Contextual Refinement Types

Antoine Gaulin
McGill University
antoine.gaulin@mail.mcgill.ca

Brigitte Pientka
McGill University
bpientka@cs.mcgill.ca

We develop an extension of the proof environment BELUGA with datasort refinement types and study its impact on mechanized proofs. In particular, we introduce *refinement schemas*, which provide fine-grained classification for the structures of contexts and binders. Refinement schemas are helpful in concisely representing certain proofs that rely on relations between contexts. Our formulation of refinements combines the type checking and sort checking phases into one by viewing typing derivations as outputs of sorting derivations. This allows us to cleanly state and prove the conservativity of our extension.

1 Introduction

Proof mechanization provides strong trust guarantees towards the validity of theorems. Contrary to informal proofs, expressing a theorem and its proof formally requires absolute precision. The resulting statement can thus become riddled with technicalities, which obscures their relation to their informal counterparts. This work combines two approaches to type systems that significantly reduce the added complexity from formalization, namely refinement types and higher-order abstract syntax (HOAS).

Datasort refinement types [10, 9] provide ways to define subtypes (called *datasorts* or just *sorts*) by imposing constraints on the constructors of (inductive) types. Intuitively, a sort S refines a types A if it is defined by a subset of its constructors. The idea originated in the simply-typed setting, where refinements enhance the expressive power of the type system. Later, Lovas and Pfenning [17, 16] extended datasort refinements to the dependently-typed Edinburgh logical framework LF [13]. They provide an equivalence between their system of refinements, LFR, and another extension of LF with proof-irrelevance. An immediate conclusion here is that refinements do not increase the expressive power of dependently-typed calculi. Rather, Lovas observes that refinements may significantly reduce the verbosity of mechanized proofs, which is demonstrated through several case studies [16].

BELUGA is a two-level programming language based on contextual modal type theory (CMTT) [18]. It uses the Edinburgh logical framework LF [13] as a specification logic (data-level), with an intuitionistic first-order reasoning logic (computation-level). The data-level is embedded in the computation-level via a (contextual) box modality similar to the one in the modal logic S4. From a logical point of view, the formula $\Box A$ (read box A) expresses that A is true under no assumptions, i.e. in the empty context. The contextual box modality generalizes this idea to arbitrary contexts, yielding formulas of the form $[\Psi \vdash A]$ expressing that A holds in context Ψ. This allows us to represent LF objects (and types) together with a context in which they are meaningful. To handle this representation, LF contexts are restricted using a notion of *schema* that acts as classifiers of contexts, similarly to how types classify terms. In addition, LF substitutions are first-class objects of BELUGA and they can be used to move objects from one context to another while preserving their meaningfulness. These features allow the expression of an object language (OL) using HOAS [19] and provide several substitution lemmas for free in our mechanizations.

We present BELUGA and its extension with refinement types in Sections 3 and 4, which discuss the data-level and computation-level, respectively. The core of the extension consists of replacing the LF

To appear in EPTCS.

layer of BELUGA with a variation on the LFR system of Lovas and Pfenning [17, 16]. We then lift the refinement relations to the computation-level through straightforward congruence rules and show that the extension is conservative, meaning that every well-sorted program of our extension is well-typed in conventional BELUGA. However, this result only applies if we consider BELUGA as a general-purpose language rather than a proof environment. This is because a BELUGA proof is a recursive function that terminates on every inputs and refinements allow specifying more precise domains. Thus, the types that we obtain from conservativity can extend the domain of a function, leading to undefined behaviour on certain inputs. In this sense, the extension permits interpreting some partial recursive functions as proofs While termination is an important part of our work, we focus here on defining the refinements and leave termination checking for future work. More details on this as well as more examples and a full definition of our system will be available in an upcoming technical report [12]. We also provide with the present paper an electronic appendix containing the definitions of the judgments that we discuss.

2 Motivation

Felty et al. [6] observed that binders can have various structures and that particular results rely only on particular aspects of those structures. This leads to challenges when invoking a lemma in the proof of a theorem since the lemma may rely on simpler binding structures than the theorem. For instance, a lemma using untyped term variables can still be useful for a theorem that uses typed term variables. They propose a series of benchmark challenges along with solutions using multiple contexts and relations between them. We design a new solution based on refinements for one of these benchmarks, namely the equivalence of algorithmic and declarative equalities for the untyped λ-calculus.

The first step of the mechanization is to encode the untyped λ-calculus as an LFR datatype:

```
LFR tm : type =
  | lam : (tm → tm) → tm
  | app : tm → tm → tm;
```

This syntax declares a new type called `tm` whose objects are built from the two given constructors, `lam` and `app`. The constructors encode function abstraction and function application, respectively. Next, we want to encode the judgments for declarative and algorithmic equalities:

```
LFR deq : tm → tm → type =
  | e-lam : ({x : tm} deq x x → deq (M x) (N x)) → deq (lam M) (lam N)
  | e-app : deq M1 N1 → deq M2 N2 → deq (app M1 M2) (app N1 N2)
  | e-refl : {M : tm} deq M M
  | e-sym : deq M N → deq N M
  | e-trans : deq M1 M2 → deq M2 M3 → deq M1 M3;
```

Here, we exploit the dependent types of LFR to express declarative equality as a binary predicate on objects of type `tm`. The constructors `e-refl`, `e-sym`, and `e-trans` encode the axioms of an equivalence relation (reflexivity, symmetry, and transitivity, respectively), while the constructors `e-lam` and `e-app` correspond to congruence rules. Algorithmic equality is just declarative equality without the three equivalence axioms. As such, we define it as a refinement of `deq` rather than as a separate atomic type:

```
LFR aeq ⊏ deq : tm → tm → sort =
  | e-lam : ({x : tm} aeq x x → aeq (M x) (N x)) → aeq (lam M) (lam N)
  | e-app : aeq M1 N1 → aeq M2 N2 → aeq (app M1 M2) (app N1 N2);
```

To declare a new (atomic) sort, users must specify three things: the type which is refined, a refinement kind, and a list of constructors together with their sort. The type must have been previously declared, the sort's kind must refine the type's kind, and each of the constructors' sort must refine their assigned type. By using a sort instead of a type, we get to reuse the same constructors for both judgments. This guarantees that any proof of algorithmic equality can be interpreted as a proof of declarative equality. Thus, we get the soundness of algorithmic equality for free.

The last step in encoding the language is to define its contexts via context schemas. The goal of a schema is to characterize the structure of contexts, which consist of tuples of assumptions rather than being flat lists. A particular context can contain various forms of assumptions (untyped term variables, typed term variables, type variables, etc.) depending on the features of the OL that we are mechanizing. We call these forms of assumptions *worlds* (or *schema elements*). Intuitively, worlds are to schemas what constructors are to atomic types: they specify how to construct a context of a given schema. Our notation for schema declaration emphasizes this idea :

```
schema xdG =
| xeW : block (x : tm, e_x : deq x x);
```

Now, if we have a context Ψ of schema `xdG`, then we can extend it with an additional block variable b with world `xeW`, yielding the context $\Psi, b{:}\texttt{xeW}$. A refinement of schema is then obtained by selecting a subset of the worlds and refinement them to sorts. Here, we want to refine the `deq` assumption to `aeq`, which we do as follows :

```
schema xaG ⊏ xdG =
| xeW : block (x : tm, e_x : aeq x x);
```

One advantage of this approach is that the sort of a block variable `b:xeW` is fully hidden in the schema of the context in which it appears. This means that a given context of schema `xaG` can also be seen as having schema `xdG`. Moreover, this idea generalizes to arbitrary schemas and can go in both directions when all the worlds of the type-level schema also appear in the refinement schema. In this case, given schemas $H \sqsubset G$ and a context $\Psi : H$, we write $\Psi^{\top}$ to indicate that we wish to interpret Ψ as a context of schema G.

In the conventional solution [8], a relation between contexts of the two schemas needs to be maintained explicitly. Here, we can simply use the refinement relation and our special $\Psi^{\top}$ context instead. Let us now look at a few cases of the proof of completeness of algorithmic equality:

```
rec aeq-sym : (Ψ : xaG) [Ψ ⊢ aeq M N] → [Ψ ⊢ aeq N M] = ...;
rec ceq : (Ψ : xaG) [Ψ^⊤ ⊢ deq M N] → [Ψ ⊢ aeq M N] =
fn d => case d of
  | [Ψ ⊢ #b.2] => [Ψ ⊢ #b.2]
  | [Ψ ⊢ e-sym D] =>
        let [Ψ ⊢ D'] = ceq [Ψ^⊤ ⊢ D] in
        aeq-sym [Ψ ⊢ D']
  | [Ψ ⊢ e-lam (λx.λe. D)] =>
        let [Ψ, b:xeW ⊢ E] = ceq [Ψ, b:xeW ⊢ D[.., b.1, b.2]] in
        [Ψ ⊢ e-lam (λx.λe. E[..,<x;e>])]
  | ...
```

where parentheses around the context variable `(Ψ : xaG)` indicate implicit quantification.

The first case is for variables, represented as the second projection on one of the block `b` in Ψ. The symbol `#` is merely a syntactic device to identify `b` as a variable. This case acts in essence just like an

identity function, except that the sort of the output does not match that of the input. When we pattern match, we know that `d` has the context $\Psi^{\top}$ from the sort of `ceq` Since $\Psi^{\top}$ has schema `xdG`, we know the world of `b` and therefore that `b.2` has sort `deq b.1 b.1`. On the other hand, when we produce the output `[Ψ ⊢ #b.2]`, then the sort of `ells us o` interpret Ψ as a `xaG`, so we assign it the sort `aeq b.1 b.1`, as desired.

Next, we have the case of symmetry, which is solved by a recursive call on the subderivation, followed by a call to the relevant lemma `aeq-sym`. We know from the sort of `ceq` that the recursive calls produce objects of sort $[\Psi \vdash \texttt{aeq}\ M\ N]$ with $\Psi : \texttt{xaG}$, which is precisely what the lemma expect.

Finally, the case for λ-abstraction requires extending the context with an additional block of assumptions. Here, we it is evident that the two contexts involved are the same, except that they are interpreted in different ways.

Our solution is simple and closely resembles an informal proof of completeness of algorithmic equality. In contrast, the conventional BELUGA solution[1] requires a total of 13 additional arguments, including 7 explicit ones that must be manipulated in every case of the proof.

3 Data-level

The main objective of BELUGA is to facilitate reasoning about the properties of OLs. To achieve this, an OL is specified using a variant of the Edinburgh logical framework LF [13], called Contextual LF, in which LF objects and types are always represented together with a context in which they are meaningful, that is containing all of its free variables. The variables of an OL are represented as LF variables, which allows reusing LF's substitution calculus to represent substitution in the OL. This kind of representation is known as HOAS and eliminates the need to prove several substitution properties. Contextual LFR applies the same idea to the LFR system of Lovas and Pfenning [17, 16].

We will start by reviewing Lovas and Pfenning's LFR [17] and discuss the numerous changes that we make to their presentation, and then we will see how refinements carry on to Contextual LFR. Unfortunately, the contextual aspect cannot be cleanly separated from LFR since the syntax and judgments of LFR have to be altered during the extension. In particular, all the judgments depend on an additional context, called the *meta-context* and denoted Ω at the refinement level and Δ at the type level. These consist of *meta-variables* which may occur within LFR objects. So, we maintain these aspects in our presentation of LFR, but defer their explanation to the last part of this section.

We follow a canonical form presentation [28] for the data-level. This means that only normal terms are allowed, which requires the use of hereditary substitutions. Simply put, hereditary substitutions are like ordinary substitutions except that they apply any β-reduction that appears during the process. For instance, the substitution $[(\lambda y.y)/x](x\ 0)$ produces 0 instead of $(\lambda y.y)\ 0$.

3.1 LFR

As previously mentioned, LFR extends LF with datasort refinement types. The objects of LFR are exactly the same as in LF and their classifiers are separated in two levels, types A and sorts S, which are related by a refinement relation $S \sqsubset A$. Due to their dependencies on types, the other syntactic categories are similarly duplicated into a type-level and a sort-level related by a refinement relation.

To facilitate the extension to Contextual LFR, it is crucial that we apply this principle to contexts instead of using a single context containing both typing and sorting assumptions like Lovas and Pfenning

[1] Available at `https://github.com/pientka/ORBI`

[17]. A contextual type $(\Gamma.A)$ encodes the LF judgment that A is a well-formed type in (typing) context Γ and a contextual sort $(\Psi.S)$ similarly encodes the LFR judgment that S is a well-formed sort in (sorting) context Ψ. The only sensible way to establish that $(\Psi.S) \sqsubset (\Gamma.A)$ is to show that $\Psi \sqsubset \Gamma$ and that $S \sqsubset A$. This new refinement relation can then be thought of as a relation between a sort-level judgment and a type-level judgment.

3.1.1 Types and sorts

We start by presenting the types and sorts of LFR and discussing the relations between them. Both types and sorts are allowed to depend on (normal) terms M. They are given by the following syntax:

	Type level	Refinement level
Atomic families	$P ::= \mathbf{a} \mid P\ M$	$Q ::= \mathbf{s} \mid Q\ M \mid P$
Canonical families	$A ::= P \mid \Pi x{:}A_1.A_2$	$S ::= Q \mid \Pi x{:}S_1.S_2$

The refinement relation ultimately boils down to what the user specifies. An atomic type family $\mathbf{a}$ is defined by its constructors and their types. An atomic sort family $\mathbf{s} \sqsubset \mathbf{a}$ is then defined by selecting a subset of the constructors of $\mathbf{a}$ and assigning them sorts that refine their previously specified types. In this sense, refinements offer a way to safely reuse constructors. Finally, the relation is lifted to other types with simple congruence rules:

$$\frac{Q \sqsubset P}{Q\ M \sqsubset P\ M} \qquad\qquad \frac{S_1 \sqsubset A_1 \quad S_2 \sqsubset A_2}{\Pi x{:}S_1.S_2 \sqsubset \Pi x{:}A_1.A_2}$$

Due to the presence of dependencies, type and sort well-formedness are non-trivial in LFR. The type well-formedness judgment, $A : \texttt{type}$, coincides exactly with LF's type well-formedness judgment. It essentially just makes sure that whenever we apply an atomic family P to an argument M, then M has the type prescribed by P's kind. We could likewise consider a sort well-formedness judgment $S : \texttt{sort}$ that validates the sorts of dependencies, but instead we consider the refinement relation itself to be the sort well-formedness judgment. This is also what was done by Lovas and Pfenning [17], however the presence of intersection sorts in their system prevents the full separation of a sort well-formedness judgment. This is because intersections are only allowed when both sorts refine the same type, so the refinement information must be present during the well-formedness derivation.

The refinement relation for atomic families $Q \sqsubset P$ is similar to the notion of constructor subtyping [1], according to which a subtyping relation $P_1 \leq P_2$ occurs between two inductive types when P_1 is defined by a subset of the constructors of P_2. As such, it is sensible to consider a notion of subsorting (i.e. subtyping at the level of sorts) such that $Q \leq P$ whenever $Q \sqsubset P$. In particular, a subsumption rule is admissible for refinements of atomic families. The natural subsorting rule for function spaces would be contra-variant in the domain, while refinement of function spaces is co-variant. As such, a subsumption principle for refinements of function spaces is not guaranteed, although it is admissible for the *weak* function spaces of LF.

In addition to the usual typing judgment $M : A$, we have a sorting judgment, denoted $M :: S$. Sorting replicates typing similarly to how sorts replicate types syntactically. A similar phenomenon occurs for all the other LF judgments (context formation, kinding, etc.). One of our key observations is that the type-level judgments can be unified with their sort-level analogues due to their close resemblances. For typing, this yields a judgment $M : S \sqsubset A$ that encompasses both the facts that $M : A$ and $M :: S$. Let us

exemplify this by considering the rules for function applications:

$$\frac{M_1 : \Pi x{:}A_1.A_2 \quad M_2 : A_1}{M_1\ M_2 : [M_2/x]A_2}\ \text{(Typing)} \qquad \frac{M_1 :: \Pi x{:}S_2.S_2 \quad M_2 :: S_2}{M_1\ M_2 :: [M_2/x]S_2}\ \text{(Sorting)}$$

$$\frac{M_1 : \Pi x{:}S_1.S_2 \sqsubset \Pi x{:}A_1.A_2 \quad M_2 : S_1 \sqsubset A_1}{M_1\ M_2 : [M_2/x]S_2 \sqsubset [M_2/x]A_2}\ \text{(Unified)}$$

For the remainder of our presentation, we will focus on this form of unified judgments. In all our judgments, everything that appears to the right of the refinement symbol is considered to be an output. For instance, in $M : S \sqsubset A$, the type A is an output, which amounts to recovering a typing derivation $M : A$ from the sorting derivation $M :: S$. The actual typing rules (see Figure 2) are bi-directional, so we have two unified judgments, one for synthesis and one for checking. This means that we consider neutral terms R and normal terms M, and that we have unified judgments for synthesis ($R \Rightarrow S \sqsubset A$) and checking ($M \Leftarrow S \sqsubset A$). We will discuss this in more details when we introduce terms in 3.1.3.

The other syntactic categories of LFR are similarly duplicated at the refinement level, except for terms which are the same at both levels since they do not contain any type information to refine. Each category is equipped with a refinement relation that is induced by the refinement for types. In all the judgments involving refinements, everything on the right of $\sqsubset$ can be considered as an output of the judgment.

Our presentation differs from that of Lovas and Pfenning [17] in two other ways. First, we use an explicit embedding of types into sorts rather than an ambiguous $\top$ sort that refines every type. This ensures that for any well-formed sort S, we can compute a type A such that $S \sqsubset A$. In turn, this allows us to combine the typing and sorting judgments into a single sorting judgment (see Figure 2). We can then perform type-checking only when it is needed for a sorting derivation, that is when we reach a type embedded into a sort. Moreover, our embedding is at the level of atomic families rather than canonical families. This being said, an embedding of canonical types within canonical sorts is admissible since we can construct a Π-sort from embedded atomic type families. This is also the case for every other syntactic category in BELUGA. Second, we have omitted intersection sorts $S_1 \wedge S_2$, which allow specifying multiple sorts for an object at once.

We note that both subsorting and intersection sorts, although useful features, can significantly slow down sort checking, much like their type-level equivalent would slow down type checking. On the other hand, sorts themselves come at a very low cost while still offering several of the benefits of richer sort systems. The results that we present in 3.1.4 can be extended to a system supporting subsorting and intersection sorts. Adding intersections is straightforward, but subsorting brings complication when it comes to validating coverage since when we pattern match on an object of sort Q, then we need to consider the cases coming from the constructors of Q as usual, but also any additional constructor coming from a subsort of Q.

3.1.2 Contexts and schemas

Next, we take a closer look at LFR contexts and schemas. Again, these are separated into a type-level and a refinement-level that are related by a refinement relation. The syntax of LFR contexts and schemas is as follows:

	Type level	Refinement level
Blocks of declarations	$B ::= \cdot \mid \Sigma x{:}A.B$	$C ::= \cdot \mid \Sigma x{:}S.C$
Schema elements	$E ::= B \mid \Pi x{:}A.E$	$F ::= C \mid \Pi x{:}S.F$
Contexts	$\Gamma ::= \cdot \mid \psi \mid \Gamma, x{:}A \mid \Gamma, b{:}E \cdot \vec{M}$	$\Psi ::= \cdot \mid \psi \mid \Psi, x{:}S \mid \Psi.b{:}F \cdot \vec{M}$
Context schemas	$G ::= \cdot \mid G + E$	$H ::= \cdot \mid H + F$

$\boxed{\Omega \vdash \Psi \sqsubset \Gamma}$ – Refinement relation for contexts

$$\frac{\vdash \Omega \sqsubset \Gamma}{\Omega \vdash \cdot \sqsubset \cdot} \qquad \frac{(\psi : H) \in \Omega}{\Omega \vdash \psi \sqsubset \psi} \qquad \frac{\Omega \vdash \Psi \sqsubset \Gamma \quad \Omega; \Psi \vdash S \sqsubset A}{\Omega \vdash (\Psi, x{:}S) \sqsubset (\Gamma, x{:}A)} \qquad \frac{\Omega \vdash \Psi \sqsubset \Gamma \quad \Omega; \Psi \vdash F \sqsubset E}{\Omega \vdash (\Psi, b{:}F[\vec{M}]) \sqsubset (\Gamma, b{:}E[\vec{M}])}$$

Figure 1: Refinement relations for contexts and schemas

Blocks of declarations represent tuples of labelled assumptions, i.e. variables. The empty block $\cdot$ is not valid on its own, rather it is a syntactic device that indicates the end of a block. Empty blocks are not strictly necessary for the system to work, but facilitate the theoretical development by providing a simple base case.

A schema element is a parameterized block of declarations. While blocks express specific instances of assumptions, schema elements encode the general requirements of a particular form of assumption. For instance, a typing assumption (informally denoted by $x : A$) is characterized by the schema element $\Pi A : \texttt{tp}.\Sigma x{:}\texttt{tm}.\Sigma t{:}\texttt{oft}\ x\ A.\cdot$, while a particular instance of this assumption would be $\Sigma x{:}\texttt{tm}.\Sigma t{:}\texttt{oft}\ x\ \texttt{nat}.\cdot$. Each schema element corresponds to an inference rule for the OL's context formation judgment. A schema is defined as a sum of schema elements and similarly corresponds to the OL's full context formation judgment. Note that in the external syntax used in our example, every schema element was assigned a name and referred to exclusively by that name. Here, we directly use the element's sort to avoid the need for extra premises performing signature lookups in our inference rules.

LFR contexts can contain two kinds of variables. Ordinary variables, denoted by x, stand for an arbitrary LFR object of the specified type. Block variables, denoted by b, stand for tuples of assumptions satisfying the specification of a schema element. In conventional Beluga, block variables are directly assigned with a block of declaration instead of a schema element applied to some objects. Here, we require that these objects be specified explicitly, so that they can be recovered when pattern matching on a context. This simplifies the schema checking rules (see Appendix for a definition) since they no longer rely on unification to establish that a block extension fits a schema element. We also allow a single context variable ψ to appear on the left-most position of LFR contexts Ψ (or Γ), but we do not consider ψ as a variable of Ψ. Instead, ψ is a placeholder for an actual context to be substituted at a later time, so it is stored in the meta-context Ω.

The refinement relations for blocks, schema elements, and schemas are very simple (see Appendix). For schema elements, we just check one sort at a time, starting with the parameters and then the assumptions in the block. Refinements of contexts are similarly checked one assumption at a time (see Figure 1). For block assumptions, we require the same parameters to be used to instantiate the schema elements on both sides of the refinement relation. The relation on schemas is similarly simple, but we take care not to allow duplicate schema elements in G (or in H for that matter). We do this mainly because duplicate elements serve no purpose in practice, but also to highlight the fact that multiple elements of H can refine the same element of G.

Contexts are validated using the schema checking judgments $\Omega \vdash \Psi : H \sqsubset G$ (at the sort-level) and $\Delta \vdash \Gamma : G$ (at the type-level). Assigning a schema H to a context Ψ requires that all the assumptions in Ψ match one of the schema elements in H (see Appendix). This means that all the assumptions in Ψ are of the form $b{:}F[\vec{M}]$, hence there is no rule associated to single variables $x{:}A$. The empty context checks against any well-formed schema. For context extensions, we use an auxiliary judgment $\Omega \vdash \vec{M} : F > D$ that checks the terms in $\vec{M}$ against the parameters of F one at a time. In the end, it produces the block of declarations D obtained by β-reducing $F[\vec{M}]$. Recall that A is an output of the judgment $\Omega; \cdot \vdash M \Leftarrow S \sqsubset A$

and that C is an output of $\Omega \vdash D \sqsubset C$, so we do not need to know them in advance in order to validate the premises.

3.1.3 Terms and substitutions

Now that we have discussed the classifiers of contextual LFR, let us look at the objects that they classify. We distinguish two kinds objects, namely terms and substitutions. Terms are classified by sorts (and types), while substitutions are classified by contexts. Only normal forms are allowed at the data-level, and this is enforced with a canonical form presentation [28]. The syntax is as follows:

Neutral term	$R ::=$	$\mathbf{c} \mid x \mid b.k \mid R\ M$	n-ary tuple	$\vec{M} ::=$	$\cdot \mid M;\vec{M}$
Normal term	$M ::=$	$R \mid u[\sigma] \mid \lambda x.M$	Substitution	$\sigma ::=$	$\cdot \mid \mathtt{id}_\psi \mid \sigma, M \mid \sigma, \vec{M}$

The separation of terms into neutral and normal ensures that no β-reduction can be done by preventing applications of λ-abstractions. The typing rules (see Figure 2) will also guarantee that all terms are η-long. n-ary tuples of normal terms are crucially used in substitutions to replace block variables b. Since b is always used in a projection $b.k$, the substitution $[\vec{M}/b]$ needs to extract the k^{th} projection of the n-ary tuple in order to avoid expressions of the form $\vec{M}.k$, which are undefined by our grammar (and not normal). This coincides nicely with the idea of hereditary substitution and can be added with only minor modifications to their definition. Substitutions can also contain individual terms, which are used to replace individual variables x. Finally, $\mathtt{id}_\psi$ is the identity substitution for the context variable ψ. Note that substitutions have the same structure as their domain, hence it is not specified explicitly.

$\boxed{\Omega;\Psi \vdash M \Leftarrow S \sqsubset A}$ and $\boxed{\Omega;\Psi \vdash R \Rightarrow S \sqsubset A}$ – Bi-directional typing

$$\frac{(u : \Psi'.P) \in \Omega \quad \Omega;\Psi \vdash \sigma : \Psi' \sqsubset \Gamma}{\Omega;\Psi \vdash u[\sigma] \Leftarrow [\sigma]Q \sqsubset [\sigma]P}$$

$$\frac{(b : F[\vec{M}]) \in \Psi \quad \Omega \vdash \vec{M} : F > C \quad \Omega;\Psi \vdash b : C \gg_1^k S \quad \Omega;\Psi \vdash S \sqsubset A}{\Omega;\Psi \vdash b.k \Rightarrow S \sqsubset A}$$

$\boxed{\Omega;\Psi_1 \vdash \sigma : \Psi_2 \sqsubset \Gamma_2}$ – σ is a well-formed substitution from Ψ_2 to Ψ_1

$$\frac{\Omega;\Psi_1 \vdash \sigma : \Psi_2 \sqsubset \Gamma_2 \quad \Omega;\Psi_2 \vdash \vec{M}_2 : F > D \quad \Omega;\cdot \vdash F \sqsubset E \quad \Omega;\Psi_1 \vdash \vec{M}_1 \Leftarrow D}{\Omega;\Psi_1 \vdash (\sigma, \vec{M}_1) : (\Psi_2, b{:}F[\vec{M}_2]) \sqsubset (\Gamma_2, b{:}E[\vec{M}_2])}$$

Figure 2: Bi-directional typing rules

Neutral terms consist of constants $\mathbf{c}$, single LFR variables x, projections of LFR block variables $b.k$, and function applications of neutral terms to normal terms $R\ M$. The sort synthesis rules for constants, single variables, and function applications are standard and coincide with those of Lovas and Pfenning [17]. Blocks of variables b are not valid LFR objects on their own, instead they are always used in projections. To synthesize the sort of a projection $b.k$, we first retrieve its classifying world $F[\vec{M}]$ from the context, then we compute the block D that it corresponds to via the judgment $\Omega \vdash \vec{M} : F > D$, and finally we extract the k^{th} component of D using the auxiliary judgment $\Omega;\Psi \vdash b : C \gg_i^k$. Normal terms are either neutral terms or λ-abstraction, and the sort-checking rules are standard.

Well-formedness of substitutions is validated with the judgment $\Omega;\Psi_1 \vdash \sigma : \Psi_2 \sqsubset \Gamma$, where Ψ_2 is the domain and Ψ_1 the range of σ. The empty substitution has domain $\cdot$ and range any context Ψ_1, so it allows weakening a closed object to any context. Substitution extension with a single term (σ, M) are validated against context extensions with a single variable $(\Psi_2, x{:}S)$ in the usual way. The substitution $\sigma, \vec{M}$ is used to substitute the n-ary tuple $\vec{M}$ for a block variable b.

3.1.4 Meta-theory

Our main result concerning LFR is that for any refinement-level derivation, there is a corresponding type-level derivation. In particular, well-sorted terms are also well-typed, which means that our extension does not provide any new meaningful terms, i.e. that it is conservative. The statement of the theorem relies on type-level LFR judgments that we have not yet discussed due to their similarities with analogous refinement-level judgments and to the fact we did not change them. Let us then first recapitulate the important refinement-level judgments and mention their type-level analogues:

Judgment	Type-level	Refinement-level
Type formation	$\Delta;\Gamma \vdash A \Leftarrow \texttt{type}$	$\Omega;\Psi \vdash S \sqsubset A$
Type checking	$\Delta;\Gamma \vdash M \Leftarrow A$	$\Omega;\Psi \vdash M \Leftarrow S \sqsubset A$
Type synthesis	$\Delta;\Gamma \vdash R \Rightarrow A$	$\Omega;\Psi \vdash R \Rightarrow S \sqsubset A$
Schema checking	$\Delta \vdash \Gamma : G$	$\Omega \vdash \Psi : H \sqsubset G$

The same separation occurs for any other judgment of the system. In particular, every refinement relation that we have introduced corresponds to the type-level formation judgment of the associated syntactic category, like for types. The rules defining a type-level judgment are roughly the same as those defining its refinement-level analogue, except that the refinement-level information is replaced by type-level information. Now, we can formulate the conservativity theorem for LFR as follows:

Theorem 3.1.5 (Conservativity for data-level)

1. *If* $\Omega;\Psi \vdash S \sqsubset A$, *then there are* Δ *and* Γ *such that:*
   (a) $\vdash \Omega \sqsubset \Delta$, *(b)* $\Omega \vdash \Psi \sqsubset \Gamma$, *(c)* $\Delta;\Gamma \vdash A \Leftarrow \texttt{type}$.
2. *If* $\Omega;\Psi \vdash M \Leftarrow S \sqsubset A$, *then there are* Δ *and* Γ *such that:*
   (a) $\vdash \Omega \sqsubset \Delta$, *(b)* $\Omega \vdash \Psi \sqsubset \Gamma$, *(c)* $\Delta;\Gamma \vdash M \Leftarrow A$.
3. *If* $\Omega \vdash \Psi : H \sqsubset G$, *then there are* Δ *and* Ψ *such that:*
   (a) $\vdash \Omega \sqsubset \Delta$, *(b)* $\Omega \vdash \Psi \sqsubset \Gamma$, *(c)* $\Delta \vdash \Gamma : G$.

The proof is discussed in 3.2.1. For now, we simply observe that the close resemblance between type-level and refinement-level judgments, combined with the fact that we can extract type-level derivations from refinement-level ones, suggests that we can lift the refinement relations to the level of LFR judgments. This idea will be reinforced by the refinement relations on contextual types.

3.2 Contextual LFR

The contextual layer (also known as the meta-layer [2]) unifies the different kinds of objects and classifiers of the data-level into unique constructs. This facilitates function abstraction at the computation-level since otherwise each kind of object would need a special kind of function space. As before, our classifiers are separated into types and refinement types. Moreover, since contextual objects include LFR contexts, we naturally obtain a refinement relation for objects as well. The syntax is as follows:

	Type level	Refinement level
Contextual types	$\mathcal{A} ::= \Gamma.P \mid \Gamma.\Gamma' \mid G$	$\mathcal{S} ::= \Psi.Q \mid \Psi.\Psi' \mid H$
Contextual objects	$\mathcal{M} ::= \hat{\Gamma}.R \mid \hat{\Gamma}.\sigma \mid \Gamma$	$\mathcal{N} ::= \hat{\Psi}.R \mid \hat{\Psi}.\sigma \mid \Psi$
Meta-contexts	$\Delta ::= \cdot \mid \Delta, X{:}\mathcal{A}$	$\Omega ::= \cdot \mid \Omega, X{:}\mathcal{S}$
Meta-substitutions	$\rho ::= \cdot \mid \rho, \mathcal{M}$	$\theta ::= \cdot \mid \theta, \mathcal{N}$
Meta-variables	$X ::= u \mid \psi$	

Contexts with hats ($\hat{\Gamma}, \hat{\Psi}$) are called *erased* and contain no type or sort information, so they consist only of variables. Erased contexts are sufficient in this setting since the LF neutral objects and LF substitution do not refer to any type or sort information present in the context. Note that if $\Psi \sqsubset \Gamma$, then $\hat{\Psi} = \hat{\Gamma}$. This means that if $\mathcal{N} \sqsubset \mathcal{M}$ are not just contexts, then $\mathcal{N} = \mathcal{M}$. Accordingly, refinements of meta-objects only provides information when contexts are used as objects.

Note that we only allow atomic LFR sorts to occur in the contextual sort $\Psi.Q$ (and similarly at the type-level). This is not a real limitation of the system since the sort $\Psi.\Pi x{:}S_1.S_2$ would be isomorphic to $(\Psi, x{:}S_1).S_2$. Similarly, we allow only neutral terms in contextual objects $\hat{\Psi}.R$, but this does not impact the expressiveness.

A meta-context can contain two kinds of variables, each associated with one of the three possible forms of contextual types. The meta-variable u stands for an LFR term, so it is given the contextual sort $\Psi.Q$. Meta-variables must be associated with an LFR substitution σ before being used in LFR terms as $u[\sigma]$. In this setting, σ is delayed until a meta-substitution is applied to replace u. The other form of meta-variables are context variables ψ, which are assigned a context schema H. Context variables are used to quantify over contexts at the computation-level. The system can also be extended with support for substitution variables [21, 3].

A meta-substitution is similar to an ordinary substitution, except that it substitutes contextual objects for contextual variables. We denote the application of a meta-substitution using double square brackets. For instance, $[\![\theta]\!]M$ applies the meta-substitution θ to the LFR normal term M. A full definition of this operation was given by Cave and Pientka [3].

The refinement relation for contextual types is obtained by lifting the corresponding refinement relations (developed previously). Similarly, the refinement relation for meta-objects is obtained by lifting the refinement relation on LFR contexts. For example, the natural refinement rule for $\Psi.Q$ is the following:

$$\frac{\Omega \vdash \Psi \sqsubset \Gamma \quad \Omega; \Psi \vdash Q \sqsubset P}{\Omega \vdash \Psi.Q \sqsubset \Gamma.P}$$

In a sense, this raises the refinement relation to the level of LFR judgments. It is helpful to pursue this idea further by formulating our rules for the meta- and computation-level as a refinement relation between judgments. The above rule would then become:

$$\frac{\vdash \Omega \sqsubset \Delta \quad \Omega \vdash \Psi \sqsubset \Gamma \quad \Omega; \Psi \vdash Q \sqsubset P}{(\Omega \vdash \Psi.Q) \sqsubset (\Delta \vdash \Gamma.P)}$$

This judgment $(\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A})$ can then serve as both a type well-formedness and a refinement (i.e. sort well-formedness) judgment. We can similarly unify the sorting and typing judgments, the context well-formedness and refinement judgments, and so on (see Appendix). In all of these judgments, the type-level part (everything on the right of $\sqsubset$) can be taken independently for the rest, in which case it defines the usual judgment of conventional BELUGA. On the other hand, we can also consider the type-level part to be an output, which highlights the fact that type-level judgments don't need to be validated prior to their sort-level analogues.

3.2.1 Meta-theory

The new refinement relation between judgments also facilitates formulating our conservativity result since we have all the type information available by assumption:

Theorem 3.2.2 (Conservativity for contextual layer)

1. *If* $(\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A})$, *then* $\Delta \vdash \mathcal{A}$.
2. *If* $(\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A})$, *then* $\Delta \vdash \mathcal{M} : \mathcal{A}$.
3. *If* $(\Omega_1 \vdash \theta : \Omega_2) \sqsubset (\Delta_1 \vdash \rho : \Delta_2)$, *then* $\Delta_1 \vdash \rho : \Delta_2$.

Conservativity for the contextual layer is proven simultaneously with conservativity for LFR due to inter-dependencies between the two. The proof is a straightforward induction on the given derivation.

An important advantage of our formulation of refinements as a relation on judgments is that it eliminates the need for several lemmas, in particular substitution properties. We note that to obtain the full benefits of the approach, we must also formulate the refinements for LF in this style, as otherwise the lemmas are still needed for conservativity of LFR.

4 Computation-level

BELUGA's computation-level is an ML-style functional programming language with support for pattern matching over contextual objects. It features an indexed function space, so that types are allowed to depend only on data-level objects. Contextual objects and types are embedded in the computation-level via a box modality.

4.1 Computation-level refinements

In our extension, the computation-level is separated into a type layer and a refinement layer, just like the data-level. Since contextual objects can occur in computation-level expression, we maintain a refinement relation for expressions in addition to all other syntactic categories. Our presentation is inspired by the one of Pientka and Abel [22], but differs in two important ways. First, we do not consider recursion since it complicates the syntax of patterns significantly. Specifically, valid recursive calls have to be specified as part of every pattern (although they can be inferred, so users don't need to provide them explicitly). Without recursion, patterns are just (boxed) contextual objects. Second, our sorting (and typing) rules do not require coverage for pattern matching. The syntax of the computation-level is the following:

	Type level	Refinement level	
Types	τ	$\zeta ::=$	$[\mathcal{S}] \mid \zeta_1 \to \zeta_2 \mid \Pi X{:}\mathcal{S}.\zeta$
Contexts	Ξ	$\Phi ::=$	$\cdot \mid \Phi, y{:}\zeta$
Expressions	e	$f ::=$	$[\mathcal{N}] \mid \texttt{fn}\ y{:}\zeta \Rightarrow e \mid e_1\ e_2 \mid \texttt{mlam}\ X{:}\mathcal{S} \Rightarrow e \mid e\ \mathcal{N}$
			$\mid \texttt{let}\ [X] = e_1\ \texttt{in}\ e_2 \mid \texttt{case}^{\zeta}\ [\mathcal{N}]\ \texttt{of}\ \vec{c}$
Branches	b	$c ::=$	$\Omega; [\mathcal{N}] \mapsto e$

Meta-types, -sorts, and -objects are embedded into the computation level via a (contextual) box modality, denote $[\mathcal{S}]$ (for sorts). The elimination form for the modality is given by the `let` expressions: an expression $e_1 : [\mathcal{S}]$ is unboxed as the meta-variable X, which may then be used in the expression e_2.

We distinguish two kinds of function spaces, the simple function space $\zeta_1 \to \zeta_2$ and the dependent function space $\Pi X{:}\mathcal{S}.\zeta$. So, dependencies are restricted to objects from the index domain, which provides strong reasoning power over the index domain without all the difficulties of full dependent types.

The language also supports pattern matching on meta-objects through the use of `case` expressions. While we do not allow pattern matching on arbitrary expressions, any expression that has a box sort can be matched against by first unboxing it with a `let` expression and then matching on the new variable. The sort superscript ζ in `case` expression corresponds to the sort invariant that must be satisfied by all the branches in $\vec{c}$. We require that invariants have the form $\Pi\Omega_1.\Pi X_0 : \mathcal{S}_0.\zeta_0$. Intuitively, a branch $\Omega;[\mathcal{N}] \mapsto e$ satisfies the invariant $\Pi\Omega_1.\Pi X_0{:}\mathcal{S}_0.\zeta_0$ if $\mathcal{N}$ has sort $\mathcal{S}_0$ and e has sort $[\![\mathcal{N}/X_0]\!]\zeta_0$, where $\mathcal{N}$, e, and their sorts can depend on Ω.

The judgments for the computation-level have a similar structure as those for contextual LFR. In particular, type-level and refinement-level judgments are performed simultaneously, with the type-level judgment seen as an output of the simultaneous judgment. Since the derivations produced on both sides of the refinement relation are almost exactly the same, we give the rules with only the refinement part. For instance, sorting and typing is expressed as $(\Omega;\Phi \vdash f : \zeta) \sqsubset (\Delta;\Xi \vdash e : \tau)$, but we define only $\Omega;\Phi \vdash f : \zeta$ for conciseness. We focus here on the rules related to pattern matching. The remaining rules are standard and can be found in the appendix. The rule for `case`-expressions is the following:

$$\frac{\zeta = \Pi\Omega_0.\Pi X_0{:}\mathcal{S}_0.\zeta_0 \quad \Omega \vdash \rho : \Omega_0 \quad \Omega \vdash \mathcal{N} : [\![\rho]\!]\mathcal{S}_0 \quad \Omega;\Phi \vdash c : \zeta \text{ (for all } c \in \vec{c})}{\Omega;\Phi \vdash (\texttt{case}^{\zeta}\ [\mathcal{N}]\ \texttt{of}\ \vec{c}) : [\![\rho,\mathcal{N}/X_0]\!]\zeta_0}$$

The important part of this rule is the last premise, which requires validating that every branch satisfies the given invariant. This is achieved with the judgment $\Omega;\Phi \vdash c : \zeta$ defined by the following rule:

$$\frac{\Omega_0 \vdash \mathcal{N}_0 : \mathcal{S}_0 \quad \Omega,\Omega_0 \vdash \mathcal{S} \doteq \mathcal{S}_0/(\rho,\Omega') \quad \Omega';[\![\rho]\!]\Phi \vdash [\![\rho]\!]f : [\![\rho]\!]\zeta_0}{\Omega;\Phi \vdash (\Omega_0;[\mathcal{N}_0] \mapsto f) : \Pi\Omega_1.\Pi X_0{:}\mathcal{S}_0.\zeta_0}$$

where the judgment $\Omega \vdash \mathcal{S} \doteq \mathcal{S}'/(\rho,\Omega')$ denotes (meta-type) unification. The main difficulty of unification is unifying the dependencies on LF(R) terms. Since we have not modified terms in our extension, the unification algorithm of Pientka and Pfenning [23] still applies.

The conservativity results discussed in section 3.2.1 carry over to the computation-level via straightforward inductions. In particular, every sorting derivation has an analogous typing derivation.

5 Related work

5.1 Refinement types

Various forms of refinement types have been used to solve various problems. Our work is inspired by the datasort tradition that was initiated by Freeman and Pfenning [10, 9] for MINIML, a monomorphic fragment of STANDARD ML's core language. Their system uses refinement type inference so that users don't need to provide annotations, but the inferred sorts are often intersections with some undesired components.

Davies [4], who coined the term datasort, extended this work to the full STANDARD ML language (including modules). They ditched sort inference in favour of a bi-directional sort-checking algorithm, so that only the desired sorts are used by the compiler. Unfortunately, even sort-checking is untenable in the presence of intersection (at least in theory). The compiler needs to choose the correct branch of an intersection when synthesizing sorts for neutral expressions, making sort-checking PSPACE-hard [24].

Jones and Ramsay [15] used refinement types to validate termination of functional programs in the presence of non-exhaustive pattern matching. Their notion of an *intensional* refinement is obtained by removing some of the constructors from a datatype, but the remaining constructors cannot be assigned

new sorts. Instead, a constructor $\mathbf{c} : A$ selected for the sort $\mathbf{s} \sqsubset \mathbf{a}$ is assigned the sort S obtained by replacing every occurrence of $\mathbf{a}$ in A by $\mathbf{s}$. Intensional refinements are weaker than datasorts, but they have full type and refinement inference for a polymorphic ML-style language with algebraic datatype. Their main ideas would also be sufficient to encode our example from section 2.

Another important (and perhaps more common) approach is index refinements, first introduced by Xi and Pfenning [29, 30] for the core language of STANDARD ML. They design a family of dependently-typed ML-style languages parameterized by an arbitrary index domain C, called DML(C). Refinements are obtained by allowing quantification over the index domain, which intuitively corresponds to having a refinement relation $\Pi x{:}S.A \sqsubset A$, where $S \in C$. In this way, most difficulties of dependent types can be avoided, similarly to how we avoid them in BELUGA's computation-level. Datasort refinements and index refinements were combined by by Dunfield [5], yielding an extension of DML with intersections.

An important development of this approach came in the form of logically qualified (or liquid) types [25], this time as an extension of OCAML. In this methodology, a refinement is expressed as $\{x : \tau \mid P(x)\}$, where τ is a type and P is a boolean-valued function over τ. The type τ can then be seen as the refinement $\{x : \tau \mid \texttt{true}\}$ and this allows combining typing and sorting into one judgment, much like we have done for datasort refinements.

To our knowledge, there is currently no work on index refinements for dependently-typed languages. A series of papers culminated in the lax logical framework with side conditions LLF$_{\mathcal{P}}$ [14], which contains types similar in spirit to liquid types. However, LLF$_{\mathcal{P}}$ uses a notion of *lock* types that is based on monads instead of refinements. This being said, side conditions in LLF$_{\mathcal{P}}$ can be interpreted as proof irrelevance, which coincides with Lovas and Pfenning's interpretation of sorts in LFR as proof irrelevance [17]. LLF$_{\mathcal{P}}$ allows more side conditions than LFR, but also puts a heavier proof burden on the user.

5.2 Proof environments

Although HOAS representations offer undeniable benefits, there remains few proof environments that support it. TWELF [20] uses it in its implementation of LF, but all contexts are implicit and this prevents representing context relations [8]. Context schemas emerged from work on TWELF [26], although their notion is more restrictive than ours. The judgments have to be indexed by the unique schema to which the ambient context is known to adhere and there is no way to consider two derivations with different schemas simultaneously.

HYBRID [7] only partially achieves the goals of HOAS: contexts and substitution are inherited from the meta-language, but substitution properties in the OL still need to be proven by hand. However, it has the advantage of being built in well-known logics from which it naturally inherits consistency.

ABELLA [11] does provide the full power of HOAS (called λ-tree syntax in their terminology). Variables are represented in the specification logic through the use of the ∇-quantifier (pronounced nabla). Contexts are represented as lists of ∇-quantified variables and schemas as types depending on these lists. ADELFA [27] is inspired by ABELLA, but uses a specification language that is more closely related to LF. It also improves ABELLA with a built-in notion of schemas.

6 Conclusion

We have developed an extension of Beluga with datasort refinement types. While datasort refinements are mainly used to provide subtyping and intersection types to a language, refinements in the setting

of Beluga offers the potential to express proofs more succinctly. Our extension mainly focused on the notion of refinement schemas, which allow extracting more precise information about contexts, much like refinements extract more precise properties (than types) about objects. In particular, refinement schemas are useful to deal with a special kind of context relations, namely those when the assumptions in two contexts are related by refinements.

6.1 Future work

There are a number of avenues that we plan to explore in the future. First, refinements allow validating the correctness of functions containing non-exhaustive pattern matching thereby supporting modular proof development. A natural next step is therefore to develop a coverage and termination checker for Beluga with refinement types. Due to the close similarity between types and their refinements, it is reasonable to expect that we can adapt [22]. However, challenges emerge from the fact that objects can have multiple sorts (but only one type), especially when it comes to unification. This will in particular impact how we check for coverage.

Second, the refinement relations that we described for schemas and schema elements are limited by the fact that our meta-theory requires sorts to refine only one type. In particular, the relation $D \sqsubset C$ for blocks requires that D and C have the same length. A more interesting relation would allow D to contain more assumptions than C. This is reminiscent of the subtyping principle that adding fields to a record type produces a subtype. Another limitation is that $H \sqsubset G$ can only be established when every element of G is refined by some element of H. Allowing more elements to appear in G would also be meaningful, especially given that every context in G is also in $G + E$ for any element E. With these two modifications, we could represent another class of context relations that Felty et al. [6] calls *linear extensions*. Consequently, our next goal is to provide a more flexible form of refinements and to modify our proof of conservativity so that it no longer relies on type uniqueness for refinements.

References

[1] G. Barthe & M. J. Frade (1999): *Constructor Subtyping*. In S. D. Swierstra, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 109–127, doi:10.1007/3-540-49099-X_8.

[2] A. Cave & B. Pientka (2012): *Programming with binders and indexed data-types*. In J. Field & M. Hicks, editors: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, ACM, pp. 413–424, doi:10.1145/2103656.2103705.

[3] A. Cave & B. Pientka (2013): *First-class substitutions in contextual type theory*. In A. Momigliano, B. Pientka & R. Pollack, editors: *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks & Meta-languages: Theory & Practice, LFMTP 2013, Boston, Massachusetts, USA, September 23, 2013*, ACM, pp. 15–24, doi:10.1145/2503887.2503889.

[4] R. Davies (2005): *Practical Refinement-Type Checking*. Ph.D. thesis, Carnegie Mellon University, USA. AAI3168521.

[5] J. Dunfield (2007): *A Unified System of Type Refinements*. Ph.D. thesis, Carnegie Mellon University. CMU-CS-07-129.

[6] A. P. Felty, A. M. & B. Pientka (2015): *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 1-A Common Infrastructure for Benchmarks*. *CoRR* abs/1503.06095. arXiv:1503.06095.

[7] A. P. Felty & A. Momigliano (2012): *Hybrid - A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax*. *J. Autom. Reason.* 48(1), pp. 43–105, doi:10.1007/s10817-010-9194-x.

[8] A. P. Felty, A. Momigliano & B. Pientka (2015): *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 2-A Survey*. *J. Autom. Reason.* 55(4), pp. 307–372, doi:10.1007/s10817-015-9327-3.

[9] T. S. Freeman (1994): *Refinement Types for ML*. Ph.D. thesis, Carnegie Mellon University, USA, USA. UMI Order No. GAX94-19722.

[10] T. S. Freeman & F. Pfenning (1991): *Refinement Types for ML*. In D. S. Wise, editor: *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, ACM, pp. 268–277, doi:10.1145/113445.113468.

[11] A. Gacek (2008): *The Abella Interactive Theorem Prover (System Description)*. In A. Armando, P. Baumgartner & G. Dowek, editors: *Automated Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 154–161, doi:10.1007/978-3-540-71070-7_13.

[12] A. Gaulin (2023): *Contextutual refinement types*. Master's thesis, McGill University. (Forthcoming).

[13] R. Harper, F. Honsell & G. D. Plotkin (1993): *A Framework for Defining Logics*. *J. ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[14] F. Honsell, L. Liquori, P. Maksimovic & I. Scagnetto (2017): $\mathsf{LLF}_{\mathcal{P}}$*: a logical framework for modeling external evidence, side conditions, and proof irrelevance using monads*. *Logical Methods in Computer Science* Volume 13, Issue 3, doi:10.23638/LMCS-13(3:2)2017.

[15] E. Jones & S. Ramsay (2021): *Intensional Datatype Refinement: With Application to Scalable Verification of Pattern-Match Safety*. *Proc. ACM Program. Lang.* 5(POPL), doi:10.1145/3434336.

[16] W. Lovas (2010): *Refinement Types for Logical Frameworks*. Ph.D. thesis, Carnegie Mellon University, USA.

[17] W. Lovas & F. Pfenning (2010): *Refinement Types for Logical Frameworks and Their Interpretation as Proof Irrelevance*. *Log. Methods Comput. Sci.* 6(4), doi:10.2168/LMCS-6(4:5)2010.

[18] A. Nanevski, F. Pfenning & B. Pientka (2008): *Contextual modal type theory*. *ACM Trans. Comput. Log.* 9(3), pp. 23:1–23:49, doi:10.1145/1352582.1352591.

[19] F. Pfenning & C. Elliott (1988): *Higher-Order Abstract Syntax*. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, Association for Computing Machinery, New York, NY, USA, p. 199–208, doi:10.1145/53990.54010.

[20] F. Pfenning & C. Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems*. In: *Automated Deduction — CADE-16*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[21] B. Pientka (2008): *A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions*. In G. C. Necula & P. Wadler, editors: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, ACM, pp. 371–382, doi:10.1145/1328438.1328483.

[22] B. Pientka & A. Abel (2015): *Well-Founded Recursion over Contextual Objects*. In T. Altenkirch, editor: *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, *LIPIcs* 38, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 273–287, doi:10.4230/LIPIcs.TLCA.2015.273.

[23] B. Pientka & F. Pfenning (2003): *Optimizing Higher-Order Pattern Unification*. In F. Baader, editor: *Automated Deduction – CADE-19*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 473–487, doi:10.1007/978-3-540-45085-6_40.

[24] J. C. Reynolds (1997): *Design of the Programming Language Forsythe*, pp. 173–233. Birkhäuser Boston, Boston, MA, doi:10.1007/978-1-4612-4118-8_9.

[25] P.M. Rondon, M. Kawaguci & R. Jhala (2008): *Liquid Types*. *SIGPLAN Not.* 43(6), p. 159–169, doi:10.1145/1379022.1375602.

[26] C. E. Schürmann (2000): *Automating the Meta Theory of Deductive Systems*. Ph.D. thesis, Carnegie Mellon University, USA. AAI9986626.

[27] M. Southern & G. Nadathur (2021): *Adelfa: A System for Reasoning about LF Specifications*. In E. Pimentel & E. Tassi, editors: *Proceedings of the Sixteenth Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2021, Pittsburgh, USA, 16th July 2021*, *EPTCS* 337, pp. 104–120, doi:10.4204/EPTCS.337.8.

[28] K. Watkins, I. Cervesato, F. Pfenning & D. Walker (2002): *A concurrent logical framework I: Judgments and properties*. Technical Report CMU-CS-02-101, Carnegie Mellon University.

[29] H. Xi (1998): *Dependent Types in Practical Programming*. Ph.D. thesis, Carnegie Mellon University, USA. AAI9918624.

[30] H. Xi & F. Pfenning (1999): *Dependent Types in Practical Programming*. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, Association for Computing Machinery, New York, NY, USA, p. 214–227, doi:10.1145/292540.292560.