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 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

General Terms term1, term2

Keywords keyword1, keyword2

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 allow programmers to omit (implicit) arguments which refine a given dependent type and can be reasonably easy inferred, an idea going back to (Pollack 1990). The objective is: the final code for dependently typed programs should not be significantly more complicated than their simply typed counter parts.

This is achieved by designing a source language where programmers can omit implicit argument, and elaborating it into a core language where type checking and fundamental properties such as normalization are well understood (Pollack 1990). However, this elaboration is rarely specified formally for dependently typed languages which support recursion and pattern matching and the conditions under which such an elaboration succeeds and a given source program is accepted are ill-understood. This makes it difficult for programmers to understand why a given program fails to type check; it also we believe one of the reasons that implementing depedently typed programming environments remains a black art known only to a few.

In this paper, we investigate the design of source language 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 (our index language) and write recursive programs about LF objects using pattern matching (Harper et al. 1993).

Our main contribution is the description of a source language for dependently typed programs where we omit implicit arguments together with a 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 allow us to concentrate on the key issues revolving around dependent case-expressions and pattern matching; it will also 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.

[*Add what the recipe in Agda and Coq is. -bp*]

We follow 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. Programmers may leave some index variables free when declaring a constant, i.e. a constructor in a datatype definition or a recursive program. As in the elaboration of LF

types, these free variables will be abstracted over and quantified at the outside. When using the constant subsequently, the programmer may not pass arguments for those index variables which were free when the constant was declared.

The key challenge is the elaboration of recursive programs which support case-analysis and pattern matching. In the dependently typed setting, pattern matching refines index arguments and hence refines types.

within this tradition; we keep the index domain abstract and assume that we have some elaboration for index terms. 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).

[*We also show completeness: given a well-typed program in our core language, we show that there exists a source program which can be elaborated to that program.*

★ *Talk about related work...*]

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, 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 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 begin by describing the *source language* that allows programmers to omit some arguments which we can infer 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 recursion (**rec** $f{:}t = e$), functions (**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 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 subexpression 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 reconstruction regarding the type of pattern variables.

We also support writing underscore (_); although our grammar allows programmers to write underscores everywhere, they can in fact only stand for inded objects, but not arbitrary expressions.

One may think of our source language as the language obtained after parsing; in other words, we may support more syntactic sugar, in particular, we may want to support let-expressions which can be modelled as a case-expression with one branch.

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

Figure 1. Grammar of Source Language

Types for computations include non-dependent function types ($t_1 \to t_2$) and dependent function types ($X{:}ut$); 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 emphasize that computation-level types can only be indexed by terms from an index domain u.

We note that we do only support one form of dependent function type $X{:}ut$; 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 follow the reciple advocated in the *Elf* language (Pfenning 1989) and subsequently implemented also in Twelf (Pfenning and Schürmann 1999): we 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 need to make our syntax more precise and make sure that our expression's variables are correctly scoped. In figure 2 we define the well formed judgement and its inference rules.

[

- *The judgment for $\delta \vdash t$ wf isn't quite satisfying because it doesn't distinguish when variables in t can be free and when they cannot. -bp*

]

2.2 Example: Elaboration of untyped terms into intrinsically typed terms

We consider here the elaboration of a simple language with numbers, booleans and some primitive operations to its typed counterpart. We first define the untyped version of our language using a recursive datatype. `ctype` defines the new type `UTm`.

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

Terms in this language can be of type `nat` for numbers or `bool` for booleans. Notice how `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.

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

$$\frac{\delta;\gamma, f \vdash e\ \text{wf} \quad \cdot \vdash_d t\ \text{wf}}{\delta;\gamma \vdash \textbf{rec}\ f{:}t = e\ \text{wf}}\ \texttt{wf-rec}$$

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

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

$$\frac{\delta;\gamma \vdash e_1\ \text{wf}}{\delta;\gamma \vdash e_1\ \_\ \text{wf}}\ \texttt{wf-apph} \quad \frac{\delta \vdash c\ \text{wf}}{\delta;\gamma \vdash [c]\ \text{wf}}\ \texttt{wf-box} \quad \frac{\delta;\gamma \vdash e\ \text{wf} \quad \text{for all } b_n \text{ in } \vec{b}\ .\ \delta;\gamma \vdash b_n\ \text{wf}}{\delta;\gamma \vdash \textbf{case}\ e\ \textbf{of}\ \vec{b}\ \text{wf}}\ \texttt{wf-case}$$

$$\frac{x \in \gamma}{\delta;\gamma \vdash x\ \text{wf}}\ \texttt{wf-var} \quad \frac{\delta;\gamma \vdash e\ \text{wf} \quad \delta \vdash t\ \text{wf}}{\delta;\gamma \vdash e{:}t\ \text{wf}}\ \texttt{wf-ann}$$

$\boxed{\delta;\gamma \vdash pat \mapsto e\ \text{wf}}$ 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

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

Figure 2. Well-formed source expressions

```
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` and elaboration will infer it. One might ask how we can provide the type explicitly - this is possible indirectly by providing type annotations. For example, when we write `If e e1 e2` one might wonder how we can fix the type of `e1` to be `Tm [nat]`. This can be for example accomplished by writing `If e (e1:TM[nat])e2`, i.e. providing a type annotation on `e1`.

We want to type-check `UTm` terms by writing a function that takes terms in the untyped representation into the 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 not only we 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 notable 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).

In the definition of type `TmOpt` we chose an explicit quantification over `T`, however another option would have been to leave it implicit. In `tc` we pattern match over the type quantified in the `Some` constructor, because in our language the programmer cannot supply values for implicit parameters, we would need to provide type annotations to constrain the types for if-expressions. In particular, if we had implicitly quantified over `T` in the `Some` constructor, we would have to provide type annotations in various places in order to constrain types. For example, the case for `UIf` would be:

```
| 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)
```

By including type annotations for `c'`, `e1'` and `e2'` we can constrain them to be of the required types.

2.3 Example 2: Type-preserving evaluation

We may also want to evaluate our now type correct programs to values of the same type. 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 can define a type preserving evaluator using our typed values 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 `{T:tp} Tm [T] →Val [T]`. Elaboration will need to translate the given program into a program in our core language which has type `{T:tp} Tm [T] →Val [T]`. It will first need to introduce the appropriate dependent function 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. 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`.

Kinds	K	::=	$\texttt{ctype} \mid \overset{e}{\Pi} X{:}U.\, K$
Atomic Types	P	::=	$\mathbf{a}\,\vec{C}$
Types	T	::=	$P \mid [U] \mid \overset{e}{\Pi} X{:}U.\, T \mid$
			$\overset{i}{\Pi} X{:}U.\, T \mid T_1 \to T_2$
Expressions	E	::=	$\mathbf{rec}\, f{:}T = E \mid \mathbf{fn}\, x{\Rightarrow}E \mid \lambda\, X{\Rightarrow}E$
			$\mid E_1\, E_2 \mid E_1\, [C] \mid [C] \mid$
			$\mathbf{case}\, E\, \mathbf{of}\, \vec{B} \mid x \mid E{:}T \mid \mathbf{c}$
Branches	$\vec{B}$	::=	$B \mid (B \mid \vec{B})$
Branch	B	::=	$\Pi\Delta; \Gamma.\, Pat{:}\theta \mapsto E$
Pattern	Pat	::=	$x \mid [C] \mid \mathbf{c}\, \overrightarrow{Pat}$
Context	Γ	::=	$\cdot \mid \Gamma, x{:}T$
Meta-Context	Δ	::=	$\cdot \mid \Delta, X{:}U$
Meta-Substitution	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 is indexed by elaborated fully explicit terms of the index level language; we use C for fully explicit index level objects, and U for elaborated types; index-variables occurring in the target language will be represented by capital letters such as X, Y[1].

Our target language (Fig. 3) is very similar to our source language. The main difference 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 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.

The introductions, functions **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. When we substitution C for X in T, we assume that substitution is defined in such a way that normal forms are preserved[2]. We also note that we only give the rules overload the Pi-type

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 lookup the type T of the constant **c** in the signature and

[1] [*explain meta-substitutions. -bp*]

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

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

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

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

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

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

$$\frac{\Delta, X{:}U;\Gamma \vdash E \Leftarrow T}{\Delta;\Gamma \vdash \lambda X \Rightarrow E \Leftarrow \overset{e}{\Pi} X{:}U.\,T}\ \texttt{t-mlam} \qquad \frac{\Delta;\Gamma \vdash E \Rightarrow S \quad \Delta;\Gamma \vdash \overrightarrow{B} \Leftarrow S \to T}{\Delta;\Gamma \vdash \textbf{case}\ E\ \textbf{of}\ \overrightarrow{B} \Leftarrow T}\ \texttt{t-case}$$

$\boxed{\Delta;\Gamma \vdash \Pi\Delta';\Gamma'.Pat{:}\theta \mapsto E \Leftarrow T}$ 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

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

Figure 4. Typing of computational expressions

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.

3.1 Elaborated examples

We show here the result of elaboration for the type-preserving evaluator given in Section 2.3.

★ Explain elaborated example below

```
rec eval: {T:tp}Tm [T]→Val [T] = λT ⇒ 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⇒
  case eval [bool] e of
  | T:tp ; . VTrue : T/T ⇒ eval [T] e1
  | T:tp ; . VFalse : T/T ⇒ eval [T] e2
```

4. Description of Elaboration

Elaboration of our source-language to our core target language is bi-directional and guided by the expected target type.

★ Explain bidirectional reconstruction (reconstructing against a type and reconstructing while synthesizing a type) and related it to bidirectional type-checking. Cite for bidirectional type checking Pierce and Turner 1998 and DML and ATS papers with bidirectional reconstruction

Recall that the target type marks the argument which is implicitly quantified (see $\overset{i}{\Pi} X{:}U.\,T$). If we check a source expression against $\overset{i}{\Pi} X{:}U.\,T$ we insert the appropriate $\lambda \Rightarrow$-abstraction in our target. If we have synthesized the type $\overset{i}{\Pi} X{:}U.\,T$ 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 Holes

Elaboration relies on *hole variables* ($?X$) to denote missing index-level terms that will be filled by elaboration. Following (Nanevski et al. 2008), holes are associated with a substitution to express which index-level-variables from the context Δ they can refer to (i.e. $?X[\theta]$). Contextual types, where we pair the type U together with the context Δ it is allowed to depend on, are a natural way to describe the type of hole variables. As soon as we know what $?X$ stands for C, we apply θ to C filling the hole. We define all hole variables in a context Θ[3].

Hole vars		::=	$?X[\theta]$
Hole types		::=	$[\Delta.U]$
Hole Contexts	Θ	::=	$\cdot \mid \Theta, ?X{:}[\Delta.U]$
Hole Inst.	ρ	::=	$\cdot \mid \rho, \Delta.C/?X$

Hole variables occur only in index terms; they do not denote computation-level expressions. 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.

$$\mathsf{genHole}\ (?Y : \Delta.U) \quad = ?Y[\mathtt{id}(\Delta)]$$

4.2 Unification

The notion of unification that reconstruction 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. For, Beluga which allows index term to be drawn from the logical framework LF, 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:

Unification of index-level terms

$\boxed{\Theta; \Delta \vdash C_1 \doteq C_2/\Theta'; \rho}$ where: $\Theta' \vdash \rho{:}\Theta$

Unification of computation-level types

$\boxed{\Theta; \Delta \vdash T_1 \doteq T_2/\Theta'; \rho}$ where: $\Theta' \vdash \rho{:}\Theta$

If unification succeeds, then we have $[\![\rho]\!]C_1 = [\![\rho]\!]C_2$ and $[\![\rho]\!]T_1 = [\![\rho]\!]T_2$.

4.3 Elaboration

4.4 Elaboration of index terms

Our elaboration of programs assumes that we know how to elaborate index terms; in the case of Beluga, we follow elaboration as described in (Pientka 2013) for terms in the logical framework LF[4]. Here, we again simply state our requirement on the index domain.

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

These judgements reconstruct an index level term or type in a closure where the substitution θ represents the refinements that occur during pattern matching and that we lazily apply during reconstruction. 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 three fold: a context Δ' which in addition to all the declarations in Δ also contains index variables together with their types for all index variables which were free in c; a context Θ' of holes is related to the original hole context Θ via the hole instantiation ρ.

4.5 Elaborating source expressions

As mentioned earlier, we elaborate expressions in a bi-directional way; expressions such as recursion, 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. Because of pattern matching, index variables in Δ may get refined to concrete index terms and we need to take into account these refinements when elaborating branches. We therefore elaborate a source expression together with a refinement substitution ρ; we note that ρ technically maps source index variables to their corresponding target refinement. We give the rules for elaborating source expressions in checking mode in Fig. 5 and in syntesis mode in Fig. 6.

$$\begin{array}{ll} \text{Synthesizing:} & \Theta; \Delta; \Gamma \vdash \wr e \ ; \theta \wr \rightsquigarrow E{:}T/\Theta'; \rho \\ \text{Checking:} & \Theta; \Delta; \Gamma \vdash \wr e \ ; \theta \wr : T \rightsquigarrow E/\Theta'; \rho \end{array}$$

Here Θ describes all the holes we have introduced so far; Δ describes variables from our index domain occurring in e; Γ describes program variables occurring in e. The result of elaboration in checking mode is described by E together with a new context of holes Θ' and a hole instantiation ρ, s.t. $[\![\rho]\!]E$ has type $[\![\rho]\!]T$ where $\Theta' \vdash \rho : \Theta$[5].

The result of elaboration in synthesis mode is similar; we return the target expression E together 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. $[\![\rho]\!]E$ has type $[\![\rho]\!]T$.

To elaborate a function (see rule `el-fn`) we simple elaborate the body extending the context Γ. To elaborate a dependent function (see rule `el-mlam`), we elaborate the body extending the context Δ. When switching to syntesis mode, we elaborate $\wr e \ ; \theta \wr$ 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 T obtaining an instantiation ρ' and return the composition of the instantiation ρ and ρ'.

The key cases are the 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.

[*Explain the rules. -bp*]

We now consider the elaboration rules in synthesis mode give in Fig. 6.

4.5.1 Elaborating branches and patterns

The judgement for branches:

$$\Theta; \Delta; \Gamma \vdash \wr b \ ; \theta \wr : T_s \to T_2 \rightsquigarrow B/\Theta'; \rho$$

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

[3] [*Is there an order on them? -bp*]

[4] [*This needs to be slightly adopted. -bp*]

[5] [*E and e should be related. -bp*]

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

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \wr \rightsquigarrow E{:}T'/\Theta'; \rho \quad \Theta'; [\![\rho]\!]\Delta \vdash T' \doteq [\![\rho]\!]T/\Theta''; \rho'}{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \wr : T \rightsquigarrow [\![\rho']\!]E/\Theta''; \rho' \circ \rho}\ \texttt{el-syn} \qquad \frac{\Theta; \Delta; \Gamma, x{:}T_1 \vdash \wr e \,;\, \theta \wr : T_2 \rightsquigarrow E/\Theta'; \rho}{\Theta; \Delta; \Gamma \vdash \wr \mathbf{fn}\, x \Rightarrow e \,;\, \theta \wr : T_1 \to T_2 \rightsquigarrow \mathbf{fn}\, x \Rightarrow E/\Theta'; \rho}\ \texttt{el-fn}$$

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

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

$$\frac{\Theta; \Delta \vdash \wr c \,;\, \theta \wr : U \rightsquigarrow C/\Theta'; \rho}{\Theta; \Delta; \Gamma \vdash \wr [c] \,;\, \theta \wr : [U] \rightsquigarrow [C]/\Theta'; \rho}\ \texttt{el-box}$$

Figure 5. Elaboration of Expressions (Checking Mode)

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

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e_1 \,;\, \theta \wr \rightsquigarrow E_1 : S \to T \;/\; \Theta'; \rho \qquad \Theta'; [\![\rho]\!]\Delta; [\![\rho]\!]\Gamma \vdash \wr e_2 \,;\, [\![\rho]\!]\theta \wr : [\![\rho]\!]S \rightsquigarrow E_2 \;/\; \Theta''; \rho'}{\Theta; \Delta; \Gamma \vdash \wr e_1\, e_2 \,;\, \theta \wr \rightsquigarrow E_1\, E_2 : [\![\rho']\!]T \;/\; \Theta''; \rho' \circ \rho}\ \texttt{el-app}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \wr \rightsquigarrow E{:}\overset{e}{\Pi} X{:}U.\, T/\Theta'; \rho \qquad \Theta'; [\![\rho]\!]\Delta \vdash \wr c \,;\, [\![\rho]\!]\theta \wr : U \rightsquigarrow C/\Theta''; \rho'}{\Theta; \Delta; \Gamma \vdash \wr e\, [c] \,;\, \theta \wr \rightsquigarrow E\, [C]{:}[C/X]([\![\rho']\!]T)/\Theta''; \rho' \circ \rho}\ \texttt{el-mapp}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \wr \rightsquigarrow E : \overset{i}{\Pi} X{:}U.\, T \;/\; \Theta'; \rho \quad \mathsf{genHole}\ (?Y : \Delta.U) = C}{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \wr \rightsquigarrow E\, [C] : [\Delta.C/X]T \;\;/\;\; \Theta', ?Y{:}[\![\rho]\!][\Delta.U] \,;\, \rho}\ \texttt{el-mapp-implicit}$$

$$\frac{\Theta; \Delta; \Gamma \vdash \wr e \,;\, \theta \wr \rightsquigarrow E{:}\overset{e}{\Pi} X{:}U.\, T/\Theta'; \rho \quad \mathsf{genHole}\ (?Y : \Delta.U) = C}{\Theta; \Delta; \Gamma \vdash \wr e\_ \,;\, \theta \wr \rightsquigarrow E\, [C] : [\Delta.C/X]T \;\;/\;\; \Theta', ?Y{:}[\![\rho]\!][\Delta.U] \,;\, \rho}\ \texttt{el-mapp-underscore}$$

$$\frac{\Theta; \Delta \vdash \wr t \,;\, \theta \wr \rightsquigarrow T/\Theta'; \rho \qquad \Theta'; [\![\rho]\!]\Delta; [\![\rho]\!]\Delta \vdash \wr e \,;\, [\![\rho]\!]\theta \wr : T \rightsquigarrow E/\Theta''; \rho'}{\Theta; \Delta; \Gamma \vdash \wr e{:}t \,;\, \theta \wr \rightsquigarrow (E{:}T){:}T/\Theta''; \rho' \circ \rho}\ \texttt{el-annotated}$$

$$\frac{\Theta; \Delta \vdash \wr t \,;\, \theta \wr \rightsquigarrow T/\Theta'; \rho \qquad \Theta'; [\![\rho]\!]\Delta; [\![\rho]\!]\Gamma, f{:}T \vdash \wr e \,;\, [\![\rho]\!]\theta \wr : T \rightsquigarrow E/\Theta''; \rho'}{\Theta; \Delta; \Gamma \vdash \wr \mathbf{rec}\, f{:}t = e \,;\, \theta \wr \rightsquigarrow \mathbf{rec}\, f{:}[\![\rho']\!]T = E{:}[\![\rho']\!]T/\Theta''; \rho' \circ \rho}\ \texttt{el-rec}$$

Figure 6. Elaborating of Expressions (Synthesizing Mode)

Intuitively, to elaborate a branch, we need to elaborate the pattern, synthesizing the type of free pattern variables occurring inside of it (Γ_r). In the dependently typed setting, pattern elaboration needs to do more work: we need to infer implicit arguments which were omitted by the programmer (Δ_r) and we need to synthesize a pattern type T_p which is compatible with the expected type T_s of the scrutinee, i.e. T_p is an instance of T_s and $[\theta]_p T_s = T_p$. Elaboration of the pattern is described by the judgment

$$\Delta \vdash pat : T_p \rightsquigarrow Pat : \theta_r/\Delta_r; \Gamma_r$$

[

- *Explain elaboration of patterns here.*
- *In* `rb-branch` *unification is different from the judgement previously mentioned, discuss how and why this one is different -ff*

]

Technically, inferring the type of free variables and reconstructing most general arguments for omitted parts is done in three steps.

1. First, given pat we elaborate it to a target pattern Pat together with its type S_1 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 `rb-subst`.

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

2. We now abstract over the hole variables in Θ_p and replace them with fresh index variables from Δ'_p. This is accomplished by the second premise of the rule `rb-subst`:

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

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

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

$$\frac{\cdot;\cdot \vdash pat \rightsquigarrow Pat : S/\Theta_p;\Delta_p;\Gamma_p \mid \cdot \qquad \Delta'_p \vdash \rho : \Theta_p \qquad \Delta,(\Delta'_p,[\![\rho]\!]\Delta_p) \vdash [\![\rho]\!]S \doteq T/\Delta_r;\theta}{\Delta \vdash pat : T \rightsquigarrow \Pi\Delta_r;[\theta_p][\![\rho]\!]\Gamma_p.[\theta_p][\![\rho]\!]Pat : \theta_r \mid \theta_e}\ \texttt{rb-subst}$$

where $\Delta'_p \vdash \rho : \Theta_p$ constructs a ground lifting substitution ρ to new index variables Δ'_p given Θ_p

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

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

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

Figure 7. Branches and patterns

$\Delta'_p \vdash \rho' : \Theta_p$

ρ is the lifting substitution which replaces holes with index variables bound in Δ'_p

3. Finally, we compute the refinement substitution θ_R which ensures that the type of the pattern $[\![\rho]\!]S_1$ is compatible with the type T_1 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

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

   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.

   The careful reader will notice that our unification judgement is not the same as the one presented in Section 4.2. In this case, we need unification for finding instances of index-level variables, while the judgment form Section 4.2 finds instances of hole variables[6].

   [

   - *top-level pattern is reconstructed while synthesizing its type, so for some cases a type annotation might be needed ★ (Notably, a variable that catches all requires an annotation, this strikes me as unnecessary, something might be needed for this case) . We don't want refine outer holes with pattern reconstruction (otherwise we would have to deal with potentially different refinements of a hole variable from several branches) and because we require the type of the scrutinee (i.e. T_1) to be closed pattern reconstruction starts with an empty hole context, and produces an empty substitution and a new context Θ_p that may only add new holes. -ff*
     I don't understand this paragraph. -bp
   - *I removed abstraction and replaced it simply by the lifting substitution. Please take a look and see if it does what it is supposed to do. I don't discuss why U must be closed, i.e. does not depend on Δ. -bp*

   ]

[6] [*Add justification -bp*]

4.5.2 Synthesis versus check in patterns

Currently, the rule for branch reconstruction synthesizes the type of the pattern, however if the pattern was just a variable, it won't be synthesizable, while it should match whatever type we need. To solve this problem we could provide another rule for branch reconstruction (e.g. `rb-branch-2`) and use them according to the available rules. Another solution would be to generate a type skeleton and always perform checking against said type.

4.5.3 On the need of `mcase`

The rule `rb-case` does pattern matching on computational objects, and builds the type $T' \to T$ that is used to reconstruct branches in the rule `rb-branch`. However, one needs to be careful, because the language pattern matches on computational types but also on boxed items, so if the scrutinee is of a boxed type, morally the type generated for the branches should be:$\overset{e}{\Pi} X{:}U.\, T$ and X may occur in T and so the refinements that the branches cause.

4.5.4 Reconstructing Patterns

Patterns are also reconstructed with a bidirectional algorithm. Reconstruction starts by checking the type of the pattern against the type of the scrutinee. Again, note that in `rb-case` the type of the scrutinee type cannot contain holes (i.e. the Θ context must be empty). During pattern reconstruction, new holes will be inserted for implicit parameters, however this will only happen during the synthesize phase, so the `r-psyn` rule will try to instantiate the holes when unifying the type of the pattern and the scrutinee.

These judgements appear more complex than those for terms due to the new suffixes of the contexts Δ and Γ they return. These extensions of the contexts are required because patterns bind new variables in those contexts.

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

$$\frac{}{\Theta; \Delta \vdash x : T \rightsquigarrow x/\Theta; \Delta; \cdot, x{:}T \mid \mathtt{id}(\Theta)}\ \mathtt{r\text{-}pvar} \qquad \frac{\Theta; \Delta \vdash c : U \rightsquigarrow C/\Theta'; \Delta'; \rho}{\Theta; \Delta \vdash [c] : [U] \rightsquigarrow [C]/\Theta'; \Delta'; \cdot \mid \rho}\ \mathtt{r\text{-}pindex}$$

$$\frac{\Theta; \Delta \vdash pat \rightsquigarrow Pat{:}S/\Theta'; \Delta'; \Gamma \mid \rho \qquad \Theta'; \Delta' \vdash S \doteq [\![\rho]\!]T/\rho'; \Theta''}{\Theta; \Delta \vdash pat : T \rightsquigarrow [\![\rho]\!]Pat/\Theta''; [\![\rho']\!]\Delta'; [\![\rho']\!]\Gamma \mid \rho' \circ \rho}\ \mathtt{r\text{-}psyn}$$

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

$$\frac{\Sigma(c) = T \quad \Theta; \Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \overrightarrow{Pat}/\Theta'; \Delta'; \Gamma \mid \rho \rangle S}{\Theta; \Delta \vdash \mathbf{c}\,\overrightarrow{pat} \rightsquigarrow \mathbf{c}\,\overrightarrow{Pat}{:}S/\Theta'; \Delta'; \Gamma \mid \rho}\ \mathtt{r\text{-}pcon}$$

$$\frac{\cdot; \cdot \vdash \wr t\ ;\ \cdot \wr \rightsquigarrow T/\Theta'; \Delta'; \rho \quad \Theta, \Theta'; [\![\rho]\!]\Delta, \Delta' \vdash pat : T \rightsquigarrow Pat/\Theta''; \Delta''; \Gamma \mid \rho'}{\Theta; \Delta \vdash (pat{:}t) \rightsquigarrow Pat{:}[\![\rho']\!]T/\Theta''; \Delta''; \Gamma \mid \rho' \circ \rho}\ \mathtt{r\text{-}pann}$$

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

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

$$\frac{\Theta; \Delta \vdash pat : T_1 \rightsquigarrow Pat/\Theta'; \Delta'; \Gamma \mid \rho \qquad \Theta'; \Delta' \vdash \overrightarrow{pat} : [\![\rho]\!]T_2 \rightsquigarrow \overrightarrow{Pat}/\Theta''; \Delta''; \Gamma' \mid \rho' \rangle S}{\Theta; \Delta \vdash pat\ \overrightarrow{pat} : T_1 \to T_2 \rightsquigarrow ([\![\rho']\!]Pat)\ \overrightarrow{Pat}/\Theta''; \Delta''; \Gamma, \Gamma' \mid \rho' \circ \rho \rangle S}\ \mathtt{r\text{-}sp\text{-}cmp}$$

$$\frac{\Theta; \Delta \vdash c : U \rightsquigarrow C/\Theta'; \Delta'; \rho \quad \Theta'; \Delta' \vdash \overrightarrow{pat} : [C/X][\![\rho]\!]T \rightsquigarrow \overrightarrow{Pat}/\Theta''; \Delta''; \Gamma \mid \rho' \rangle S}{\Theta; \Delta \vdash [c]\,\overrightarrow{pat} : \overset{e}{\Pi} X{:}U.\,T \rightsquigarrow ([\![\rho']\!][C])\ \overrightarrow{Pat}/\Theta''; \Delta''; \Gamma \mid \rho' \circ \rho \rangle S}\ \mathtt{r\text{-}sp\text{-}explicit}$$

$$\frac{\mathsf{genHole}\ (?Y{:}\Delta.U) = C \qquad \Theta, ?Y{:}\Delta.U; \Delta \vdash \overrightarrow{pat} : [C/X]T \rightsquigarrow \overrightarrow{Pat}/\Theta'; \Delta'; \Gamma \mid \rho \rangle S}{\Theta; \Delta \vdash \overrightarrow{pat} : \overset{i}{\Pi} X{:}U.\,T \rightsquigarrow ([\![\rho]\!]C)\ \overrightarrow{Pat}/\Theta'; \Delta'; \Gamma \mid \rho \rangle S}\ \mathtt{r\text{-}sp\text{-}implicit}$$

Figure 8. Elaboration of patterns and pattern spines

Spines

5. Soundness of reconstruction

★ Explain the main lemmas, and specifically what properties we assume of the main judgements

Theorem 1 (Soundness)**.**

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

Lemma 2 (Branches)**.**
If $\Theta; \Delta; \Gamma \vdash \wr b\ ;\ \theta \wr : T_1 \to T_2 \rightsquigarrow B/\Theta'; \rho$
then $\Theta'; [\![\rho]\!]\Delta; [\![\rho]\!]\Gamma \vdash B \Leftarrow [\![\rho]\!]T_1 \to [\![\rho]\!]T_2$.

Lemma 3 (Refinement)**.**

- *If* $\Theta; \Delta \vdash pat : T \rightsquigarrow Pat/\Theta_r; \Delta_r; \Gamma_r \mid \rho_r$ *then* $\Theta_r \vdash \rho_r{:}\Theta$ *and* $\Theta_r; \Delta_r; \Gamma_r \vdash Pat \Leftarrow [\![\rho_r]\!]T$
- *If* $\Theta; \Delta \vdash pat \rightsquigarrow Pat{:}T/\Delta_r; \Gamma_r \mid \theta_r$ *then* $\Theta; \Delta_r \vdash \theta_r{:}\Delta$ *and* $\cdot; \Delta_r; \Gamma_r \vdash Pat \Leftarrow T$

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 we support pattern matching on index objects. Our elaboration, in contrast to the elaboration 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 (Pollack 1990) although no concrete algorithm to reconstruct omitted arguments was given. Luther (**?**) refined these ideas as part of the TYPELab project. He describes an elaboration and reconstruction for the calculus of construction without treating recursive functions and pattern matching.

Our approach is different from the one found in Agda (Norell 2007), Idris, Coq or Matita, where all variables occurring in a type need to be abstracted over when a constant is declared. In all three systems, abstractions, which denote arguments the user can freely omit, are statically labelled as such. Both systems give the user the possibility to locally override the implicit arguments mechanism.

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.

There is little work on elaborating dependently-typed source language supporting recursion and pattern matching. (Norell 2007) for example describes the bi-directional type inference algorithm implemented in the Agda. However, he concentrates on a core dependently typed calculus enriched with dependent pairs, but omits the rules for its extension with recursion and pattern matching.

A notable exception, is the work by (Asperti et al. 2012) on describing a bi-directional elaboration algorithm for the Calculus of (Co)Inductive Constructions implemented in for Matita. Their setting is very different from ours: they are much more ambitious 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, we are only allowed to write total programs and all types must be positive. For these reasons their source language is more verbose than ours; in particular when writing case-expressions, the programmer needs to supply the overall type[7] as an annotation. And the target language is very different as well; while we have simple decidable type checking algorithm and store the refinement substitutions in each branch, their system ??? . The elaboration to the target language is more complex than ours.

[

- *Does Matita impose similar "closed" requirements as we do on the type of the scrutinee of a pattern?*

]

Another related work is the elaboration of Idris (Brady 2013) which uses a different technique. Idris starts by adding holes at 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. They do not prove soundness of the elaboration, still they propose an a reverse elaboration phase. The authors conjecture that given a type correct program its elaboration followed by a reverse elaboration produces a matching source level program.

7. Conclusion

This paper concludes.

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.

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.

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 metaprogramming. 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.

[7] [*Check -bp*]

A. Soundness Proof

Theorem 4 (Soundness)**.**

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

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

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

$\Theta; \Delta; \Gamma \vdash \wr e \ ; \ \theta \wr \rightsquigarrow E{:}S/\cdot; \rho$ 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)

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

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

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 `rb-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 `rb-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 $[\theta][\![\rho]\!]S' = [\theta_r]S$

$\cdot; \Delta_r; [\theta_r]\Gamma, \Gamma_r \vdash \wr e \ ; \ \theta_r \circ \theta, \ \theta_e \wr : [\theta_r]T \rightsquigarrow E/\cdot; \cdot$ 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 5 (Pattern elaboration)**.**

1. *If* $\Theta; \Delta \vdash pat \rightsquigarrow Pat{:}T/\Theta_1; \Delta_1; \Gamma_1 \mid \rho_1$ *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* $\Delta_i \vdash \epsilon{:}\Theta_2$
   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* $\Theta; \Delta \vdash pat : T \rightsquigarrow Pat/\Theta_1; \Delta_1; \Gamma_1 \mid \rho_1$ *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* $\Delta_i \vdash \epsilon{:}\Theta_2$
   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* $\Theta; \Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \overrightarrow{Pat}/\Theta_1; \Delta_1; \Gamma_1 \mid \rho_1 \rangle S$ *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* $\Delta_i \vdash \epsilon{:}\Theta_2$
   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 $\mathcal{D} : \Theta; \Delta \vdash \mathbf{c}\,\overrightarrow{pat} \rightsquigarrow \mathbf{c}\,\overrightarrow{Pat}{:}S/\Theta_1; \Delta_1; \Gamma_1 \mid \rho_1$

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

$\Theta; \Delta \vdash \overrightarrow{pat} : T \rightsquigarrow \overrightarrow{Pat}/\Theta_1; \Delta_1; \Gamma_1 \mid \rho_1 \rangle S$ 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 (3):

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

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

$\Theta_1; \Delta_1 \vdash \overrightarrow{pat} : [\![\rho]\!]T_2 \rightsquigarrow \overrightarrow{Pat}/\Theta_2; \Delta_2; \Gamma_2 \mid \rho_2 \rangle S$ 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

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

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

we note that in pattern elaboration we have:

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

and $\Gamma_2 = [\![\rho_2]\!]\Gamma_1, \Gamma_2'$ 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:

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

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

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

which is what we wanted to show.

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

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

$\Theta_1; \Delta_1 \vdash \overrightarrow{pat} : [C/X][\![\rho_1]\!]T \rightsquigarrow \overrightarrow{Pat}/\Theta_2; \Delta_2; \Gamma_2 \mid \rho_2 \rangle S$ 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

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

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

as before, we note that:

$\Delta_2 = [\![\rho_2]\!]\Delta_1, \Delta_2'$ 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:

$\Delta_i, [\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]\Delta_1, [\![\epsilon]\!][\![\rho_3]\!]\Delta_2' \vdash [\![\epsilon]\!][\![\rho_3 \circ \rho_2]\!]C \Leftarrow [\![\epsilon]\!][\![\rho_3 \circ \rho_2 \circ \rho_1]\!]U$

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

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

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

which is what we wanted to show.

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

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

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

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

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat} \Leftarrow [[\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]C/X]([\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T) \rangle [\![\epsilon]\!][\![\rho_r]\!]S$ by properties of substitution

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [[\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]C]\,[\![\epsilon]\!][\![\rho_r]\!]\overrightarrow{Pat} \Leftarrow \overset{i}{\Pi} X{:}[\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]U.\,([\![\epsilon]\!][\![\rho_r \circ \rho_1]\!]T) \rangle [\![\epsilon]\!][\![\rho_r]\!]S$ by `t-spi`

$\Delta_i, [\![\epsilon]\!][\![\rho_r]\!]\Delta_1; [\![\epsilon]\!][\![\rho_r]\!]\Gamma_1 \vdash [\![\epsilon]\!][\![\rho_r]\!][[\![\rho_1]\!]C]\,\overrightarrow{Pat} \Leftarrow [\![\epsilon]\!][\![\rho_r \circ \rho_1]\!](\overset{i}{\Pi} X{:}U.\,T) \rangle [\![\epsilon]\!][\![\rho_r]\!]S$ by properties of substitution

which is what we wanted to show

□