we must annotate case-expressions with an invariant, i.e. the type of the scrutinee, and the return type, our source language does not require such annotations. Instead we will infer the type of the scrutinee and for each branch, we infer the type of the pattern and compute how the pattern refines the type of the scrutinee. This makes our source language lightweight. Our elaboration of source expressions to target expressions is type-directed, inferring omitted arguments and producing a closed well-typed target program. Finallly, we prove soundness of our elaboration, i.e. if elaboration succeeds our resulting program type checks in our core language. Our framework provides post-hoc explanation for elaboration found in the programming and proof environment, Beluga [Pientka 2008; Cave and Pientka 2012], where we use as the index domain terms specified in the logical framework LF [Harper et al. 1993].

The paper is organized as follows: We first give the grammar of our source language. Showing two example programs, we explain informally what elaboration does. We then revisit our core language, describe the elaboration algorithm formally and prove soundness. We conclude with a discussion of related and future work.

2. Source language

We consider here a dependently typed language where types are indexed by terms from an index domain. Our language is similar to Beluga [Pientka and Dunfield 2010], a dependently typed programming environment where we specify formal systems in the logical framework LF and we can embed LF objects into computation-level types and computation-level programs which analyze and pattern match on LF objects. However, in our description, as in for example [Cave and Pientka 2012], we will keep the index domain abstract, but only assume that equality in the index domain is decidable and unification algorithms exist. This will allow us to focus on the essential challenges when elaborating a dependently typed language in the presence of pattern matching.

We describe the *source language* that allows programmers to omit some arguments in Fig. 1. As a convention we will use lowercase c to refer to index level objects, lowercase u for index level types, and upper case letters X, Y for index-variables. Index objects can be embedded into computation expressions by writing $[c]$. Our language supports functions ($\textbf{fn}\, x{\Rightarrow}e$), dependent functions ($\lambda\, X{\Rightarrow}e$), function application ($e_1\; e_2$), dependent function application ($e_1\; [c]$), and case-expressions. We also support writing underscore ($\_$) instead of providing explicitly the index argument in a dependent function application ($e\;\_$). Note that we are overloading syntax: we write $e\;[c]$ to describe the application of the expression e of type $[u] \rightarrow t$ to the expression $[c]$; we also write $e\;[c]$ to describe the dependent application of the expression e of type $\{X{:}u\}t$ to the (unboxed) index object c.

We may write type annotations anywhere in the program ($e{:}t$ and in patterns $pat{:}t$); type annotations are particularly useful to make explicit the type of a sub-expression and name index variables occurring in the type. This allows us to resurrect index variables which are kept implicit. In patterns, type annotations are useful since they provide hints to type elaboration regarding the type of pattern variables.

A program signature Σ consists of type declarations ($\mathbf{a}{:}k$ and $\mathbf{c}{:}t$) and declaration of recursive functions ($\textbf{rec}\, f{:}t = e$). This can be extended to allow mutual recursive functions in a straightforward way.

One may think of our source language as the language obtained after parsing after where for example let-expressions have been translated into case-expression with one branch.

Types for computations include non-dependent function types ($t_1 \rightarrow t_2$) and dependent function types ($\{X{:}u\}t$); we can also embed index types into computation types via $[u]$ and index computation-level types by an index domain written as $\mathbf{a}\,\overrightarrow{[c]}$. We also include the grammar for computation-level kinds which emphasizes that computation-level types can only be indexed by terms from an index domain u. We write `ctype` (i.e. computation-level type) for the base kind, since we will use **type** for kinds of the index domain.

Kinds	k	$::=$	$\texttt{ctype} \mid \{X{:}u\}\,K$
Atomic Types	p	$::=$	$\mathbf{a}\,\overrightarrow{[c]}$
Types	t	$::=$	$p \mid [u] \mid \{X{:}u\}\,t \mid t_1 \rightarrow t_2$
Expressions	e	$::=$	$\textbf{fn}\, x{\Rightarrow}e \mid \lambda\, X{\Rightarrow}e \mid x \mid \mathbf{c} \mid [c] \mid$
			$e_1\; e_2 \mid e_1\; [c] \mid e\;\_ \mid \textbf{case}\; e\; \textbf{of}\; \vec{b} \mid e{:}t$
Branches	$\vec{b}$	$::=$	$b \mid (b \mid \vec{b})$
Branch	b	$::=$	$pat \mapsto e$
Pattern	pat	$::=$	$x \mid [c] \mid \mathbf{c}\,\overrightarrow{pat} \mid pat{:}t$
Declarations	d	$::=$	$\textbf{rec}\, f{:}t = e \mid \mathbf{c}{:}t \mid \mathbf{a}{:}k$

Figure 1. Grammar of Source Language

We note that we do only support one form of dependent function type $\{X{:}u\}t$; the source language does not provide any means for programmers to mark a given dependently typed variable as implicit as for example in Agda. Instead, we will allow programmers to leave some index variables occurring in computation-level types free; elaboration will then infer their types and abstract over them explicitly at the outside. The programmer must subsequently omit providing instantiation for those "free" variables. We will explain this idea more concretely below.

2.1 Well-formed source expressions

Before elaborating source expressions, we state when a given source expression is accepted as a well-formed expression. In particular, it will highlight that free index variables are only allowed in declarations when specifying kinds and declaring the type of constants and recursive functions. We use δ to describe the list of index variables and γ the list of program variables. We rely on two judgments from the index language:

$\delta \vdash\; c\;\mathrm{wf}$	Index object c is well formed and closed with respect to δ
$\delta \vdash_f\; c\;\mathrm{wf}$	Index object c is well formed with respect to δ and may contain free index variables

We describe declaratively the well-formedness of declarations and source expressions in Fig. 2. For brevity, we omit the full definition of well-formedness for kinds and types which is as expected and straightforward.

In branches, pattern variables from γ must occur linearly while we put no such requirement on variables from our index language listed in δ.

2.2 Some example programs

2.2.1 Translating untyped terms to intrinsically typed terms

We implement a program to translate a simple language with numbers, booleans and some primitive operations to its typed counterpart to illustrate declaring an index domain, using index computation-level types, and explaining the use and need to pattern match on index objects. We first define the untyped version of our language by the recursive datatype `UTm`. Note use of the keyword `ctype` to define a computation-level recursive data-type.

```
datatype UTm : ctype =
| UNum  : Nat→UTm
| UPlus : UTm→UTm→UTm
| UTrue : UTm
| UFalse: UTm
```

$\boxed{\vdash d \text{ wf}}$ Declaration d is well-formed

$$\frac{\cdot;\ f \vdash e \text{ wf} \quad \cdot \vdash_f t \text{ wf}}{\vdash \textbf{rec}\ f{:}t = e \text{ wf}}\ \texttt{wf-rec} \qquad \frac{\cdot \vdash_f t \text{ wf}}{\vdash c : t \text{ wf}}\ \texttt{wf-types} \qquad \frac{\cdot \vdash_f k \text{ wf}}{\vdash a : k \text{ wf}}\ \texttt{wf-kinds}$$

$\boxed{\delta;\gamma \vdash e \text{ wf}}$ Expression e is well-formed in context δ and γ

$$\frac{\delta;\gamma,x \vdash e \text{ wf}}{\delta;\gamma \vdash \textbf{fn}\ x{\Rightarrow}e \text{ wf}}\ \texttt{wf-fn} \qquad \frac{\delta,X;\gamma \vdash e \text{ wf}}{\delta;\gamma \vdash \lambda\, X{\Rightarrow}e \text{ wf}}\ \texttt{wf-mlam} \qquad \frac{\delta;\gamma \vdash e_1 \text{ wf} \quad \delta;\gamma \vdash e_2 \text{ wf}}{\delta;\gamma \vdash e_1\, e_2 \text{ wf}}\ \texttt{wf-app}$$

$$\frac{\delta;\gamma \vdash e_1 \text{ wf} \quad \delta \vdash c \text{ wf}}{\delta;\gamma \vdash e\,[c] \text{ wf}}\ \texttt{wf-app-explicit} \qquad \frac{\delta;\gamma \vdash e_1 \text{ wf}}{\delta;\gamma \vdash e_1\ \_ \text{ wf}}\ \texttt{wf-apph} \qquad \frac{\delta \vdash c \text{ wf}}{\delta;\gamma \vdash [c] \text{ wf}}\ \texttt{wf-box}$$

$$\frac{\delta;\gamma \vdash e \text{ wf} \quad \text{for all } b_n \text{ in } \vec{b}\,.\ \delta;\gamma \vdash b_n \text{ wf}}{\delta;\gamma \vdash \textbf{case}\ e\ \textbf{of}\ \vec{b} \text{ wf}}\ \texttt{wf-case} \qquad \frac{x \in \gamma}{\delta;\gamma \vdash x \text{ wf}}\ \texttt{wf-var} \qquad \frac{\delta;\gamma \vdash e \text{ wf} \quad \delta \vdash t \text{ wf}}{\delta;\gamma \vdash e{:}t \text{ wf}}\ \texttt{wf-ann}$$

$\boxed{\delta;\gamma \vdash pat \mapsto e \text{ wf}}$ Branch is well-formed in δ and γ

$$\frac{\delta';\gamma' \vdash pat \text{ wf} \quad \delta,\delta';\gamma,\gamma' \vdash e \text{ wf}}{\delta;\gamma \vdash pat \mapsto e \text{ wf}}\ \texttt{wf-branch}$$

$\boxed{\delta;\gamma \vdash pat \text{ wf}}$ Pattern pat is well-formed synthesizing a context δ for index variables and a context γ for pattern variables

$$\frac{}{\delta;x \vdash x \text{ wf}}\ \texttt{wf-p-var} \qquad \frac{\delta \vdash c \text{ wf}}{\delta;\cdot \vdash [c] \text{ wf}}\ \texttt{wf-p-i} \qquad \frac{\text{for all } p_i \text{ in } \overrightarrow{Pat}.\ \ \delta;\gamma_i \vdash p_i \text{ wf}}{\delta;\gamma_1,\ldots,\gamma_n \vdash \mathbf{c}\,\overrightarrow{Pat} \text{ wf}}\ \texttt{wf-p-con} \qquad \frac{\delta;\gamma \vdash pat \text{ wf} \quad \delta \vdash t \text{ wf}}{\delta;\gamma \vdash pat{:}t \text{ wf}}\ \texttt{wf-p-ann}$$

Figure 2. Well-formed source expressions

```
| UNot  : UTm→UTm
| UIf   : UTm→UTm→UTm→UTm;
```

Terms can be of type `nat` for numbers or `bool` for booleans. Our goal is to define our language of typed terms using a computation-level type family `Tm` which is indexed by objects `nat` and `bool` which are constructors of our index type `tp`. Note that `tp` is declared as having the kind **type** which implies that this type lives at the index level and that we will be able to use it as an index for computation-level type families.

```
datatype tp : type =
| nat  : tp
| bool : tp;
```

Using indexed families we can now define the type `Tm` that specifies only type correct terms of the language, by indexing terms by their type using the index level type `tp`.

```
datatype Tm : [tp]→ctype =
| Num      : Nat       →Tm [nat]
| Plus     : Tm [nat]  →Tm [nat]→Tm [nat]
| True     : Tm [bool]
| False    : Tm [bool]
| Not      : Tm [bool] →Tm [bool]
| If       : Tm [bool] →Tm [T]→Tm [T]→Tm [T];
```

When the `Tm` family is elaborated, the free variable `T` in the `If` constructor will be abstracted over by an implicit Π-type, as in the Twelf [Pfenning and Schürmann 1999] tradition. Because `T` was left free by the programmer, the elaboration will add an implicit quantifier; when we use the constant `If`, we now must omit passing an instantiation for `T`. For example, we must write `(If True (Num 3) (Num 4))` and elaboration will infer that `T` must be `nat`.

One might ask how we can provide the type explicitly - this is possible indirectly by providing type annotations. For example, writing `If e (e1:TM[nat]) e2` will fix the type of `e1` to be `Tm [nat]`.

Our goal is to write a program to translate an untyped term `UTm` to its corresponding typed representation. Because this operation might fail for ill-typed `UTm` terms we need an option type to reflect the possibility of failure.

```
datatype TmOpt : ctype =
| None : TmOpt
| Some : {T : tp}Tm [T] → TmOpt;
```

A value of type `TmOpt` will either be empty (i.e. `None`) or some term of type `T`. We chose to make `T` explicit here by quantifying over it explicitly using the curly braces. When returning a `Tm` term, the program must now provide the instantiation of `T` in addition to the actual term.

So far we have declared types and constructors for our language. These declarations will be available in a global signature. The next step is to declare a function that will take untyped terms into typed terms if possible. Notice that for the function to be type correct it has to respect the specification provided in the declaration of the type `Tm`. We only show a few interesting cases below.

```
rec tc : UTm → TmOpt = fn e ⇒ case e of  ...
| UNum n ⇒ Some [nat] (Num n)
| UNot e ⇒ (case tc e of
  | Some [bool] x ⇒ Some [bool] (Not x)
  | other ⇒ None)
| UIf c e1 e2 ⇒  (case tc c of
  | Some [bool] c' ⇒ (case (tc e1 , tc e2) of
    | (Some [T] e1' , Some [T] e2') ⇒
       Some [T] (If c' e1' e2')
    | other ⇒ None )
  | other ⇒ None )
;
```

In the `tc` function the first four cases are completely straightforward. The case for negation (i.e. constructor `UNot`) is important because we need to pattern match on the result of type-checking the sub-expression `e` to refine it to type `bool` otherwise we cannot construct the intrinsically typed term, i.e. the constructor `Not` requires a boolean term. Additionally the case for `UIf` is also interesting because we not only need a boolean condition but we also need to have both branches of the `UIf` term to be of the same type. Again we use pattern matching on the indices to verify that the condition is of type `bool` but notably we use non-linear pattern matching to ensure that the type of the branches coincides (and to refine the types so we are able to construct a typed term with the constructor `If` that requires both branches to be of the same type). We note that `If` has an implicit argument which will be inferred during elaboration.

In the definition of type `TmOpt` we chose to explicitly quantify over `T`, however another option would have been to leave it implicit. When pattern matching on `Some e`, we would need to resurrect the type of the argument `e` to be able to inspect it and check whether it has the appropriate type. Again we employ type annotations in the code below to constrain the type of `e`.

```
| UIf c e1 e2 ⇒  (case tc c of
  | Some (c':Tm [bool]) ⇒  (case (tc e1, tc e2) of
    | (Some (e1':Tm [T]), Some (e2':Tm [T])) ⇒
        Some (If c' e1' e2')
    | other ⇒ None)
  | other ⇒ None)
```

2.2.2 Type-preserving evaluation

Our previous program used dependent types sparingly; most notably there were no dependent types in the type declaration given to the function `tc`. We now discuss the implementation of an evaluator, which evaluates type correct programs to values of the same type, to highlight writing dependently typed functions. Because we need to preserve the type information, we index the values by their types in the following manner:

```
datatype Val : [tp] → ctype =
| VNum  : Nat → Val [nat]
| VTrue : Val [bool]
| VFalse: Val [bool];
```

We define a type preserving evaluator below; again, we only show some interesting cases.

```
rec eval : Tm [T] → Val [T] = fn e ⇒ case e of ...
| Num n ⇒ VNum n
| Plus e1 e2 ⇒ (case (eval e1 , eval e2) of
  | (VNum x , VNum y) ⇒ VNum (add x y))
| Not e ⇒ (case eval e of
  | VTrue ⇒ VFalse
  | VFalse ⇒ VTrue)
| If e e1 e2 ⇒  (case eval e of
  | VTrue ⇒ eval e1
  | VFalse ⇒ eval e2)
;
```

First, we specify the type of the evaluation function as `Tm [T]` → `Val [T]` where `T` remains free. As a consequence, elaboration will infer its type and abstract over `T` at the outside. We now elaborate the body of the function against Π^e`T:tp. Tm [T]` → `Val [T]`. Elaboration translate the given program into a program in our core language which has type Π^e`T:tp. Tm [T]` → `Val [T]`. It will first need to introduce the appropriate dependent function abstraction in the program before we introduce the non-dependent function **fn** $x{\Rightarrow}e$. Moreover, we need to infer omitted arguments in the pattern in addition to inferring the type of pattern variables. Since `T` was left free in the type given to `eval`, we must also infer the omitted argument in the recursive calls to `eval`. Finally, we need to keep track of refinements the pattern match induces: our scrutinee has type `Tm [T]`; pattern matching against `Plus e1 e1` which has type `Tm [nat]` refines `T` to `nat`.

We will come back to this example and discuss the fully elaborated program in the next section.

Kinds	K	$::=$	$\Pi^e X{:}U.\,K \mid \Pi^i X{:}U.\,K \mid$ `ctype`
Atomic Types	P	$::=$	$\mathbf{a}\,\vec{C}$
Types	T	$::=$	$\Pi^e X{:}U.\,T \mid \Pi^i X{:}U.\,T$
			$P \mid [U] \mid T_1 \to T_2$
Expressions	E	$::=$	$\mathbf{fn}\,x{\Rightarrow}E \mid \lambda\,X{\Rightarrow}E$
			$\mid E_1\,E_2 \mid E_1\,[C] \mid [C] \mid$
			$\mathbf{case}\,E\,\mathbf{of}\,\vec{B} \mid x \mid E{:}T \mid \mathbf{c}$
Branches	$\vec{B}$	$::=$	$B \mid (B \mid \vec{B})$
Branch	B	$::=$	$\Pi\Delta;\Gamma.Pat{:}\theta \mapsto E$
Pattern	Pat	$::=$	$x \mid [C] \mid \mathbf{c}\,\overrightarrow{Pat}$
Declarations	D	$::=$	$\mathbf{c}{:}T \mid \mathbf{a}{:}K \mid \mathbf{rec}\,f{:}T = E$
Context	Γ	$::=$	$\cdot \mid \Gamma, x{:}T$
Index-Var-Context	Δ	$::=$	$\cdot \mid \Delta, X{:}U$
Refinement	θ	$::=$	$\cdot \mid \theta, C/X \mid \theta, X/X$

Figure 3. Target language

3. Target language

The *target language* is similar to the computational language described in Cave and Pientka [2012] which has a well developed meta-theory including descriptions of coverage [Dunfield and Pientka 2009] and termination [Pientka et al. 2014]. The target language (see Fig. 3), which is similar to our source language, is indexed by fully explicit terms of the index level language; we use C for fully explicit index level objects, and U for elaborated index types; index-variables occurring in the target language will be represented by capital letters such as X, Y. Moreover, we rely on a substitution which replaces index variables X with index objects. For convenience, we also allow explicit identity substitutions X/X, since not in all index languages X may be a well-defined term[1].

3.1 Typing of target language

The main difference between the source and target language is in the description of branches. In each branch, we make the type of the pattern variables (see context Γ) and variables occurring in index objects (see context Δ) explicit. Moreover, we associate each pattern with a refinement substitution θ which specifies how the given pattern refines the type of the scrutinee. The typing rules for our core language are given in Fig. 4. We again omit the rules for types and kind which are given in Cave and Pientka [2012].

We use a bidirectional type system for the target language which is similar to the one in Cave and Pientka [2012] but we simplify the presentation by omitting recursive types and instead we assume that constructors together with their types are declared in a signature Σ.

We rely on the fact that our index domain comes with rules which check that a given index object is well-typed. This is described by the judgment: $\Delta \vdash C : U$.

[1] When choosing as an index language, contextual LF for example, all index variables X are associated with a post-poned substitution.

$\boxed{\vdash D \text{ wf}}$ Target declaration D is well-formed

$$\frac{\cdot \vdash T \Leftarrow \mathsf{ctype} \quad \cdot\,;\ f{:}T \vdash E \Leftarrow T}{\vdash \mathbf{rec}\, f{:}T = E \text{ wf}}\ \texttt{t-rec} \qquad \frac{\cdot \vdash T \Leftarrow \mathsf{ctype}}{\vdash \mathbf{c}{:}T \text{ wf}}\ \texttt{t-type} \qquad \frac{\cdot \vdash K \Leftarrow \mathsf{kind}}{\vdash \mathbf{a}{:}K \text{ wf}}\ \texttt{t-kind}$$

$\boxed{\Delta; \Gamma \vdash E \Rightarrow T}$ E synthesizes type T

$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow S \to T \quad \Delta;\Gamma \vdash E_2 \Leftarrow S}{\Delta;\Gamma \vdash E_1\, E_2 \Rightarrow T}\ \texttt{t-app} \qquad \frac{\Delta;\Gamma \vdash E \Rightarrow \Pi^{*} X{:}U.\, T \quad * = \{i, e\} \quad \Delta \vdash C \Leftarrow U}{\Delta;\Gamma \vdash E\, [C] \Rightarrow [C/X]T}\ \texttt{t-app-index}$$

$$\frac{\Gamma(x) = T}{\Delta;\Gamma \vdash x \Rightarrow T}\ \texttt{t-var} \qquad \frac{\Delta;\Gamma \vdash E \Leftarrow T}{\Delta;\Gamma \vdash E{:}T \Rightarrow T}\ \texttt{t-ann}$$

$\boxed{\Delta; \Gamma \vdash E \Leftarrow T}$ E type checks against type T

$$\frac{\Delta;\Gamma \vdash E \Rightarrow T}{\Delta;\Gamma \vdash E \Leftarrow T}\ \texttt{t-syn} \qquad \frac{\Delta;\Gamma, x{:}T_1 \vdash E \Leftarrow T_2}{\Delta;\Gamma \vdash (\mathbf{fn}\, x{\Rightarrow}E) \Leftarrow T_1 \to T_2}\ \texttt{t-fn}$$

$$\frac{\Delta, X{:}U;\Gamma \vdash E \Leftarrow T \quad * = \{i, e\}}{\Delta;\Gamma \vdash (\lambda\, X{\Rightarrow}E) \Leftarrow \Pi^{*} X{:}U.\, T}\ \texttt{t-mlam} \qquad \frac{\Delta;\Gamma \vdash E \Rightarrow S \quad \Delta;\Gamma \vdash \vec{B} \Leftarrow S \to T}{\Delta;\Gamma \vdash \mathbf{case}\, E\, \mathbf{of}\, \vec{B} \Leftarrow T}\ \texttt{t-case}$$

$\boxed{\Delta; \Gamma \vdash \Pi\Delta'; \Gamma'. Pat{:}\theta \mapsto E \Leftarrow T}$ Branch $B = \Pi\Delta'; \Gamma'. Pat{:}\theta \mapsto E$ checks against type T

$$\frac{\Delta' \vdash \theta{:}\Delta \quad \Delta';\Gamma' \vdash Pat \Leftarrow [\theta]S \quad \Delta'; [\theta]\Gamma, \Gamma' \vdash E \Leftarrow [\theta]T}{\Delta;\Gamma \vdash \Pi\Delta'; \Gamma'. Pat{:}\theta \mapsto E \Leftarrow S \to T}\ \texttt{t-branch}$$

$\boxed{\Delta; \Gamma \vdash Pat \Leftarrow T}$ Pattern Pat checks against T

$$\frac{\Delta \vdash C \Leftarrow U}{\Delta;\Gamma \vdash [C] \Leftarrow [U]}\ \texttt{t-pindex} \qquad \frac{\Gamma(x) = T}{\Delta;\Gamma \vdash x \Leftarrow T}\ \texttt{t-pvar} \qquad \frac{\Sigma(\mathbf{c}) = T \quad \Delta;\Gamma \vdash \overrightarrow{Pat} \Leftarrow T \rangle S}{\Delta;\Gamma \vdash \mathbf{c}\, \overrightarrow{Pat} \Leftarrow S}\ \texttt{t-pcon}$$

$\boxed{\Delta; \Gamma \vdash \overrightarrow{Pat} \Leftarrow T \rangle S}$ Pattern spine $\overrightarrow{Pat}$ checks against T and has result type S

$$\frac{\Delta \vdash C \Leftarrow U \quad \Delta;\Gamma \vdash \overrightarrow{Pat} \Leftarrow [C/X]T \rangle S}{\Delta;\Gamma \vdash [C]\, \overrightarrow{Pat} \Leftarrow \Pi^{e} X{:}U.\, T \rangle S}\ \texttt{t-spi} \qquad \frac{\Delta;\Gamma \vdash Pat \Leftarrow T_1 \quad \Delta;\Gamma \vdash \overrightarrow{Pat} \Leftarrow T_2 \rangle S}{\Delta;\Gamma \vdash Pat\, \overrightarrow{Pat} \Leftarrow T_1 \to T_2 \rangle S}\ \texttt{t-sarr} \qquad \frac{}{\Delta;\Gamma \vdash \cdot \Leftarrow S \rangle S}\ \texttt{t-snil}$$

Figure 4. Typing of computational expressions

The introductions, functions $\mathbf{fn}\, x{\Rightarrow}e$ and dependent functions $\lambda\, x{\Rightarrow}e$, check against their respective type. Their corresponding eliminations, application $E_1\, E_2$ and dependent application $E\,[C]$, synthesize their type. We rely in this rule on the index-level substitution operation and we assume that it is defined in such a way that normal forms are preserved[2].

To type-check a case-expressions $\mathbf{case}\, E\, \mathbf{of}\, \vec{B}$ against T, we synthesize a type S for E and then check each branch against $S \to T$. A branch $\Pi\Delta'; \Gamma'. Pat{:}\theta \mapsto E$ checks against $S \to T$, if: 1) θ is a refinement substitution mapping all index variables declared in Δ to a new context Δ', 2) the pattern Pat is compatible with the type S of the scrutinee, i.e. Pat has type $[\theta]S$, and the body E checks against $[\theta]T$ in the index context Δ' and the program context $[\theta]\Gamma, \Gamma_i$. Note that the refinement substitution effectively performs a context shift.

We present the typing rules for patterns in spine format which will simplify our elaboration and inferring types for pattern variables. We start checking a pattern against a given type and check index objects and variables against the expected type. If we encounter $\mathbf{c}\, \overrightarrow{Pat}$ we look up the type T of the constant $\mathbf{c}$ in the signature and continue to check the spine $\overrightarrow{Pat}$ against T with the expected return type S. Pattern spine typing succeeds if all patterns in the spine $\overrightarrow{Pat}$ have the corresponding type in T and yields the return type S.

3.2 Elaborated examples

We show here the result of elaboration for the type-preserving evaluator given in Section 2.2.2.

```
rec eval: Πⁱ. T:tp.Tm [T]→Val [T] = λ T ⇒ fn e ⇒
case e of ...
| . ;  e1:Tm [nat], e2:Tm [nat] .
  Plus e1 e2 : nat/T ⇒
  case (eval [nat] e1 , eval [nat] e2) of
  |. ;x:Tm [nat],y:Tm [nat]. (VNum x, VNum y)  : .
      ⇒ VNum (add x y)

| T:tp ; e:Tm [bool], e1:Tm [T], e2:Tm [T].
 If [T] e e1 e2 : T/T⇒
```

[2] In Beluga, this is for example achieved by relying hereditary substitutions[Cave and Pientka 2012].

```
case eval [bool] e of
| T:tp ; .  VTrue : T/T ⇒ eval [T] e1
| T:tp ; .  VFalse : T/T ⇒ eval [T] e2
```

To elaborate a recursive declaration we start by reconstructing the type annotation. In this case the user left the variable `T` free which becomes an implicit argument and we abstract over this variable with Π^i`. T:Tp` marking it implicit. Next, elaborate the function body given the fully elaborated type. First, we therefore add the corresponding abstraction λ `T⇒ .`

Elaboration proceeds recursively on the term. We reconstruct the case-expression, considering the scrutinee `e` and we infer its type as `Tm [T]`. We elaborate the branches next. Recall that each branch in the source language consists of a pattern and a body. Moreover, the body can refer to any variable in the pattern or variables introduced in outer patterns. However, in the target language branches abstract over the context $\Delta; \Gamma$ and add a refinement substitution θ. The body of the branch, refers to variables declared in the branch contexts only. In each branch, we list explicitly the index variables and pattern variables. For example in the branch for `If` we added `T:tp` to the index context Δ of the branch and `e:Tm [bool], e1:Tm [T], e2:Tm [T]` to computational context Γ. The refinement substitution moves terms from the outer context to the branch context, refining the appropriate index variables as expressed by the pattern. For example in the `Plus` branch, the substitution refines the type `T` to `nat`.

As we mentioned before, elaboration added an implicit parameter to the type of function `eval`, and the user is not allowed to directly supply an instantiation for it. Implicit parameters have to be inferred by elaboration. In the recursive calls to `eval`, we add the parameter that represents the type of the term being evaluated.

The output of the elaboration process is a target language term that can be type checked with the rules from figure 4. If elaboration fails it can either be because the source level program describes a term that would be ill-typed when elaborated, or in some cases, elaboration fails because it cannot infer all the implicit parameters. In this case, the user can add type annotations to simplify the job of elaboration.

4. Description of elaboration

Elaboration of our source-language to our core target language is bi-directional and guided by the expected target type. Recall that we mark in the target type argument which are implicitly quantified (see $\Pi^i X{:}U.\,T$). This annotation is added when we elaborate a source type with free variables. If we check a source expression against $\Pi^i X{:}U.\,T$ we insert the appropriate λ-abstraction in our target. If we have synthesized the type $\Pi^i X{:}U.\,T$ for an expression, we insert *hole variables* for the omitted argument of type U. When we switch between synthesizing a type S for a given expression and checking an expression against an expected type T, we will rely on unification to make them equal. A key challenge is how to elaborate case-expressions where we pattern match on a dependently typed expression and we might pattern in the branches might refine it. Our elaboration is parametric in the index domain, hence we keep our definitions of holes, instantiation of holes and unification abstract and only state their definitions and properties.

4.1 Elaboration of index objects

To elaborate a source expression, we insert holes for omitted index arguments and elaborate index objects which occur in it. We hence make a few requirements about our index domain. We assume

1. A function genHole $(?Y : \Delta.U)$ that generates a term standing for a hole of type U in the context Δ, i.e. its instantiation may refer to the index variables in Δ. If the index language is first-order, then we can characterize holes for example by meta-variables [Nanevski et al. 2008]. If our index language is higher-order, for example if we choose contextual LF as in Beluga, we characterize holes using meta^2-variables as described in Boespflug and Pientka [2011].
2. A typing judgment for guaranteeing that index objects with holes are well-typed:

   $\Theta; \Delta \vdash C : U$ Index object C has index type U in context Δ and all holes in C are declared in Θ

3. Unification algorithm which finds the most general unifier for two index objects. In Beluga, we rely on higher-order pattern unification [Miller 1991; Dowek et al. 1996]. We characterize here abstractly the unification judgement for computation-level types, which in turn will rely on unifying index-level terms; technically, we in fact rely on two unification judgments: one finding instantiations for holes, the other finding most general instantiations for index variables such that two index terms become equal. We use the first one during elaboration, the second one is used to make two index objects equal as for example during matching or when computing the type refinement in branches.

   $$\begin{array}{rcll} \Theta; \Delta & \vdash & C_1 \doteq C_2 / \Theta'; \rho & \text{where: } \Theta' \vdash \rho{:}\Theta \\ \Delta & \vdash & C_1 \doteq C_2 / \Delta'; \theta & \text{where: } \Delta' \vdash \theta{:}\Delta \end{array}$$

   where ρ describes the instantiation for holes in Θ. If unification succeeds, then we have $[\![\rho]\!]C_1 = [\![\rho]\!]C_2$ and $[\theta]C_1 = [\theta]C_2$ respectively.

4. Elaboration of index objects themselves. If the index language is simply typed, the elaboration has nothing to do; however, if as in Beluga, our index objects are objects described in the logical framework LF, then we need to elaborate them and infer omitted arguments following [Pientka 2013]. There are two related forms of elaboration for index objects we use:

   $$\begin{array}{ll} \Theta; \Delta & \vdash c : U \rightsquigarrow C/\Theta'; \Delta'; \rho \\ \Theta; \Delta & \vdash \wr c \,;\, \theta \wr : U \rightsquigarrow C/\Theta'; \rho \end{array}$$

   The first judgment reconstructs the index object c by checking it against U. We thread through a context Θ of holes and a context of index variables Δ, we have seen so far. The object c however may contain additional free index variables whose type we infer during elaboration. All variables occurring in C will be eventually declared with their corresponding type in Δ'. As we elaborate c, we may refine holes and add additional holes. ρ describes the mapping between Θ and Θ', i.e. it records refinement of holes. Finally, we know that $\Delta' = [\![\rho]\!]\Delta, \Delta_0$, i.e. Δ' is an extension of Δ. We use the first judgment in elaborating patterns and type declarations in the signature.

   The second judgment is similar to the first, but does not allow free index variables in c. We elaborate c together with a refinement substitution θ, which records refinements obtained from earlier branches. When we encounter an index variable, we look up what it is mapped to in θ and return it. Given a hole context Θ and a index variable context Δ, we elaborate an index term c against a given type U. The result is two fold: a context Θ' of holes is related to the original hole context Θ via the hole instantiation ρ. We use the second judgment to elaborate index objects embedded into target expressions.

$$\frac{\cdot;\cdot \mid \cdot \vdash t \rightsquigarrow T/\Theta;\Delta;\cdot \qquad \Delta_i \vdash \epsilon : \Theta}{\vdash \mathbf{c} : t \rightsquigarrow \Pi^i(\Delta_i, [\![\epsilon]\!]\Delta).\, [\![\epsilon]\!]T}\ \texttt{el-typ} \qquad \frac{\cdot;\cdot \mid \cdot \vdash k \rightsquigarrow K/\Theta;\Delta;\cdot \qquad \Delta_i \vdash \epsilon : \Theta}{\vdash \mathbf{a} : k \rightsquigarrow \Pi^i(\Delta_i, [\![\epsilon]\!]\Delta).\, [\![\epsilon]\!]K}\ \texttt{el-kind}$$

$$\frac{\cdot;\cdot \mid \cdot \vdash t \rightsquigarrow T/\Theta;\Delta;\cdot \qquad \Delta_i \vdash \epsilon : \Theta \qquad \cdot; f{:}\Pi^i\Delta_i, [\![\epsilon]\!]\Delta.\, T \vdash \wr e\ ;\ \cdot \int : \Pi^i(\Delta_i, [\![\epsilon]\!]\Delta).\, [\![\epsilon]\!]T \rightsquigarrow E/\cdot;\cdot}{\vdash \mathbf{rec}\, f{:}t = e \rightsquigarrow \mathbf{rec}\, f{:}\Pi^i(\Delta_i, [\![\epsilon]\!]\Delta).\, [\![\epsilon]\!]T = E}\ \texttt{el-rec}$$

Figure 5. Elaborating declarations

4.2 Elaborating declarations

We begin our discussion of elaborating source programs in a top-down manner starting with declarations. Types and kinds in declarations may contain free variables and there are two different tasks: we need to fill in omitted arguments, infer the type of free variables and abstract over the free variables and holes which are left over in the elaborated type / kind. We rely here on the fact that the index language provides a way of inferring the type of free variables.

To abstract over holes in a given type T, we employ a lifting operation: $\Delta \vdash \epsilon : \Theta$ which mpas each hole to a fresh index variable.

$$\frac{}{\cdot \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash \epsilon : \Theta}{\Delta, X : U \ \ \vdash \epsilon,\ (\ .X)/X : \Theta, X : (\ .U)}$$

If holes do not have atomic type U this lifting fails. Removing this restriction would require us to be able to allow higher-order index variables, i.e. $X\ Y$, which we currently do not support[3].

To elaborate a constant declarations $\mathbf{c} : t$ we elaborate the type t to a target type T where free index variables are listed in Δ and the remaining holes in T are described in Θ. We then lift all the holes in Θ to proper declarations in Δ_i via the lifting substitution ϵ. The final elaborated type of the constant **c** is: $\Pi^i(\Delta_i, [\![\epsilon]\!]\Delta).\, [\![\epsilon]\!]T$. Note that both the free variables in the type t and the lifted holes described in Δ_i form the implicit arguments and are marked with Π^i. The elaboration of kinds follows the same principle.

To elaborate recursive function declarations, we first elaborate the type t abstracting over all the free variables and lifting the remaining holes to obtain $\Pi^i(\Delta_i, [\![\epsilon]\!]\Delta).\, [\![\epsilon]\!]T$. Second, we assume f of this type and elaborate the body e checking it against $\Pi^i(\Delta_i, [\![\epsilon]\!]\Delta).\, [\![\epsilon]\!]T$. We note that we always elaborate a source expression e together with a possible refinement substitution θ. In the beginning, θ will be empty. We describe elaboration of source expressions in the next section.

4.3 Elaborating source expressions

We elaborate source expressions bi-directionally. Expressions such as non-dependent functions and dependent functions are elaborated by checking the expression against a given type; expressions such as application and dependent application are elaborated to a corresponding target expression and at the same time synthesize the corresponding type.

Synthesizing: $\Theta;\Delta;\Gamma \vdash \wr e\ ;\theta \int \quad \rightsquigarrow E{:}T/\Theta';\rho$
Checking: $\Theta;\Delta;\Gamma \vdash \wr e\ ;\theta \int : T \rightsquigarrow E \ \ /\Theta';\rho$

We first explain the judgment for elaborating a source expression e by checking it against T given holes in Θ, index variables Δ, and program variables Γ. Because of pattern matching, index variables in Δ may get refined to concrete index terms. Abusing slightly notation, we write θ for the map of free variables occurring in e to their refinements and consider a source expression e together with the refinement map θ, written as $\wr e\ ;\ \theta \int$. The result of elaborating $\wr e\ ;\ \theta \int$ is a target expression E, a new context of holes Θ', and a hole instantiation ρ which instantiates holes in Θ, i.e. $\Theta' \vdash \rho : \Theta$. The result E has type $[\![\rho]\!]T$.

The result of elaboration in synthesis mode is similar; we return the target expression E together with its type T, a new context of holes Θ' and a hole instantiation ρ, s.t. $\Theta' \vdash \rho : \Theta$. The result is well-typed, i.e. E has type T.

We give the rules for elaborating source expressions in checking mode in Fig. 6 and in synthesis mode in Fig. 7. To elaborate a function (see rule `el-fn`) we simple elaborate the body extending the context Γ. There are two cases when we elaborate an expression of dependent function type. In the rule `el-mlam`, we elaborate a dependent function $\lambda\, X \Rightarrow e$ against $\Pi^e X{:}U.\, T$ by elaborating the body e extending the context $\Delta$with the declaration $X{:}U$. In the rule `el-mlam-i`, we elaborate an expression e against $\Pi^i X{:}U.\, T$ by elaborating e against T extending the context Δ with the declaration $X{:}U$. The result of elaborating e is then wrapped in a dependent function.

When switching to synthesis mode, we elaborate $\wr e\ ;\ \theta \int$ and obtain the corresponding target expression E and type T' together with an instantiation ρ for holes in Θ. We then unify the synthesized type T' and the expected type $[\![\rho]\!]T$ obtaining an instantiation ρ' and return the composition of the instantiation ρ and ρ'. When elaborating an index object $[c]$ (see rule `el-box`), we resort to elaborating c in our indexed language which we assume.

One of the key cases is the one for case-expressions. In the rule `el-case`, we elaborate the scrutinee synthesizing a type S; we then elaborate the branches. Note that we verify that S is a closed type, i.e. it is not allowed to refer to holes. To put it differently, the type of the scrutinee must be fully known. This is done to keep a type refinement in the branches from influencing the type of the scrutinee. For a similar reason, we enforce that the type T, the overall type of the case-expression, is closed; were we to allow holes in T, we would need to reconcile the different instantiations found in different branches.

When elaborating a constant, we look up its type T_c in the signature Σ and then insert holes for the arguments marked implicit in its type (see Fig. 7). Recall that all implicit arguments are quantified at the outside, i.e. $T_c = \Pi^i X_n{:}U_n.\, \ldots \Pi^e X_1{:}U_1.\, S$ where S does not contain any implicit dependent types Π^i. We generate for each implicit declaration $X_k{:}U_k$ a new hole which can depend on the currently available index variables Δ. When elaborating a variable, we look up its type in Γ and because the variable can correspond to a recursive function with implicit parameters we insert holes for the arguments marked as implicit as in the constant case.

Elaboration of applications in the synthesis mode threads through the hole context and its instantiation, but is otherwise straightforward. In each of the application rules, we elaborate the first argument of the application obtaining a new hole context Θ_1 together with a hole instantiation ρ_1. We then apply the hole instantiation ρ_1 to the context Δ and Γ and to the refinement substitution θ, before elaborating the second part.

[3] In our implementation of elaboration in Beluga, we did not find this restriction to matter in practice.

$\boxed{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \int : T \rightsquigarrow E/\Theta'; \rho}$ Elaborate source $\wr e \,;\, \theta \int$ to target expression E checking against type T

$$\frac{\Theta; \Delta \vdash \wr c \,;\, \theta \int : U \rightsquigarrow C/\Theta'; \rho}{\Theta; \Delta; \Gamma \vdash \wr [c] \,;\, \theta \int : [U] \rightsquigarrow [C]/\Theta'; \rho}\ \texttt{el-box} \qquad \frac{\Theta; \Delta; \Gamma, x{:}T_1 \vdash \wr e \,;\, \theta \int : T_2 \rightsquigarrow E/\Theta'; \rho}{\Theta; \Delta; \Gamma \vdash \wr \mathbf{fn}\, x \Rightarrow e \,;\, \theta \int : T_1 \rightarrow T_2 \rightsquigarrow \mathbf{fn}\, x \Rightarrow E/\Theta'; \rho}\ \texttt{el-fn}$$

$$\frac{\Theta; \Delta, X{:}U; \Gamma \vdash \wr e \,;\, \theta, X/X \int : T \rightsquigarrow E/\Theta'; \rho}{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \int : \Pi^i X{:}U.\, T \rightsquigarrow \lambda\, X \Rightarrow E/\Theta'; \rho}\ \texttt{el-mlam-i} \qquad \frac{\Theta; \Delta, X{:}U; \Gamma \vdash \wr e \,;\, \theta, X/X \int : T \rightsquigarrow E/\Theta'; \rho}{\Theta; \Delta; \Gamma \vdash \wr \lambda\, X \Rightarrow e \,;\, \theta \int : \Pi^e X{:}U.\, T \rightsquigarrow \lambda\, X \Rightarrow E/\Theta'; \rho}\ \texttt{el-mlam}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \int \rightsquigarrow E{:}S/\cdot; \rho \qquad [\![\rho]\!]\Delta; [\![\rho]\!]\Gamma \vdash \wr \vec{b} \,;\, [\![\rho]\!]\theta \int : S \rightarrow [\![\rho]\!]T \rightsquigarrow \vec{B}}{\Theta; \Delta; \Gamma \vdash \wr \mathbf{case}\, e\, \mathbf{of}\, \vec{b} \,;\, \theta \int : T \rightsquigarrow \mathbf{case}\, E\, \mathbf{of}\, \vec{B}/\cdot; \rho}\ \texttt{el-case}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \int \rightsquigarrow E{:}T_1/\Theta_1; \rho \quad \Theta_1; [\![\rho]\!]\Delta \vdash T_1 \doteq [\![\rho]\!]T/\Theta_2; \rho'}{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \int : T \rightsquigarrow [\![\rho']\!]E/\Theta_2; \rho' \circ \rho}\ \texttt{el-syn}$$

Figure 6. Elaboration of Expressions (Checking Mode)

$\boxed{\Theta; \Delta \vdash E : T \rightsquigarrow E' : T'/\Theta'}$ Apply E to holes for representing omitted arguments based on T obtaining a term E' of type T'

$$\frac{\mathsf{genHole}\ (?Y{:}\Delta.U) = C \quad (\Theta, ?Y{:}\Delta.U);\ \Delta \vdash E\,[C] : [C/X]T \rightsquigarrow E'{:}T' \,/\, \Theta'}{\Theta; \Delta \vdash E : \Pi^i X{:}U.\, T \rightsquigarrow E'{:}T' \,/\, \Theta'}\ \texttt{el-impl} \qquad \frac{S \neq \Pi^i X{:}U.\, T}{\Theta; \Delta \vdash E{:}S \rightsquigarrow E{:}S \,/\, \Theta}\ \texttt{el-impl-done}$$

$\boxed{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \int \rightsquigarrow E{:}T/\Theta'; \rho}$ Elaborate source $\wr e \,;\, \theta \int$ to target E and synthesize type T

$$\frac{\Gamma(x) = T \quad \Theta; \Delta; \Gamma \vdash x{:}T \rightsquigarrow E'{:}T'/\Theta'}{\Theta; \Delta; \Gamma \vdash \wr x \,;\, \theta \int \rightsquigarrow E'{:}T' \,/\, \Theta'; \mathtt{id}(\Theta')}\ \texttt{el-var} \qquad \frac{\Sigma(c) = T_c \quad \Theta; \Delta \vdash \mathbf{c} : T_c \rightsquigarrow E : T \,/\, \Theta'}{\Theta; \Delta; \Gamma \vdash \wr \mathbf{c} \,;\, \theta \int \rightsquigarrow E : T \,/\, \Theta'; \mathtt{id}(\Theta')}\ \texttt{el-const}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e_1 \,;\, \theta \int \rightsquigarrow E_1{:}S \rightarrow T \,/\, \Theta_1; \rho_1 \quad \Theta_1; [\![\rho_1]\!]\Delta; [\![\rho_1]\!]\Gamma \vdash \wr e_2 \,;\, [\![\rho_1]\!]\theta \int : [\![\rho]\!]_1 S \rightsquigarrow E_2 \,/\, \Theta_2; \rho_2}{\Theta; \Delta; \Gamma \vdash \wr e_1\, e_2 \,;\, \theta \int \rightsquigarrow E_1\, E_2 : [\![\rho_2]\!]T \,/\, \Theta_2; \rho_2 \circ \rho_1}\ \texttt{el-app}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \int \rightsquigarrow E_1{:}\Pi^e X{:}U.\, T/\Theta_1; \rho_1 \quad \Theta_1; [\![\rho_1]\!]\Delta \vdash \wr c \,;\, [\![\rho_1]\!]\theta \int : U \rightsquigarrow C/\Theta_2; \rho_2}{\Theta; \Delta; \Gamma \vdash \wr e\,[c] \,;\, \theta \int \rightsquigarrow E_1\,[C]{:}[C/X]([\![\rho_2]\!]T)/\Theta_2; \rho_2 \circ \rho_1}\ \texttt{el-mapp}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \int \rightsquigarrow E{:}\Pi^e X{:}U.\, T/\Theta_1; \rho \quad \mathsf{genHole}\ (?Y : ([\![\rho]\!]\Delta).U) = C}{\Theta; \Delta; \Gamma \vdash \wr e\, \_ \,;\, \theta \int \rightsquigarrow E\,[C] : [C/X]T \ \,/\, \ \Theta_1, ?Y{:}([\![\rho_1]\!]\Delta).U \,;\, \rho}\ \texttt{el-mapp-underscore}$$

$$\frac{\Theta; \Delta \vdash \wr t \,;\, \theta \int \rightsquigarrow T/\Theta_1; \rho_1 \qquad \Theta_1; [\![\rho]\!]\Delta; [\![\rho]\!]\Delta \vdash \wr e \,;\, [\![\rho]\!]\theta \int : T \rightsquigarrow E/\Theta_2; \rho_2}{\Theta; \Delta; \Gamma \vdash \wr e{:}t \,;\, \theta \int \rightsquigarrow (E{:}T){:}T/\Theta_2; \rho_2 \circ \rho_1}\ \texttt{el-annotated}$$

Figure 7. Elaborating of Expressions (Synthesizing Mode)

4.3.1 Elaborating branches

We give the rules for elaborating branches in Fig. 8. Recall that a branch $pat \mapsto e$ consists of the pattern pat and the body e. We elaborate a branch under the refinement θ, because the body e may contain index variables declared earlier and which might have been refined in earlier branches.

Intuitively, to elaborate a branch, we need to elaborate the pattern and synthesize the type of index and pattern variables bound inside of it. In the dependently typed setting, pattern elaboration needs to do however a bit more work: we need to infer implicit arguments which were omitted by the programmer and we need to establish how the synthesized type of the pattern refines the type of the scrutinee.

Moreover, there is a mismatch between the variables the body e may refer to (see rule `wf-branch` in Fig. 2) and the context the elaborated body E is meaningful (see rule `t-branch` in Fig. 4). While our source expression e possibly can refer to index variables declared prior, the elaborated body E is not allowed to refer to any index variables which were declared at the outside; those index variables are replaced by their corresponding refinements. To account for these additional refinements, we not only return an elaborated pattern $\Pi\Delta_r; \Gamma_r.Pat{:}\theta_r$ when elaborating a pattern pat (see rule `el-subst` in Fig. 8), but in addition return a map θ_e between source variables declared explicitly outside to their refinements.

Technically, elaborating a pattern is done in three steps.

1. First, given pat we elaborate it to a target pattern Pat together with its type S_p synthesizing the type of index variables Δ_p and the type of pattern variables Γ_p together with holes (Θ_p) which denote omitted arguments. This is accomplished by the first premise of the rule `el-subst`:

   $$\cdot; \cdot \vdash pat \rightsquigarrow \Pi\Delta_p; \Gamma_p.Pat : S_1/\Theta_p; \cdot$$

   Our pattern elaboration judgment threads through the hole context and the context of index variables, both of which are empty

$\boxed{\Delta;\Gamma \vdash \wr b \; ; \; \theta \int : S \to T \rightsquigarrow B}$ Elaborate source branch $\wr b \; ; \; \theta \int$ to target branch B

$$\frac{\Delta \vdash pat : S \rightsquigarrow \Pi\Delta_r;\Gamma_r.Pat : \theta_r \mid \theta_e \quad \cdot;\Delta_r;[\theta_r]\Gamma,\Gamma_r \vdash \wr e \; ; \; \theta_r \circ \theta,\ \theta_e \int : [\theta_r]T \rightsquigarrow E/\cdot;\cdot}{\Delta;\Gamma \vdash \wr pat \mapsto e \; ; \; \theta \int : S \to T \rightsquigarrow \Pi\Delta_r;\Gamma_r.Pat{:}\theta_r \mapsto E}\ \texttt{el-branch}$$

$\boxed{\Delta \vdash pat : T \rightsquigarrow \Pi\Delta_r;\Gamma_r.Pat : \theta_r \mid \theta_e}$

$$\frac{\cdot;\cdot \vdash pat \rightsquigarrow \Pi\Delta_p;\Gamma_p.Pat : S_p/\Theta_p;\cdot \qquad \Delta'_p \vdash \rho : \Theta_p \qquad \Delta,(\Delta'_p,[\![\rho]\!]\Delta_p) \vdash [\![\rho]\!]S_p \doteqdot S/\Delta_r;\theta_R}{\Delta \vdash pat : S \rightsquigarrow \Pi\Delta_r;[\theta_p][\![\rho]\!]\Gamma_p.[\theta_p][\![\rho]\!]Pat : \theta_r \mid \theta_e}\ \texttt{el-subst}$$

where $\theta_R = \theta_r,\theta_p$ s.t. $\Delta_r \vdash \theta_p : (\Delta'_p,[\![\rho]\!]\Delta_p)$ and $\theta_p = \theta_i,\theta_e$ s.t. $\Delta_r \vdash \theta_i : \Delta'_p$

Figure 8. Branches and patterns

in the beginning. Because program variables occur linearly, we do not thread them through but simply combine program variable contexts when needed. The result of elaborating pat is a pattern Pat in our target language where Δ_p describes all index variables in Pat, Γ_p contains all program variables and Θ_p contains all holes, i.e. most general instantiations of omitted arguments. We describe pattern elaboration in detail in Section 4.3.2.

2. Second, we abstract over the hole variables in Θ_p by lifting all holes to fresh index variables from Δ'_p. This is accomplished by the second premise of the rule `el-subst`.
3. Finally, we compute the refinement substitution θ_r which ensures that the type of the pattern $[\![\rho]\!]S_p$ is compatible with the type S of the scrutinee. We note that the type of the scrutinee could also force a refinement of holes in the pattern. This is accomplished by the judgment:

$$\Delta,(\Delta'_p,[\![\rho]\!]\Delta_p) \vdash [\![\rho]\!]S_1 \doteqdot T_1/\Delta_r;\theta_R \qquad \theta_R = \theta_r,\theta_p$$

We note because θ_R maps index variables from $\Delta,(\Delta'_p,[\![\rho]\!]\Delta_p)$ to Δ_r, it contains two parts: θ_r provides refinements for variables Δ in the type of the scrutinee; θ_p provides possible refinements of the pattern forced by the scrutinee. This can happen, if the scrutinee's type is more specific than the type of the pattern.

4.3.2 Elaborating patterns

Pattern elaboration is bi-directional. The judgements for elaborating patterns by checking them against a given type and synthesizing their type are:

$$\begin{array}{ll} \text{Synthesizing:} & \Theta;\Delta\vdash pat \quad\ \rightsquigarrow \Pi\Delta';\Gamma.Pat{:}T \ / \ \Theta';\rho \\ \text{Checking:} & \Theta;\Delta\vdash pat : T \rightsquigarrow \Pi\Delta';\Gamma.Pat \quad\ / \ \Theta';\rho \end{array}$$

As mentioned earlier, we thread through a hole context Θ together with the hole substitution ρ that relates: $\Theta' \vdash \rho{:}\Theta$. Recall that as our examples show index-level variables in patterns need not to be linear and hence we accumulate index variables and thread them through as well. Program variables on the other hand must occur linearly, and we can simply combine them. In synthesis mode, elaboration returns a reconstructed pattern Pat, a type T where Δ' describes the index variables in Pat and Γ' contains all program variables occurring in Pat. The hole context Θ' describes the most general instantiations for omitted arguments which have been inserted into Pat. In checking mode, we elaborate pat given a type T to the target expression Pat and index variable context Δ', pattern variable context Γ' and the hole context Θ'.

Pattern elaboration starts in synthesis mode, i.e. either elaborating an annotated pattern $(e : t)$ (see rule `el-pann`) or a pattern $\mathbf{c}\ \overrightarrow{pat}$ (see rule `el-pcon`). To reconstruct patterns that start with a constructor we first look-up the constructor in the signature Σ to get its fully elaborated type T_c and then elaborate the arguments $\overrightarrow{pat}$ against T_c. Elaborating the spine of arguments is guided by the type T_c. If $T_c = \Pi^i X{:}U.T$, then we generate a new hole for the omitted argument of type U. If $T_c = T_1 \to T_2$, then we elaborate the first argument in the spine $pat\ \overrightarrow{pat}$ against T_1 and the remaining arguments $\overrightarrow{pat}$ against T_2. If $T_c = \Pi^e X{:}U.T$, then we elaborate the first argument in the spine $[c]\ \overrightarrow{pat}$ against U and the remaining arguments $\overrightarrow{pat}$ against $[C/X]T$. When the spine is empty, denoted by $\cdot$, we simply return the final type and check that constructor was fully applied by ensuring that the type S we reconstruct against is either of index level type, i.e. $[U]$, or a recursive type, i.e. $\mathbf{a}\overrightarrow{[C]}$.

For synthesizing the patterns with a type annotation, first we elaborate the type t in an empty context using a judgement that returns the reconstructed type T, its holes and index variables (contexts Θ' and Δ'). Once we have the type we elaborate the pattern checking against the type T.

To be able to synthesize the type of pattern variables and return it, we check variables against a given type T during elaboration (see rule `el-pvar`). For index level objects, rule `el-pindex`we defer to the index level elaboration that the index domain provides[4]. Finally, when elaborating a pattern against a given type it is possible to switch to synthesis mode using rule `el-psyn`, where first we elaborate the pattern synthesizing its type S and then we make sure that S unifies against the type T it should check against.

5. Soundness of elaboration

We establish soundness of our elaboration: if we start with a well-formed source expression, we obtain a well-typed target expression E which may still contain some holes and E is well-typed for any ground instantiation of these holes. In fact, our final result of elaborating a recursive function and branches must always return a closed expression.

Theorem 1 (Soundness)**.**

1. If $\Theta;\Delta;\Gamma \vdash \wr e \; ; \; \theta \int : T \rightsquigarrow E/\Theta';\rho$
then for any grounding hole instantiation ρ' *s.t.* $\cdot \vdash \rho' : \Theta'$ *and* $\rho_0 = \rho' \circ \rho$*, we have* $[\![\rho_0]\!]\Delta;[\![\rho_0]\!]\Gamma \vdash [\![\rho']\!]E \Leftarrow [\![\rho_0]\!]T$*.*

[4] Both, elaboration of pattern variables and of index objects can be generalized by for example generating a type skeleton in the rule `el-subst`given the scrutinee's type. This is in fact what is done in the implementation of Beluga.

Pattern (synthesis mode) $\boxed{\Theta;\Delta \vdash pat \rightsquigarrow \Pi\Delta';\Gamma.Pat{:}T \;/\; \Theta' \;;\; \rho}$

$$\frac{\Sigma(c) = T \quad \Theta;\Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \Pi\Delta';\Gamma.\overrightarrow{Pat} \rangle\ S \;/\; \Theta';\rho}{\Theta;\Delta \vdash \mathbf{c}\,\overrightarrow{pat} \rightsquigarrow \Pi\Delta';\Gamma.\mathbf{c}\,\overrightarrow{Pat}{:}S \;/\; \Theta';\rho}\ \texttt{el-pcon}$$

$$\frac{\cdot;\cdot \vdash \wr t\;;\;\cdot \smallint \rightsquigarrow T/\Theta';\Delta';\cdot \quad (\Theta,\Theta')\;;\;(\Delta,\Delta') \vdash pat : T \rightsquigarrow \Pi\Delta'';\Gamma.Pat \;/\; \Theta'';\rho'}{\Theta;\Delta \vdash (pat{:}t) \rightsquigarrow \Pi\Delta'';\Gamma.Pat{:}\llbracket\rho'\rrbracket T \;/\; \Theta'';\rho'}\ \texttt{el-pann}$$

Pattern (checking mode) $\boxed{\Theta;\Delta \vdash pat : T \rightsquigarrow \Pi\Delta';\Gamma.Pat \;/\; \Theta';\rho}$

$$\frac{}{\Theta;\Delta \vdash x : T \rightsquigarrow \Pi\Delta\;;\; x{:}T.x \;/\; \Theta;\texttt{id}(\Theta)}\ \texttt{el-pvar} \qquad \frac{\Theta;\Delta \vdash c : U \rightsquigarrow C/\Theta';\Delta';\rho}{\Theta;\Delta \vdash [c] : [U] \rightsquigarrow \Pi\Delta';\cdot\,.\,[C]/\Theta'\;;\;\rho}\ \texttt{el-pindex}$$

$$\frac{\Theta;\Delta \vdash pat \rightsquigarrow \Pi\Delta';\Gamma.Pat{:}S \;/\; \Theta';\rho \qquad \Theta';\Delta' \vdash S \doteq \llbracket\rho\rrbracket T \;/\; \rho';\Theta''}{\Theta;\Delta \vdash pat : T \rightsquigarrow \Pi\llbracket\rho'\rrbracket\Delta';\llbracket\rho'\rrbracket\Gamma\,.\,\llbracket\rho\rrbracket Pat \;/\; \Theta''\;;\;\rho'\circ\rho}\ \texttt{el-psyn}$$

Pattern Spines $\boxed{\Theta;\Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \Pi\Delta';\Gamma.\overrightarrow{Pat} \rangle\ S \;/\; \Theta';\rho}$

$$\frac{\text{either } T = [U] \text{ or } T = \mathbf{a}\ \overrightarrow{[C]}}{\Theta;\Delta \vdash \cdot : T \rightsquigarrow \Pi\Delta;\cdot\,.\,\cdot \rangle\ T \;/\; \Theta;\texttt{id}(\Theta)}\ \texttt{el-sp-empty}$$

$$\frac{\Theta;\Delta \vdash pat : T_1 \rightsquigarrow \Pi\Delta';\Gamma.Pat/\Theta';\rho \qquad \Theta';\Delta' \vdash \overrightarrow{pat} : \llbracket\rho\rrbracket T_2 \rightsquigarrow \Pi\Delta'';\Gamma'.\overrightarrow{Pat} \rangle\ S \;/\; \Theta'';\rho'}{\Theta;\Delta \vdash pat\ \ \overrightarrow{pat} : T_1 \to T_2 \rightsquigarrow \Pi\Delta'';(\Gamma,\Gamma')\,.\,(\llbracket\rho'\rrbracket Pat)\ \ \overrightarrow{Pat} \rangle\ S \;/\; \Theta'';\rho'\circ\rho}\ \texttt{el-sp-cmp}$$

$$\frac{\Theta;\Delta \vdash c : U \rightsquigarrow C/\Theta';\Delta';\rho \quad \Theta';\Delta' \vdash \overrightarrow{pat} : [C/X]\llbracket\rho\rrbracket T \rightsquigarrow \Pi\Delta'';\Gamma.\overrightarrow{Pat} \rangle\ S \;/\; \Theta'';\rho'}{\Theta;\Delta \vdash [c]\,\overrightarrow{pat} : \Pi^e X{:}U.\,T \rightsquigarrow \Pi\Delta'';\Gamma\,.\,(\llbracket\rho'\rrbracket[C])\,\overrightarrow{Pat} \rangle\ S \;/\; \Theta'';\rho'\circ\rho}\ \texttt{el-sp-explicit}$$

$$\frac{\mathsf{genHole}\ (?Y{:}\Delta.U) = C \qquad \Theta, ?Y{:}\Delta.U;\Delta \vdash \overrightarrow{pat} : [C/X]T \rightsquigarrow \Pi\Delta';\Gamma.\overrightarrow{Pat} \rangle\ S \;/\; \Theta';\rho}{\Theta;\Delta \vdash \overrightarrow{pat} : \Pi^i X{:}U.\,T \rightsquigarrow \Pi\Delta';\Gamma.(\llbracket\rho\rrbracket C)\ \ \overrightarrow{Pat} \rangle\ S \;/\; \Theta';\rho}\ \texttt{el-sp-implicit}$$

Figure 9. Elaboration of patterns and pattern spines

2. *If* $\Theta;\Delta;\Gamma \vdash \wr e\;;\;\theta \smallint \rightsquigarrow E{:}T/\Theta_1;\rho$
   then for any grounding hole instantiation ρ' *s.t.* $\cdot \vdash \rho' : \Theta_2$ *and* $\rho_0 = \rho' \circ \rho$, *we have* $\llbracket\rho_0\rrbracket\Delta;\llbracket\rho_0\rrbracket\Gamma \vdash \llbracket\rho'\rrbracket E' \Rightarrow \llbracket\rho'\rrbracket T'$.

3. *If* $\Delta;\Gamma \vdash \wr pat \mapsto e\;;\;\theta \smallint : S \to T \rightsquigarrow \Pi\Delta';\Gamma'.Pat : \theta' \mapsto E$
   then $\Delta;\Gamma \vdash \Pi\Delta';\Gamma'.Pat : \theta' \mapsto E \Leftarrow S \to T$.

To establish soundness of elaboration of case-expressions and branches, we rely on pattern elaboration which abstracts over the variables in patterns as well as over the holes which derive from most general instantiations inferred for omitted arguments. We abstract over these holes using a lifting substitution ϵ. In practice, we need a slightly more general lemma than the one stated below which takes into account the possibility that holes in Pat are further refined (see Appendix).

Lemma 2 (Pattern elaboration)**.**

1. *If* $\Theta;\Delta \vdash pat \rightsquigarrow \Pi\Delta_1;\Gamma_1.Pat{:}T/\Theta_1;\rho_1$ *and*
   ϵ *is a ground lifting substitution, such as* $\Delta_i \vdash \epsilon{:}\Theta_1$
   then $\Delta_i, \llbracket\epsilon\rrbracket\Delta_1;\llbracket\epsilon\rrbracket\Gamma_1 \vdash \llbracket\epsilon\rrbracket Pat \Leftarrow \llbracket\epsilon\rrbracket T$.

2. *If* $\Theta;\Delta \vdash pat : T \rightsquigarrow \Pi\Delta_1;\Gamma_1.Pat/\Theta_1;\rho_1$ *and*
   ϵ *is a ground lifting substitution, such as* $\Delta_i \vdash \epsilon{:}\Theta_1$
   then $\Delta_i, \llbracket\epsilon\rrbracket\Delta_1;\llbracket\epsilon\rrbracket\Gamma_1 \vdash \llbracket\epsilon\rrbracket Pat \Leftarrow \llbracket\epsilon\rrbracket\llbracket\rho_1\rrbracket T$.

3. *If* $\Theta;\Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \Pi\Delta_1;\Gamma_1.\overrightarrow{Pat} \rangle\ S/\Theta_1;\rho_1$ *and*
   ϵ *is a ground lifting substitution, such as* $\Delta_i \vdash \epsilon{:}\Theta_1$
   then $\Delta_i, \llbracket\epsilon\rrbracket\Delta_1;\llbracket\epsilon\rrbracket\Gamma_1 \vdash \llbracket\epsilon\rrbracket\overrightarrow{Pat} \Leftarrow \llbracket\epsilon\rrbracket\llbracket\rho_1\rrbracket T \rangle \llbracket\epsilon\rrbracket S$.

6. Related work

Our language contains indexed families of types that are related to Zenger's work [Zenger 1997] and the Dependent ML (DML) [Xi 2007] and Applied Type System (ATS) [Xi 2004; Chen and Xi 2005]. The objective in these systems is: a program that is typable in the extended indexed type system is already typable in ML. By essentially erasing all the type annotations necessary for verifying the given program is dependently typed, we obtain a simply typed ML-like program. In contrast, our language supports pattern matching on index objects. Our elaboration, in contrast to the one given in Xi [2007], inserts omitted arguments producing programs in a fully explicit dependently typed core language. This is different from DML-like systems which treat *all* index arguments as implicit and do not provide a way for programmers to manipulate and pattern match directly on index objects. Allowing users to explicitly access and match on index arguments changes the game substantially.

Elaboration from implicit to explicit syntax for dependently typed systems has first been mentioned by Pollack [1990] although no concrete algorithm to reconstruct omitted arguments was given. Luther [Luther 2001] refined these ideas as part of the TYPELab project. He describes an elaboration and reconstruction for the calculus of constructions without treating recursive functions and pattern matching. There is in fact little work on elaborating dependently-typed source language supporting recursion and pattern matching. For example, Agda's the bi-directional type inference algorithm described in [Norell 2007] concentrates on a core dependently typed calculus enriched with dependent pairs, but omits the rules for its extension with recursion and pattern matching. Idris, a dependently typed language developed by Brady [2013] uses a different technique. Idris starts by adding holes for all the implicit variables and it tries to instantiate these holes using unification. However, the language uses internally a tactic based elaborator that is exposed to the user who can interactively fill the holes using tactics. He does not prove soundness of the elaboration, but conjectures that given a type correct program its elaboration followed by a reverse elaboration produces a matching source level program.

A notable example, is the work by [Asperti et al. 2012] on describing a bi-directional elaboration algorithm for the Calculus of (Co)Inductive Constructions (CCIC) implemented in Matita. Their setting is very different from ours: CCIC is more powerful than our language since the language of recursive programs can occur in types and there is no distinction between the index language and the programming language itself. Moreover in Matita, we are only allowed to write total programs and all types must be positive. For these reasons their source and target language is more verbose than ours and refinement, i.e. the translation of the source to the target, is much more complex than our elaboation. The difference between our language and Matita particularly comes to light when writing case-expressions. In Matita as in Coq, the programmer needs to supply an invariant for the scrutinee and the overall type of the case expression as a type annotion. Each branch then is checked against the type given in the invariant. In contrast, our case-expressions require no type annotations and we refine each branch according to refinement imposed by the pattern in each branch. This makes our source and target language more light-weight and closer to a standard simply typed functional language.

Finally, refinement in Matita may leave some holes in the final program which then can be refined further by the user using for example tactics. We support no such interaction; in fact, we fail, if holes are left-over and the programmer is asked to provide more information.

Agda, Idris, Matita and Coq require users to abstract over all variables occurring in a type and the user statically labells arguments the user can freely omit. To ease the requirement of declaring all variables occurring in type, many of these systems such as Agda supports simply listing the variables occurring in a declaration without the type. This however can be brittle since it requires that the user chose the right order. Moreover, the user has the possibility to locally override the implicit arguments mechanism and provide instantiations for implicit arguments explicitly. This is in contrast to our approach where we guide elaboration using type annotations and omit arguments based on the free variables occurring in the declared type.

7. Conclusion and future work

In this paper we describe a surface language for writing dependently typed programs where we separate the language of types and index objects from the language of programs. Our programming language supports indexed data-types, dependent pattern matching and recursion. Programmers can leave index variables free when declaring the type of a constructor or recursive program as a way of stating that arguments for these free variables should be inferred by the type-directed elaboration. This offers a lightweight mechanism for writing compact programs which resemble their ML counterparts and information pertaining index arguments can be omitted. In particular, our handling of case-expressions does not require programmers to specify the type invariants the patterns and their bodies must satisfy. This is achieved by computing refinement substitutions. Moreover, we support nested pattern matching inside function (as opposed to languages such as Agda or Idris that only do pattern matching at the level of function declarations).

To guide elaboration and type inference, we allow type annotations which indirectly refine the type of sub-expressions; type annotations in patterns are also convenient to name index variables which do not occur explicitly in a pattern.

We prove our elaboration sound, in the sense that if elaboration produces a fully explicit term, this term will be well-typed. Finally, our elaboration is implemented in Beluga, where we use as the index domain contextual LF, and has been shown practical (see for example the implementation of a type-preserving compiler [Belanger et al. 2013]). We believe our work sheds some light into how to design and implement a dependently typed language where we have a separate index language.

In the future work, we would like to explore an appropriate notion of completeness of elaboration. This would provide stronger guarantees for programmers stating that all terms in the target language can be written as terms in the source language such that elaboration succeeds. Moreover, we would like to explore more powerful type-systems for the computational language, such as polymorphism.

References

A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8:1–49, 2012.

O. S. Belanger, S. Monnier, and B. Pientka. Programming type-safe transformations using higher-order abstract syntax. In G. Gonthier and M. Norrish, editors, *Third International Conference on Certified Programs and Proofs (CPP'13)*, Lecture Notes in Computer Science (LCNS 8307), pages 243–258. Springer, 2013.

M. Boespflug and B. Pientka. Multi-level contextual modal type theory. In G. Nadathur and H. Geuvers, editors, *6th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'11)*, Electronic Proceedings in Theoretical Computer Science (EPTCS), 2011.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

A. Cave and B. Pientka. Programming with binders and indexed datatypes. In *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM Press, 2012.

C. Chen and H. Xi. Combining programming with theorem proving. In O. Danvy and B. C. Pierce, editors, *10th International Conference on Functional Programming*, pages 66–77, 2005.

G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Joint International Conference and Symposium on Logic Programming*, pages 259–273. MIT Press, Sept. 1996.

J. Dunfield and B. Pientka. 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, June 2009.

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

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

C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

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

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

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007. Technical Report 33D.

F. Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

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

B. Pientka. 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, 2008.

B. Pientka. An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. *Journal of Functional Programming*, 1(1–37), 2013.

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

B. Pientka, S. S. Ruan, and A. Abel. Structural recursion over contextual objects. Technical report, School of Computer Science, McGill, January 2014.

R. Pollack. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes, 1990.

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

H. Xi. Dependent ml an approach to practical programming with dependent types. *Journal of Functional Programming*, 17:215–286, 3 2007.

H. Xi and F. Pfenning. Dependent types in practical programming. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, 1999.

C. Zenger. Indexed types. *Theoretical Computer Science*, 187(1-2):147–165, 1997.

A. Index language

We summarize here our requirements on the index domain. We denote index terms in the source language with c and index types in the source language as u.

Well-formedness of index objects (source) First, we define requirements on well-formedeness.

$\delta \vdash c \text{ wf}$	Index object c is well formed and closed with respect to δ
$\delta \vdash_f c \text{ wf}$	Index object c is well formed with respect to δ and may contain free index variables

Well-typed index objects (target)

$\Delta \vdash C : U$ Index object C has index type U in context Δ

Substitution C/X in an index object C' is defined as $[C/X]C'$.

Well-typed index objects with holes

$\Theta; \Delta \vdash C : U$ Index object C has index type U in context Δ and all holes in C are well-typed wrt Θ

$$\begin{array}{llll} \text{Hole types} & & ::= & \Delta.U \\ \text{Hole Contexts} & \Theta & ::= & \cdot \mid \Theta, ?X{:}\Delta.U \\ \text{Hole Inst.} & \rho & ::= & \cdot \mid \rho, \Delta.C/?X \end{array}$$

When we insert hole variables for omitted arguments in a given context Δ, we rely on the abstract function $\mathsf{genHole}\ (?Y : \Delta.U)$ which returns an index term containing a new hole variable.

$$\mathsf{genHole}\ (?Y : \Delta.U) \quad = \quad C \qquad \text{where } C \text{ describes a hole.}$$

Unification of index objects The notion of unification that elaboration needs depends on the index level language. As we mentioned, we require that equality on our index domain is decidable; for elaboration, we also require that there is a decidable unification algorithm which makes two terms equal. In fact, we need two forms: one which allows us to infer instantiations for holes and another which unifies two index objects finding most general instantiations for index variables such that the two objects become equal. We use the first one during elaboration, the second one is used to make two index objects equal as for example during matching.

$$\begin{array}{rcll} \Theta; \Delta & \vdash & C_1 \doteq C_2 / \Theta'; \rho & \text{where: } \Theta' \vdash \rho{:}\Theta \\ \Delta & \vdash & C_1 \doteq C_2 / \Delta'; \theta & \text{where: } \Delta' \vdash \theta{:}\Delta \end{array}$$

where ρ describes the instantiation for holes in Θ. If unification succeeds, then we have $[\![\rho]\!]C_1 = [\![\rho]\!]C_2$ and $[\theta]C_1 = [\theta]C_2$ respectively.

Elaboration of index objects Elaboration of index objects themselves. If the index language is simply typed, the elaboration has nothing to do; however, if as in Beluga, our index objects are objects described in the logical framework LF, then we need to elaborate them and infer omitted arguments following [Pientka 2013]. There are two related forms of elaboration for index objects we use:

$$\begin{array}{lll} \Theta; \Delta & \vdash c : U & \rightsquigarrow C/\Theta'; \Delta'; \rho \\ \Theta; \Delta & \vdash \wr c \,;\, \theta \wr : U & \rightsquigarrow C/\Theta'; \rho \end{array}$$

The first judgment elaborates the index object c by checking it against U. We thread through a context Θ of holes and a context of index variables Δ, we have seen so far. The object c however may contain additional free index variables whose type we infer during elaboration. All variables occurring in C will be eventually declared with their corresponding type in Δ'. As we elaborate c, we may refine holes and add additional holes. ρ describes the mapping between Θ and Θ', i.e. it records refinement of holes. Finally, we know that $\Delta' = [\![\rho]\!]\Delta, \Delta_0$, i.e. Δ' is an extension of Δ. We use the first judgment in elaborating patterns and type declarations in the signature.

The second judgment is similar to the first, but does not allow free index variables in c. We elaborate c together with a refinement substitution θ, which records refinements obtained from earlier branches. When we encounter an index variable, we look up what it is mapped to in θ and return it. Given a hole context Θ and a index variable context Δ, we elaborate an index term c against a given type U. The result is two fold: a context Θ' of holes is related to the original hole context Θ via the hole instantiation ρ. We use the second judgment to elaborate index objects embedded into target expressions.

B. Elaborating kinds and types in declarations

Recall that programmers may leave index variables free in type and kind decarations. Elaboration must infer the type of the free index variables in addition to reconstructing omitted arguments.

$\boxed{\Theta; \Delta_f \mid \Delta \vdash k \rightsquigarrow K / \Theta'; \Delta'_f; \rho'}$ Elaborate kind k to target kind K

$$\frac{}{\Theta; \Delta_f \mid \Delta \vdash \texttt{ctype} \rightsquigarrow \texttt{ctype} / \Theta; \Delta_f; \texttt{id}(\Theta)}\ \texttt{el-k-ctype}$$

$$\frac{\Theta; \Delta_f \mid \Delta \vdash u \rightsquigarrow U / \Theta'; \Delta'_f; \rho' \quad \Theta'; \Delta'_f \mid \Delta, X{:}U \vdash k \rightsquigarrow K / \Theta''; \Delta''_f; \rho''}{\Theta; \Delta_f \mid \Delta \vdash \{X{:}u\}\, k \rightsquigarrow \Pi^e X{:}(\llbracket \rho' \rrbracket U).\, K / \Theta''; \Delta''_f; \rho'' \circ \rho'}\ \texttt{el-k-pi}$$

$\boxed{\Theta; \Delta_f \mid \Delta \vdash t \rightsquigarrow T / \Theta'; \Delta'_f; \rho'}$ Elaborate type t to target type T

$$\frac{\Theta; \Delta_f \mid \Delta \vdash t_1 \rightsquigarrow T_1 / \Theta'; \Delta'_f; \rho' \quad \Theta'; \Delta'_f \mid \Delta \vdash t_2 \rightsquigarrow T_2 / \Theta''; \Delta''_f; \rho''}{\Theta; \Delta_f \mid \Delta \vdash t_1 \to t_2 \rightsquigarrow (\llbracket \rho'' \rrbracket T_1) \to T_2 / \Theta''; \Delta''_f; \rho'' \circ \rho'}\ \texttt{el-t-arr} \qquad \frac{\Theta; \Delta_f \mid \Delta \vdash u \rightsquigarrow U / \Theta'; \Delta'_f; \rho'}{\Theta; \Delta_f \mid \Delta \vdash [u] \rightsquigarrow [U] / \Theta'; \Delta'_f; \rho'}\ \texttt{el-t-idx}$$

$$\frac{\Theta; \Delta_f \mid \Delta \vdash u \rightsquigarrow U / \Theta'; \Delta'_f; \rho' \quad \Theta'; \Delta'_f \mid \Delta, X{:}U \vdash t \rightsquigarrow T / \Theta''; \Delta''_f; \rho''}{\Theta; \Delta_f \mid \Delta \vdash \{X{:}u\}\, t \rightsquigarrow \Pi^e X{:}(\llbracket \rho' \rrbracket U).\, T / \Theta''; \Delta''_f; \rho'' \circ \rho'}\ \texttt{el-t-pi}$$

$$\frac{\Sigma(\mathbf{a}) = K \quad \Theta; \Delta_f \mid \Delta \vdash \overrightarrow{[c]} : K \rightsquigarrow \overrightarrow{[C]} / \Theta'; \Delta'_f; \rho'}{\Theta; \Delta_f \mid \Delta \vdash \mathbf{a}\, \overrightarrow{[c]} \rightsquigarrow \mathbf{a}\, \overrightarrow{C} / \Theta'; \Delta'_f; \rho'}\ \texttt{el-t-con}$$

$\boxed{\Theta; \Delta_f \mid \Delta \vdash \overrightarrow{[c]} : K \rightsquigarrow \overrightarrow{[C]} / \Theta'; \Delta'_f; \rho'}$ Elaborate fully applied spine $\overrightarrow{[c]}$ checking against kind K to target spine $\overrightarrow{[C]}$

$$\frac{\Theta; \Delta_f \mid \Delta \vdash c : U \rightsquigarrow C / \Theta'; \Delta'_f; \rho' \quad \Theta'; \Delta'_f \mid \Delta \vdash \overrightarrow{[c]} : [C/X]K \rightsquigarrow \overrightarrow{[C]} / \Theta''; \Delta''_f; \rho''}{\Theta; \Delta_f \mid \Delta \vdash [c]\, \overrightarrow{[c]} : \Pi^e X{:}U.\, K \rightsquigarrow (\llbracket \rho' \rrbracket [C])\, \overrightarrow{[C]} / \Theta''; \Delta''_f; \rho'' \circ \rho'}\ \texttt{el-t-sp-explicit}$$

$$\frac{\mathsf{genHole}\ (?Y{:}(\Delta_f, \Delta).U) = C \quad \Theta; \Delta_f \mid \Delta \vdash \overrightarrow{c} : [C/X]K \rightsquigarrow \overrightarrow{C} / \Theta'; \Delta'_f; \rho'}{\Theta; \Delta_f \mid \Delta \vdash \overrightarrow{c} : \Pi^i X{:}U.\, K \rightsquigarrow (\llbracket \rho' \rrbracket C)\, \overrightarrow{C} / \Theta'; \Delta'_f; \rho'}\ \texttt{el-t-sp-implicit}$$

$$\frac{}{\Theta; \Delta_f \mid \Delta \vdash \cdot : \texttt{ctype} \rightsquigarrow \cdot / \Theta; \Delta_f; \texttt{id}(\Theta)}\ \texttt{el-t-sp-empty}$$

Figure 10. Elaborating kinds and types in declarations

We require that the index language provides us with the following judgments:

$$\Theta; \Delta_f \mid \Delta \vdash u \rightsquigarrow U / \Theta'; \Delta'_f; \rho'$$
$$\Theta; \Delta \vdash \wr u \,;\, \theta \wr \rightsquigarrow U / \Theta'; \rho'$$

Hence, we assume that the index language knows how to infer the type of free variables, for example. In Beluga where the index language is LF, we fall back to the ideas described in [Pientka 2013].

The first judgment collects free variables in Δ_f that later in elaboration will become implicit parameters. The context Δ_f is threaded through in addition to the hole context Θ.

The judgments for elaborating computation-level kinds and types are similar:

1. $\Theta; \Delta_f \mid \Delta \vdash k \rightsquigarrow K / \Theta'; \Delta'_f; \rho'$
2. $\Theta; \Delta_f \mid \Delta \vdash t \rightsquigarrow T / \Theta'; \Delta'_f; \rho'$
3. $\Theta; \Delta_f \mid \Delta \vdash \overrightarrow{[c]} : K \rightsquigarrow \overrightarrow{[C]} / \Theta'; \Delta'_f; \rho'$

We again collect free index variables in Δ_f which are threaded through together with the holes context Θ (see Figure 5 and Figure 10).

C. Soundness proof

Theorem 3 (Soundness)**.**

1. *If* $\Theta; \Delta; \Gamma \vdash \wr e \ ; \ \theta \wr : T \rightsquigarrow E/\Theta_1; \rho_1$ *then for any grounding hole instantiation* ρ_g *s.t.* $\cdot \vdash \rho_g : \Theta_1$ *and* $\rho_0 = \rho_g \circ \rho_1$*, we have* $[\![\rho_0]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]E \Leftarrow [\![\rho_0]\!]T$.

2. *If* $\Theta; \Delta; \Gamma \vdash \wr e \ ; \ \theta \wr \rightsquigarrow E{:}T/\Theta_1; \rho_1$ *then for any grounding hole instantiation* ρ_g *s.t.* $\cdot \vdash \rho_g : \Theta_1$ *and* $\rho_0 = \rho_g \circ \rho_1$*, we have* $[\![\rho_0]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]E \Rightarrow [\![\rho_g]\!]T$.

3. *If* $\Delta; \Gamma \vdash \wr pat \mapsto e \ ; \theta \wr : S \to T \rightsquigarrow \Pi\Delta'; \Gamma'.Pat : \theta' \mapsto E$ *then* $\Delta; \Gamma \vdash \Pi\Delta'; \Gamma'.Pat : \theta' \mapsto E \Leftarrow S \to T$.

Proof. By simultaneous induction on the first derivation.

For (1):

Case $\mathcal{D} : \Theta; \Delta; \Gamma \vdash \wr \mathbf{case}\, e\, \mathbf{of}\, \overrightarrow{b} \ ; \theta \wr : T \rightsquigarrow \mathbf{case}\, E\, \mathbf{of}\, \overrightarrow{B}/\Theta'; \rho$

$\Theta; \Delta; \Gamma \vdash \wr e \ ; \theta \wr \rightsquigarrow E{:}S/\cdot; \rho$ — by inversion on `el-case`

$[\![\rho]\!]\Delta; [\![\rho]\!]\Gamma \vdash \wr \overrightarrow{b} \ ; [\![\rho]\!]\theta \wr : S \to [\![\rho]\!]T \rightsquigarrow \overrightarrow{B}$ — by inversion on `el-case`

for any grounding hole inst. ρ' we have $[\![\rho]\!]\Delta; [\![\rho]\!]\Gamma \vdash E \Rightarrow S$ — by I.H. noting $\rho' = \cdot$ and $\rho' \circ \rho = \rho$

$[\rho]\Delta; [\rho]\Gamma \vdash B{:}S \to [\rho]T$ — for every branch by (3)

$[\rho]\Delta; [\rho]\Gamma \vdash \mathbf{case}\, E\, \mathbf{of}\, \overrightarrow{B} \Leftarrow [\![\rho]\!]T$ — by `t-case`

Note that because E is ground then the only grounding hole inst. is the empty substitution.

Case $\mathcal{D} : \Theta; \Delta; \Gamma \vdash \wr \mathbf{fn}\, x {\Rightarrow} e \ ; \theta \wr : T_1 \to T_2 \rightsquigarrow \mathbf{fn}\, x {\Rightarrow} E/\Theta_1; \rho_1$

$\Theta; \Delta; \Gamma, x{:}T_1 \vdash \wr e \ ; \theta \wr : T_2 \rightsquigarrow E/\Theta_1; \rho_1$ — by assumption

for any grounding hole inst. ρ_g we have: $[\![\rho_0]\!]\Delta; [\![\rho_o]\!](\Gamma, x{:}T_1) \vdash [\![\rho_g]\!]E \Leftarrow [\![\rho_0]\!]T_2$ — by i.h. (1) with $\rho_0 = \rho_g \circ \rho_1$

$[\![\rho_0]\!]\Delta; ([\![\rho_o]\!]\Gamma), x{:}([\![\rho_0]\!]T_1) \vdash [\![\rho_g]\!]E \Leftarrow [\![\rho_0]\!]T_2$ — by properties of substitution

$[\![\rho_0]\!]\Delta; [\![\rho_o]\!]\Gamma) \vdash \mathbf{fn}\, x {\Rightarrow} ([\![\rho_g]\!]E) \Leftarrow ([\![\rho_0]\!]T_1) \to ([\![\rho_0]\!]T_2)$ — by `t-fn`

$[\![\rho_0]\!]\Delta; [\![\rho_o]\!]\Gamma) \vdash [\![\rho_g]\!](\mathbf{fn}\, x {\Rightarrow} E) \Leftarrow [\![\rho_0]\!](T_1) \to T_2)$ — by properties of substitution

which is what we wanted to show

Case $\mathcal{D} : \Theta; \Delta; \Gamma \vdash \wr \lambda\, X {\Rightarrow} e \ ; \theta \wr : \Pi^e X{:}U.\, T \rightsquigarrow \lambda\, X {\Rightarrow} E/\Theta_1; \rho_1$

$\Theta; \Delta, X{:}U; \Gamma \vdash \wr e \ ; \theta, X/X \wr : T \rightsquigarrow E/\Theta_1; \rho_1$ — by assumption

for any grounding hole inst. ρ_g we have$[\![\rho_0]\!](\Delta, X{:}U); [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]E \Leftarrow [\![\rho_o]\!]T$ — by i.h.(1) with $\rho_0 = \rho_g \circ \rho_1$

$[\![\rho_o]\!]\Delta, X{:}([\![\rho_0]\!]U); [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]E \Leftarrow [\![\rho_o]\!]T$ — by properties of subst

$[\![\rho_o]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash \lambda\, X {\Rightarrow} [\![\rho_g]\!]E \Leftarrow \Pi^e X{:}[\![\rho_0]\!]U.\, ([\![\rho_o]\!]T)$ — by `t-mlam`

$[\![\rho_o]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]\lambda\, X {\Rightarrow} E \Leftarrow [\![\rho_0]\!]\Pi^e X{:}U.\, T$ — by properties of substitution

which is what we wanted to show

Case $\mathcal{D} : \Theta; \Delta; \Gamma \vdash \wr e \ ; \theta \wr : \Pi^i X{:}U.\, T \rightsquigarrow \lambda\, X {\Rightarrow} E/\Theta_1; \rho_1$

this case follows the same structure as the previous

Case $\mathcal{D} : \Theta; \Delta; \Gamma \vdash \wr [c] \ ; \theta \wr : [U] \rightsquigarrow [C]/\Theta_1; \rho_1$

$\Theta; \Delta \vdash \wr c \,;\, \theta \wr : U \rightsquigarrow C/\Theta_1; \rho_1$ by assumption

for any grounding inst. ρ_g we have $[\![\rho_0]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]C \Leftarrow [\![\rho_0]\!]U$ by properties of the index language and $\rho_0 = \rho_g \circ \rho_1$

$[\![\rho_0]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!][C] \Leftarrow [\![\rho_0]\!][U]$ by `t-box` and properties of subst.

which is what we wanted to show

Case $\mathcal{D} : \Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \wr : T \rightsquigarrow [\![\rho_2]\!]E/\Theta_2; \rho_2 \circ \rho_1$

$\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \wr \rightsquigarrow E{:}T_1/\Theta_1; \rho_1$
$\Theta_1; [\![\rho_1]\!]\Delta \vdash T_1 \doteq [\![\rho_1]\!]T/\Theta_2; \rho_2$ by assumption

for any grounding inst. ρ_g we have $[\![\rho_o]\!]\Delta; [\![\rho_0]\!]\Gamma \vdash [\![\rho_g]\!]E \Rightarrow [\![\rho_g]\!]T_1$ by i.h. (2) where $\rho_o = \rho_g \circ \rho_1$[*]

for any grounding inst. ρ_g' we have $[\![\rho_g' \circ \rho_2]\!]T_1 = [\![\rho_g' \circ \rho_2 \circ \rho_1]\!]T$ by prop of unification and applying a grounding subst [**]

$[\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash [\![\rho_g' \circ \rho_2]\!]E \Rightarrow [\![\rho_g' \circ \rho_2]\!]T_1$ from [*] using $\rho_g = \rho_g' \circ \rho_2$

$[\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash [\![\rho_g' \circ \rho_2]\!]E \Rightarrow [\![\rho_g' \circ \rho_2 \circ \rho_1]\!]T$ by [**]

$[\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash [\![\rho_g' \circ \rho_2]\!]E \Leftarrow [\![\rho_g' \circ \rho_2 \circ \rho_1]\!]T$ by `t-syn`

which is what we wanted to show

For(2):

Case $\mathcal{E} : \Theta; \Delta; \Gamma \vdash \wr e\,[c] \,;\, \theta \wr \rightsquigarrow E_1\,[C]{:}[C/X]([\![\rho_2]\!]T)/\Theta_2; \rho_2 \circ \rho_1$

$\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \wr \rightsquigarrow E_1{:}\Pi^e X{:}U.\,T/\Theta_1; \rho_1$

$\Theta_1; [\![\rho_1]\!]\Delta \vdash \wr c \,;\, [\![\rho_1]\!]\theta \wr : U \rightsquigarrow C/\Theta_2; \rho_2$ by assumption

for any grounding instantiation ρ_g s.t. $\cdot \vdash \rho_g{:}\Theta_1$ we have $[\![\rho_g \circ \rho_1]\!]\Delta; [\![\rho_g \circ \rho_1]\!]\Gamma \vdash [\![\rho_g]\!]E_1 \Rightarrow [\![\rho_g]\!]\Pi^e X{:}U.\,T$ by i.h. (2)[*]

for any grounding instantiation ρ_g' s.t. $\cdot \vdash \rho_g'{:}\Theta_2$ we have $[\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Delta \vdash [\![\rho_g']\!]C \Leftarrow [\![\rho_g' \circ \rho_2]\!]U$ by soundness of index reconstruction

$[\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash [\![\rho_g' \circ \rho_2]\!]E_1 \Rightarrow [\![\rho_g' \circ \rho_2]\!]\Pi^e X{:}U.\,T$ Note that in [*] $\cdot \vdash \rho_g{:}\Theta_1$ so we can instantiate $\rho_g = \rho_g' \circ \rho_2$

$[\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash [\![\rho_g' \circ \rho_2]\!]E_1 \Rightarrow \Pi^e X{:}([\![\rho_g' \circ \rho_2]\!]U).\,([\![\rho_g' \circ \rho_2]\!]T)$ by properties of substitutions

$[\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash ([\![\rho_g' \circ \rho_2]\!]E_1)\;[\![\rho_g']\!]C \Rightarrow [[\![\rho_g']\!]C\}/X]([\![\rho_g' \circ \rho_2]\!]T)$ by `t-app-index`

$[\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Delta; [\![\rho_g' \circ \rho_2 \circ \rho_1]\!]\Gamma \vdash [\![\rho_g']\!]((([\![\rho_2]\!]E_1)\;C) \Rightarrow [\![\rho_g']\!]([C/X]([\![\rho_2]\!]T)$ by properties of substitutions

which is what we wanted to show.

Case $\mathcal{E} : \Theta; \Delta; \Gamma \vdash \wr x \,;\, \theta \wr \rightsquigarrow E_1{:}T_1 \;/\; \Theta_1; \mathtt{id}(\Theta_1)$

$\Gamma(x) = T$

$\Theta; \Delta; \Gamma \vdash x{:}T \rightsquigarrow E_1{:}T_1/\Theta_1$ by assumption

$\Delta; \Gamma \vdash x \Rightarrow T$ by rule `t-var`[*]

for any grounding inst. ρ_g s.t. $\cdot \vdash \Theta_1$ we have:
$[\![\rho_g \circ \rho_1]\!]\Delta; [\![\rho_g \circ \rho_1]\!]\Gamma \vdash [\![\rho_g]\!]E_1{:}[\![\rho_g]\!]T_1$ by [*], weakening and lemma 4 with $\rho_1 = \mathtt{id}(\Theta_1)$

which is what we wanted to show

For (3):

Case $\mathcal{F} : \Delta; \Gamma \vdash \wr pat \mapsto e \,;\, \theta \wr : S \to T \rightsquigarrow \Pi\Delta_r; \Gamma_r.Pat'{:}\theta \mapsto E$

$\Delta \vdash pat : S \rightsquigarrow \Pi\Delta_r; \Gamma_r.Pat{:}\theta_r \mid \theta_e$ by assumption

$\cdot; \cdot \vdash pat \rightsquigarrow Pat : S'/\Theta_p; \Delta_p; \Gamma_p \mid \cdot$

$\Delta_p' \vdash \rho : \Theta_p$ and $\Gamma_r = [\theta_p][\![\rho]\!]\Gamma_p$, $Pat' = [\theta_p][\![\rho]\!]Pat$ by inversion on `el-subst`

$\Delta_p', [\![\rho]\!]\Delta_p; [\![\rho]\!]\Gamma_p \vdash [\![\rho]\!]Pat \Leftarrow [\![\rho]\!]S'$ by pattern elaboration lemma

$\Delta, \Delta'_p, [\![\rho]\!]\Delta_p \vdash [\![\rho]\!]S' \doteq S/\Delta_r, \theta$ by inversion on `el-subst`

where we can split θ as $\theta = \theta_r, \theta_i, \theta_e$ so that: $\begin{cases} \Delta_r \vdash \theta_r{:}\Delta \\ \Delta_r \vdash \theta_i{:}\Delta'_p \\ \Delta_r \vdash \theta_i, \theta_e{:}\Delta'_p, [\![\rho]\!]\Delta_p \end{cases}$

let $\theta_p = \theta_i, \theta_e$

$[\underbrace{\theta_i, \theta_e}_{\theta_p}][\![\rho]\!]S' = [\theta_r]S$ by soundness of unification and the fact that Δ and $\Delta'_p, [\![\rho]\!]\Delta_p$ are distinct

$\Delta_r; [\theta_p][\![\rho]\!]\Gamma_p \vdash [\theta_p][\![\rho]\!]Pat \Leftarrow [\theta_p][\![\rho]\!]S'$ by substitution lemma

$\Delta_r; \underbrace{[\theta_p][\![\rho]\!]\Gamma_p}_{\Gamma_r} \vdash \underbrace{[\theta_p][\![\rho]\!]Pat}_{Pat'} \Leftarrow [\theta_r]S$ by $[\theta][\![\rho]\!]S' = [\theta_r]S$

$\cdot; \Delta_r; [\theta_r]\Gamma, \Gamma_r \vdash \wr e\ ; \theta_r \circ \theta,\ \theta_e \int : [\theta_r]T \rightsquigarrow E/\cdot; \cdot$ by assumption

$\Delta_r; [\theta_r]\Gamma, \Gamma_r \vdash E \Leftarrow [\theta_r]T$ by (1)

$\Delta; \Gamma \vdash \Pi\Delta_r; \Gamma_r.Pat'{:}\theta_r \mapsto E \Leftarrow S \to T$ by `t-branch`

which is what we wanted to show.

□

Lemma 4 (Implicit parameter instantiation)**.** *Let's consider the judgement:* $\Theta; \Delta; \Gamma \vdash E{:}T \rightsquigarrow E_1{:}T_1/\Theta_1$*, where* Θ_1 *is a weakening of* Θ*.*

We want to prove that, if ρ_g *is a grounding instantiation such as* $\cdot \vdash \rho_g{:}\Theta_1$ *where we split* $\rho_g = \rho'_g, \rho''_g$ *and* $\cdot \vdash \rho'_g{:}\Theta$ *and* $\cdot; [\![\rho'_g]\!]\Delta; [\![\rho'_g]\!]\Gamma \vdash [\![\rho'_g]\!]E{:}[\![\rho'_g]\!]T$ *then* $\cdot; [\![\rho_g]\!]\Delta; [\![\rho_g]\!]\Gamma \vdash [\![rho_g]\!]E_1{:}[\![\rho_g]\!]T_1$*.*

Proof. The proof follows by induction on the rules of the judgment where the base case for `el-impl-done` is trivial and the inductive step for `el-impl` has also a very direct proof. □

Lemma 5 (Pattern elaboration)**.**

1. *If* $\Theta; \Delta \vdash pat \rightsquigarrow \Pi\Delta_1; \Gamma_1.Pat{:}T/\Theta_1; \rho_1$ *and* ρ_r *is a further refinement substitution, such as* $\Theta_2 \vdash \rho_r{:}\Theta_1$ *and* ϵ *is a ground lifting substitution, such as* $\Delta_i \vdash \epsilon{:}\Theta_1$ *then* $\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_r]\!]Pat \Leftarrow [\![\epsilon]\!][\![\rho_r]\!]T.$

2. *If* $\Theta; \Delta \vdash pat : T \rightsquigarrow \Pi\Delta_1; \Gamma_1.Pat/\Theta_1; \rho_1$ *and* ρ_r *is a further refinement substitution, such as* $\Theta_2 \vdash \rho_r{:}\Theta_1$ *and* ϵ *is a ground lifting substitution, such as* $\Delta_i \vdash \epsilon{:}\Theta_1$ *then* $\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_r]\!]Pat \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T.$

3. *If* $\Theta; \Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \Pi\Delta_1; \Gamma_1.\overrightarrow{Pat} \rangle\ S/\Theta_1; \rho_1$ *and* ρ_r *is a further refinement substitution, such as* $\Theta_2 \vdash \rho_r{:}\Theta_1$ *and* ϵ *is a ground lifting substitution, such as* $\Delta_i \vdash \epsilon{:}\Theta_1$ *then* $\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat} \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T \rangle\ [\![\epsilon]\!][\![\rho_r]\!]S.$

Proof. By simultaneous induction on the first derivation.

For (1):

Case $\mathcal{D} : \Theta; \Delta \vdash \mathbf{c}\,\overrightarrow{pat} \rightsquigarrow \Pi\Delta_1; \Gamma_1.\mathbf{c}\,\overrightarrow{Pat}{:}S/\Theta_1; \rho_1$

$\Sigma(\mathbf{c}) = T$

$\Theta; \Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \Pi\Delta_1; \Gamma_1.\overrightarrow{Pat} \rangle\ S/\Theta_1; \rho_1$ by assumption

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat} \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T \rangle\ [\![\epsilon]\!][\![\rho_r]\!]S$ by i.h. (3)

Note that types in the signature (i.e. Σ) are ground so $[\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T = T$

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash \mathbf{c}\,([\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat}) \Leftarrow [\![\epsilon]\!][\![\rho_r]\!]S$ by `t-pcon`.

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_r]\!](\mathbf{c}\,\overrightarrow{Pat}) \Leftarrow [\![\epsilon]\!][\![\rho_r]\!]S$ by properties of substitution

which is what we wanted to show.

For (2):

Case $\mathcal{E} : \Theta; \Delta \vdash x : T \rightsquigarrow \Pi\Delta_1\ ;\ \underbrace{x{:}T}_{\Gamma_1}.x\ /\ \Theta; \mathtt{id}(\Theta)$

$\Gamma_1(x) = T$ by x being the only variable in Γ_1

$[\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 = [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 T$ by applying ϵ and ρ_r to Δ_1, Γ_1 and T

$[\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash x \Leftarrow [\![\epsilon]\!][\![\rho_r]\!]T$ by rule `t-pvar` which is what we wanted to prove

For (3):

Case $\mathcal{F} : \Theta; \Delta \vdash pat\ \overrightarrow{pat} : T_1 \to T_2 \rightsquigarrow \Pi\Delta_2; \Gamma_1, \Gamma_2.(\llbracket\rho'\rrbracket Pat)\ \ \overrightarrow{Pat} \rangle\ S/\Theta_2; \rho_2 \circ \rho_1$

$\Theta; \Delta \vdash pat : T_1 \rightsquigarrow \Pi\Delta_1; \Gamma_1.Pat/\Theta_1; \rho_1$

$\Theta_1; \Delta_1 \vdash \overrightarrow{pat} : \llbracket\rho\rrbracket T_2 \rightsquigarrow \Pi\Delta_2; \Gamma_2.\overrightarrow{Pat} \rangle\ S/\Theta_2; \rho_2$ by assumption

$\Theta_2 \vdash \rho_2{:}\Theta_1$ by invariant of rule

$\Theta_3 \vdash \rho_3 \circ \rho_2{:}\Theta_1$ *(further refinement substitution)* by composition

$\Delta_i \vdash \epsilon{:}\Theta_3$ lifting substitution

$\Delta_i, \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket\Delta_1; \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket\Gamma_1 \vdash \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket Pat \Leftarrow \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2 \circ \rho_1\rrbracket T_1$ by i.h. on (1). [*]

$\Delta_i, \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Delta_2; \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Gamma_2 \vdash \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\overrightarrow{Pat} \Leftarrow \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2 \circ \rho_1\rrbracket T_2 \rangle\ \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket S$ by i.h. on (2)

we note that in pattern elaboration we have:

$\Delta_2 = \llbracket\rho_2\rrbracket\Delta_1, \Delta_2'$ Δ_2 is the context Δ_1 with the hole instantiation applied and some extra assumptions(i.e. Δ_2').

and $\Gamma_2 = \llbracket\rho_2\rrbracket\Gamma_1, \Gamma_2'$ Γ_2 is the context Γ_1 with the hole instantiation applied and some extra assumptions(i.e. Γ_2').

and we can weaken [*] to:

$\Delta_i, \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket\Delta_1, \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Delta_2'; \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket\Gamma_1, \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Gamma_2' \vdash \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket Pat \Leftarrow \llbracket\rho\rrbracket\llbracket\rho_3 \circ \rho_2 \circ \rho_1\rrbracket T_1$

$\Delta_i, \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Delta_2; \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Gamma_2 \vdash (\llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket Pat)(\llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\overrightarrow{Pat}) \Leftarrow \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2 \circ \rho_1\rrbracket T_1 \to \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2 \circ \rho_1\rrbracket T_2 \rangle\ \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket S$ by `t-sarr`.

$\Delta_i, \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Delta_2; \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Gamma_2 \vdash \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket(\llbracket\rho_2\rrbracket Pat\, \overrightarrow{Pat}) \Leftarrow \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2 \circ \rho_1\rrbracket(T_1 \to T_2) \rangle\ \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket S$ by properties of substitution

which is what we wanted to show.

Case $\mathcal{F} : \Theta; \Delta \vdash [c]\, \overrightarrow{pat} : \Pi^e X{:}U.\, T \rightsquigarrow \Pi\Delta_2; \Gamma_2.(\llbracket\rho_1\rrbracket[C])\ \overrightarrow{Pat} \rangle\ S/\Theta_2; \rho_2 \circ \rho_1$

$\Theta; \Delta \vdash c : U \rightsquigarrow C/\Theta_1; \Delta_1; \rho_1$

$\Theta_1; \Delta_1 \vdash \overrightarrow{pat} : [C/X]\llbracket\rho_1\rrbracket T \rightsquigarrow \Pi\Delta_2; \Gamma_2.\overrightarrow{Pat} \rangle\ S/\Theta_2; \rho_2$ by assumption

$\Theta_2 \vdash \rho_2{:}\Theta_1$ by invariant of rule

$\Theta_3 \vdash \rho_3 \circ \rho_2{:}\Theta_1$ *(further refinement substitution)* by composition

$\Delta_i \vdash \epsilon{:}\Theta_3$ lifting substitution

$\Delta_i, \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket\Delta_1 \vdash \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket C \Leftarrow \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2 \circ \rho_1\rrbracket U$ by property of the index language[*]

$\Delta_i, \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Delta_2; \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Gamma_2 \vdash \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\overrightarrow{Pat} \Leftarrow \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket([C/X]\llbracket\rho_1\rrbracket T) \rangle\ \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket S$ by i.h. (3)

as before, we note that:

$\Delta_2 = \llbracket\rho_2\rrbracket\Delta_1, \Delta_2'$ Δ_2 is the context Δ_1 with the hole instantiation applied and some extra assumptions(i.e. Δ_2').

and we can weaken [*] to:

$\Delta_i, \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket\Delta_1, \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Delta_2' \vdash \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket C \Leftarrow \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2 \circ \rho_1\rrbracket U$

Note that $\llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket([C/X]\llbracket\rho_1\rrbracket T) = [(\llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket C)/X](\llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2 \circ \rho_1\rrbracket T)$ by properties of substitution

$\Delta_i, \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Delta_2; \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Gamma_2 \vdash [\llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2\rrbracket C]\ (\llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\overrightarrow{Pat}) \Leftarrow \Pi^e X{:}(\llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2 \circ \rho_1\rrbracket U).\ (\llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2 \circ \rho_1\rrbracket T) \rangle\ \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket S$ by `t-spi`

$\Delta_i, \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Delta_2; \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket\Gamma_2 \vdash \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket([\llbracket\rho_2\rrbracket C]\ \overrightarrow{Pat})) \Leftarrow \llbracket\epsilon\rrbracket\llbracket\rho_3 \circ \rho_2 \circ \rho_1\rrbracket(\Pi^e X{:}U.\, T) \rangle\ \llbracket\epsilon\rrbracket\llbracket\rho_3\rrbracket S$ by properties of substitution

which is what we wanted to show.

Case $\mathcal{F} : \Theta; \Delta \vdash \overrightarrow{pat} : \Pi^i X{:}U.\, T \rightsquigarrow \Pi\Delta_1; \Gamma_1.(\llbracket\rho_1\rrbracket C)\ \overrightarrow{Pat} \rangle\ S/\Theta_1; \rho_1$

$\mathsf{genHole}\ (?Y : \Delta.U) = C$

$\Theta, ?Y{:}\Delta.U; \Delta \vdash \overrightarrow{pat} : [C/X]T \rightsquigarrow \Pi\Delta'; \Gamma'.\overrightarrow{Pat}/\Theta'; \rho \rangle\ S$ by assumption

$\Theta, ?Y{:}\Delta.U; \Delta \vdash C \Leftarrow U$ by genhole invariant

$\Delta_i, [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]\Delta \vdash [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]C \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]U$ applying substitutions $\epsilon, \rho_r and \rho_1$

noting that $\Delta_1 = [\![\rho_1]\!]\Delta, \Delta_1'$

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]([\![\rho_1]\!]\Delta, \Delta_1') \vdash [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]C \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]U$ by weakening

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma' \vdash [\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat} \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!][C/X]T \rangle [\![\epsilon]\!][\![\rho_r]\!]S$ by i.h. (3)

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma' \vdash [\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat} \Leftarrow [[\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]C/X]([\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T) \rangle [\![\epsilon]\!][\![\rho_r]\!]S$ by properties of substitution

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma' \vdash [[\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]C]\ [\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat} \Leftarrow \Pi^i X{:}[\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]U.\,([\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T) \rangle [\![\epsilon]\!][\![\rho_r]\!]S$ by `t-spi`

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma' \vdash [\![\epsilon]\!][\![\rho_r]\!][[\![\rho_1]\!]C]\ \overrightarrow{Pat} \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!](\Pi^i X{:}U.\,T) \rangle [\![\epsilon]\!][\![\rho_r]\!]S$ by properties of substitution

which is what we wanted to show

$\square$