Beluga Reference Guide

1 Lexical Conventions

1.1 Reserved Characters

The following characters are reserved and cannot be used anywhere in ordinary identifiers.

.	period	%	percent sign	( )	parentheses
,	comma	|	vertical bar	[ ]	brackets
:	colon	”	quotation mark	{ }	braces
;	semicolon	\	backslash	<>	chevrons

1.2 Names and Identifiers

The following characters can be used in addition to letters and integers anywhere in names and identifiers.

!	exclamation mark	$	dollar sign	&	ampersand
’	apostrophe	*	asterisk	+	addition
-	hyphen	/	forward slash	=	equality
@	at sign	ˆ	caret	_	underscore
˜	tilde				

Note that while the number sign (#) is not a reserved character strictly speaking and may be used in identifiers, it cannot be the first character.

1.3 Keywords

The following are Beluga keywords and cannot be used otherwise.

..	->	<-	=>	==	::
FN	and	block	Bool	case	fn
else	if	impossible	in	let	mlam
of	rec	schema	some	then	type
module	ttrue	ffalse	?	struct	

1.4 Pragmas

Pragmas provide a means of setting compilation behavior within a program. Each begins with the percent sign `%` to indicate that it does not accept parsing.

`--coverage`	Initiate coverage check
`--warncoverage`	Enable coverage check warnings
`--nostrengthen`	Disable automatic meta-variable strengthening
`--infix`	Create an infix operator from a two-argument constructor
`--prefix`	Create a prefix operator from a two-argument constructor
`--assoc`	Set the default associativity
`--name`	Set name preference
`--abbrev`	Set module abbreviation
`--not`	Guard an expression from type-checking
`--open`	Open a module

1.5 Comments

Beluga supports single-line and multi-line comments using `%` and `%{ ... }%`, respectively. Both can contain any character.

`%` *this is a single-line comment*	`%{` *this is a multi-line comment* `}%`

1.6 Command Options

The Beluga executable accepts the following options from the command line.

+annot	Creates an annotation
+d	Enables debugging printing
+ext	Print external syntax before reconstruction
+s=debruijn	Print substitution in deBruijn style
+implicit	Print implicit arguments
+t	Print timing information
+tfile	Print timing information into time.txt
+printSubord	Print subordination relations
-print	Disable printing
-logic	Disable the logic engine
-width nnn	Set output width to nnn (default 86, min 40)
+realNames	Print holes using original names
+HTML	Enable HTML mode
+HTML -css	Enable HTML mode without CSS
+HTML +cssfile <filepath>	Specify a CSS page for HTML mode
+latex	Enables LaTeX mode

2 Grammar

Beluga is a two-level system:

- the LF logical framework level, supporting specification of formal systems

- the computation-level, supporting programming with LF specifications

This section describes both levels in EBNF[1] metasyntax. In particular:

- $\{a\}$ represents repeat production a zero or more times
- $[a]$ represents optionally apply production a
- ($*$ and $*$) enclose comments in the grammar

```
sig ::= { lf_decl          (* LF constant declaraction *)
        | c_decl }         (* Computation-level declaration *)
```

id refers to identifiers starting with a lower case letter and *ID* identifiers starting with an upper case letter.

2.1 LF-level

Beluga's first-level implements the logical framework LF for defining logics and representing proofs and derivations using higher order abstract syntax (HOAS). Specifically, a formal system is represented as a ***signature***, a series of declarations of type families and the constants which inhabit them. The signature establishes the system's syntax, judgments, and inference rules.

There are two ways of making LF declarations. The first follows closely the syntax used by the Twelf system whereas the second is akin to Standard ML datatype declarations.

```
lf_decl ::= lf_datatype_decl    (* new syntax for LF signatures *)
          | id ":" lf_type "."  (* Twelf style LF signatures *)
          | id ":" lf_kind "."  (* Twelf style LF signatures *)
          | lf_decl --name id ID "."    (* name preference pragma *)
          | lf_decl --infix id assoc int "."    (* infix pragma *)
          | lf_decl --prefix id int "."         (* prefix pragma *)

infix_prg ::=  lf_decl --infix id [assoc] nat "."

assoc ::= none
   | left
   | right

op_arrow ::= "->" | "<-"        (* A <- B same as B -> A *)

lf_kind ::= type
```

[1]Extended BackusNaur Form: a family of metasyntax notations to describe formal languages

```
          | lf_type op_arrow lf_kind
          | "{" id ":" lf_type "}" lf_kind

lf_type ::= id {term}                       (* A M1 ... Mn *)
          | lf_type op_arrow lf_type
          | "{" id ":" lf_type "}" lf_type  (* Pi x:A. K  or  Pi x:A. B *)

lf_datatype_decl ::= datatype id ":" lf_kind "=" [lf_con_decl] {"|" lf_con_decl} ";"

lf_con_decl ::= id ":" lf_type

term ::= (id | ID)         (* variable x or constant a or c *)
       | "\" id "." term   (* lambda x. M *)
       | term term         (* M N *)
       | _                 (* hole, inferred by term reconstruction *)
```

The constructs `{x:U} V` and `x |- V` bind the identifier `x` in `V` which may shadow other constants or bound variables. As usual in type theory, `U -> V` is treated as an abbreviation for `{x:U}V` where `x` does not appear in `V`.

In the order of precedence, we disambiguate the syntax and impose restrictions as follows:

- Juxtaposition (application) is left associative and has highest precedence
- −> is right and <− left associative with equal precedence
- : is left associative
- {} and \ are weak prefix operators
- Bound variables and constants must be written with lower-case letters
- Free variables can be written with upper-case letters or lower case letters; but we use the convention to write them with upper-case letters
- All terms need to be written in beta-normal form, i.e. no redeces can occur in terms. However, variables do not need to be eta-expanded.

The following examples illustrate which terms Beluga will parse. For example, the following are parsed identically using the old style LF syntax:

```
d : a <- b <- {x:term} c x -> p x.
d : ({x:term} c x -> p x) -> b -> a.
d : ((a <- b) <- ({x: term} ((c x) -> (p x)))).
```

The following declarations using old-style LF syntax parses:

```
d : p x <- a <- b <-  c x.  % against our convention but parses
d : p X <- a <- b <-  c X.  % Better code: uses convention

P:term -> type              % Parse error
```

The following in the old-style LF syntax will not parse:

```
d: p ((\x.x) a) .    % this will give an error since there is a redex
d: p a.              % this will parse.
```

We advocate using a new style for LF declarations based on datatypes.

```
datatype foo: {x:A} p x -> type =
  | d : a -> b -> foo c
  | f : d -> e -> foo k
;
```

Other differences between the Twelf system include:

- Lambda-abstractions cannot be annotated with the type of the input. Writing `x:nat.x` is illegal in Beluga.
- Omitting the type in a Pi-type is illegal in Beluga. One cannot simply write `({x}foo x T) -> foo E T'.` . One must give the type of the variable `x`.

2.2 Computation-level

The computation-level provides a dependently typed functional language to manipulate and analyze object-level data. A term is passed along with its context as a ***contextual object***, enclosed between brackets `[ ]` to indicate that no further computations are required to reach a closed value. ***Contextual variables*** and ***projections*** on bound variables embed the term into computations. Context dependency is captured by ***contextual types***, taken as base types. Contexts themselves are classified by ***schemas*** which are analogous to types with respect to terms. ***Inductive datatypes*** define context relationships such as strengthening and weakening.

Beluga leverages higher order syntax to support:

- tuples `(T1 * T2)` (introduction for product types)
- function types `T1 -> T2`
- universal quantification ($\forall$ `x :  T`)
- nameless function (`fn x => e`) (introduction for function types)
- abstraction (λ `X => e`) (introduction for universal types)

- recursion (`rec f => e`)
- case-analysis

Unlike Twelf, Beluga permits nesting of quantifiers and implications and supports higher-order functions.

```
c_decl ::= ctx_schema_decl      (* schema declaration *)
         | c_datatype_decl      (* inductive datatypes *)
         | c_typedef            (* type abbreviation *)
         | c_rec_prog           (* recursive programs *)
         | c_let_prog           (* non-recursive programs *)
         | c_module             (* module *)
         | exp ";"              (* computation-level expression *)

c_typedef  ::= typedef id ":" c_kind "=" c_type ";"

c_rec_prog ::= rec id ":" c_type "=" exp {and id ":" c_type "=" exp} ";"

c_let_prog ::= let id [":" c_type] "=" exp ";"

c_module ::= module ID "=" struct sig end ";"

% Schema
ctx_schema_decl ::= schema [some "[" ctx "]"] block
                                  | schema lf_type

hyp ::= id ":" lf_type
                     | block

% Context
ctx ::= [hyp {"," hyp}]
      | id ["," hyp {"," hyp}]  (* Context variable *)

ctx_hat ::= [id {"," id}]           (* A context with the types erased. *)

c_datatype_decl ::=
         datatype id ":" c_kind "=" [ID : c_type] {"|" ID : c_type}
         {and id ":" c_kind "=" [ID : c_type] {"|" ID : c_type}} ";"

exp    ::= "[" ctx_hat "|-" term "]"      (* Contextual object *)
         | "[" ctx "|-" term "]"          (* Type annotated contextual object *)
         | "(" exp "," exp ")"            (* Tuple *)
         | id                             (* Variable *)
         | ID                            (* Computation-level constructor *)
         | fn id "=>" exp                (* Computation-level function *)
         | mlam ID "=>" exp              (* Dependent type abstraction *)
         | mlam id "=>" exp              (* Context abstraction *)
         | exp exp                       (* Application *)
         | exp "[" ctx "]"               (* Context application *)
         | case exp of branch {branch}   (* Case analysis *)
         | case exp of "{" "}"           (* Case analysis at uninhabited type *)
         | let patt = exp in exp         (* Case expression with one branch *)


case_anl ::= case exp of branch {branch}    (* Case analysis *)
           | case exp of "{" "}"            (* Case analysis at uninhabited type *)
           | impossible exp                 (* Case analysis at uninhabited type *)

branch ::= {quantif} patt "=>" exp

patt ::= "[" ctx_hat "|-" term "]"       (* Contextual object *)
       | "[" ctx "|-" term "]"           (* Type annotated contextual object *)
       | "(" patt  "," patt ")"          (* Tuple *)
       | id                              (* Pattern variable *)
       | ID {index_obj}                  (* Computation-level constructor *)
       | patt ":" c_type                 (* Type annotation in pattern *)

term, lf_type ::= cvar                (* contextual variable *)
                | id "." int          (* k-th projection of a bound variable x *)
                | id "." id           (* projection with specified name of x *)
                | block               (* Sigma x:A.B *)


block ::= block id ":" lf_type {"," id ":" lf_type}

cvar ::= id                    (* context variable *)
       | ID subst              (* meta-variable *)
       | "#" id subst          (* parameter variable *)
       | "#" id "." int subst  (* k-th projection *)
       | "#" id "." id subst   (* projection with specified name *)

subst ::=                      (* nothing *)
       | "[" sigma "]"         (* explicit substitution *)


sigma ::= ^                   (* empty substitution *)
```

```
| ..                   (* Identity substitution *)
| sigma, term           (* substitution sigma , M *)
```

Note that Beluga restricts context variables to always occur on their own (i.e. sigma is always empty). Moreover, if `sigma` is the identity substitution, then it may be omitted and reconstruction will infer it, i.e. writing `[g |- X[..] ]` is equivalent to writing `[g |- X]`.

In order of precedence, Beluga disambiguates the syntax and imposes restrictions as follows.

- Juxtaposition (application) is left associative and has highest precedence
- Bound variables and constants must be written with lower-case letters
- Free meta-variables must be written with upper-case letters
- All LF terms need to be written in β-normal form, i.e. no redeces can occur in terms. *LF terms occurring as part of a contextual object must be written in eta-expanded form.* For example, one must write `[g |- lam \x.E]` which is equivalent to `[g |- lam \x.E[..,x] ]` whereas `[g |- lam E]` is currently rejected.

3 Syntax

Recall that Beluga is a two-level system. The LF-level comprises the object layer or data layer. The computation-level makes up Beluga's functional programming language for reasoning about LF data.

3.1 LF Declarations

The LF signature defines a deductive system at the object level with constants declared using higher-order abstract syntax. Data is defined via data-level types, introduced with the `type` keyword. Constructors result in a particular base type. Data-level types and constructors are analogous to kinds and types in LF type theory. Dependent data-level types which take arguments correspond to type judgments whose constructors are inference rules.

All LF constants take lower-case identifiers. Meta-variables within constants are upper-case, replacing free terms. Constructors can quantify over bound variables, written right after the declaration as {`id : id`} where the first identifier represents the variable and the second annotates its type.

The grammar describes two styles of LF declaration. Experienced users will be accustomed to the Twelf syntax used below to represent the untyped lambda calculus.

```
term : type.
app : term -> term -> term.
lam : (term -> term) -> term.
z : term.
s : term -> term.
```

The new Beluga style stresses that constructors inhabit a primitive type.

```
LF term : type =
| app : term -> term -> term
| lam : (term -> term) -> term
| z : term
| s : term -> term
;
```

3.2 Operators

A type constructor which takes two arguments can be declared an infix operator using the `--infix` pragma. The `--infix` pragma is proceeded by the identifier of the operator which must match the identifier of the LF declaration to which it is applied. The association is specified next to be left, right, or non-associative. Operator precedence is specified last as a natural number, delineated from largest to smallest such that the largest number takes the highest precedence.

The precedence of ordinary prefix operators can be adjusted using the `--prefix` pragma which takes all the same arguments as `--infix` except associativity. Prefix operators default to the low precedence of -1 if none is specified.

The operator is treated as non-associative by default if no associativity is stated explicitly. The default associativity can be altered with the `--assoc` pragma.

Example: A user may wish to declare `app` and `lam` as infix operators while ensuring that `red` has higher precedence. Seeing as no associativity was listed, `lam` takes the default associativity, here defined as `right`.

```
--assoc right

app: term -> term -> term. --infix app none 1.
lam: (term -> term) -> term. --infix lam 1.
red: term -> term --prefix red 2.
```

3.3 Schemas and Context Variables

A schema, declared with the `schema` keyword, consists of ***blocks*** of atomic assumptions. The `block` keyword introduces a comma-separated list of atoms. The first assumption is always a type judgment of a free variable, followed by terms which are well-typed under that judgment. Schemas comprised of several blocks separated by the plus sign `+` denote alternating assumptions.

Schemas classify terms as types classify terms. A context `g` has the schema `sCtx` if every declaration in `g` is an instance of schema `sCtx`. Schemas may quantify over a particular bound variable, written right after the declaration as `[id : id]` where the first identifier represents the bound variable and the second provides its type. Contexts can be passed as ***context variables*** annotated with their associated schema.

```
schema natCtx = block (x:tm, u: eq_tm x x);
schema zeroCtx = some [t:tp] block (x:tm, u: x has_type t);
schema altCtx = block (x:tp, u:eq_tp x x) + block (x:tm, u:eq_tm x x)
```

3.4 Contextual Objects and Types

A ***contextual object*** relates that `M` is an open term in a context `g`, written `g |- M`. The context `g` provides bindings for the free variables, represented as a context variables bound to a particular schema, explicit blocks, or a context variable extended with a block. The turnstile `|-` separates the context of assumptions from the conclusion. Contextual objects appear within functions in ***boxes***, enclosed between brackets `[ ]` to indicate that the data is closed. Note that boxing does not preclude meta-variables within the contextual object so long as they are associated with a delayed substitution to close the object as soon as the appropriate mapping is determined.

Contextual types encapsulate assumptions about the type of an object depending on its ***context***. A data-level object `M` has contextual type `[g |- A]` if `M : A` occurs in `g`. Beluga takes contextual types as base types.

In Beluga, contextual objects are always embedded within functional programs. Likewise, contexts are classified at the computation level. Keeping with the two-level system approach in mind, it is important to realize that even when contexts and contextual objects are manipulated by computations, they occupy LF-space regardless.

3.5 Meta-variables and Parameter Variables

Contextual objects may contain bound variables and meta-variables representing open terms. If a binding is explicit in the context, the bound variable is called with its identifier. A bound variable implicit to a context variable is retrieved from the schema with a ***parameter variable*** `#p`, which always takes a lower-case identifier preceded by the the tag `#` . A ***concrete parameter***, likewise lower-case, calls a binding from a block using the block's identifier. If the block or schema contains more than one element, a particular component is referenced with a ***projection***. The element is specified either by name or an integer indexing its relative position in the block.

A ***meta-variable*** `M` stands for an LF term. Note that meta-variables are always upper-case. Seeing as variables may occur with that term, the meta-variable is associated with a delayed substitution applied when the appropriate mapping is determined from the context. Likewise, a projection also takes a delayed substitution if the element to which it refers contains variables. The substitution is specified directly after the meta-variable, seperated by whitespace. A space-separated list of substitution tokens denotes a union of their domains. Beluga encodes substitutions as follows:

`..`	the identity substitution with respects to the context variable
`^`	the empty substitution
`id`	a bound variable
`id.[id,int]`	a projection of a block declaration
`#id.[id,int]`	a projection of a schema parameter
`#ID subst`	a substitution variable

Example: Given a schema `eCtx = block x :  term, e:eq x x`, a context variable `g:eCtx`, and a block `b:block x':tp, e':eqt x' x'`, a substitution may be defined as:

Writing identity substitutions explicit

`[g |- M[..]  ]`	`M` is bound by assumptions in `g`
`[g |- #p.e[..]  ]`	`e:eq x x` is bound by assumptions in `g`
`[g, x:tp |- M[.., x]]`	`M` is bound by `x:tp` or assumptions in `g`

Identity substitutions may be omitted

`[g |- M]`	`M` is bound by assumptions in `g`
`[g |- #p.e]`	`e:eq x x` is bound by assumptions in `g`
`[g, x:tp |- M]`	`M` is bound by `x:tp` or assumptions in `g`

Weakening substitutions

`[g, b |- M[.., b.1] ]`	`M` is bound by `x':tp` or assumptions in `g`
`[g, b |- M[b.1, b.2] ]`	`M` is bound by assumptions in `b`

3.6 Substitution Variables

Substitutions can be represented as ***substitution variables***, denoted with the tag `#` and an upper-case identifier. They are indexed as `#S: [g |- h]`, where `#S` is a substitution variable which stands for a mapping with domain `g` and range `h`. Substitution variables are associated with a substitution closure just as contextual variables are associated with a delayed substitution. A substitution closure specifies the substitution to be applied to the substitution variable itself, written with brackets `[ ]` after the substitution variable as `#S`[σ] where σ is the closure substitution. Closure substitutions are encoded with the same tokens as delayed substitutions for meta-variables.

Note that the parameter variable `#id`, the meta-variable `Id`, and the substitution variable `#Id` are all unrelated. The case of the identifier and the presence of the tag `#` are merely syntactic devices to distinguish variable types.

3.7 Inductive and Stratified Datatypes

Inductive datatypes are indexed by contextual objects to express relationships between contextual objects and contexts. They are declared in a similar fashion to Beluga-style LF type families, using the `ctype` keyword instead of `type` to invoke the computation level. Inductive datatypes may be formed of other inductive datatypes, of contexts and of contextual objects, guarded by a constructor. Inductive datatypes take upper-case identifiers. We distinguish between *inductive types* and *stratified types*. Inductive types correspond to fix-point definitions in logic and must be strictly positive, i.e. the type family we are defining cannot occur in a negative occurrence. Stratified types define a recursive type by induction on an index argument. As a consequence, the type family we are defining may occur in a negative position, but the index is decreasing.

Example 1: The inductive datatype `Opt`, for example, encodes an option type which is built in to many ML-style languages. A value of type `Opt` is either empty (`None`) or contains a term of type `T`.

```
inductive Opt : ctype =
| None : Opt
| Some : {T : tp}Tm [T] -> Opt;
```

Example 2: The following datatype `Tm` is not inductive.

```
inductive Tm : ctype =
| Lam : (Tm -> Tm) -> Tm;
```

This clearly illustrates the difference between LF definitions and inductive data types. Note that the function space `Tm -> Tm` is a strong function space; we cannot match and inspect the structure of this function, but can only observe its behavior. Moreover, this function may be an arbitary function that is defined by recursion, pattern matching or by referring to other inductive definitions.

The problem can be best illustrated by the following problematic definitions.

```
inductive D : ctype =
| Inj : (D -> D) -> D;

rec out : D -> (D -> D) =
fn x => case x of Inj f =>  f;

let omega  : D = Inj (fn x -> (out x) x);
let omega' : D = (out omega) omega ;
```

The definition of D seems sensible at first; but we can cleverly create a non-terminating computation omega' that will keep on reproducing itself. The problem is the negative occurrence in the definition D.

Example 3: The following type family Tm is stratified.

```
stratified Tm : tp -> ctype =
| Lam : (Tm A -> Tm B) -> Tm (arr A B);
```

Here the constructor Lam takes a function of type (Tm A -> Tm B) as an argument. However, we can never pattern match on this function to inspect its shape and we can never recurse on it; we can recurse on the index A instead.

3.8 Functions

Functions analyze contextual objects, contexts, and indexed datatypes. They are declared with the let keyword or the rec keyword for recursive functions. A function signature defines the function type using higher-order abstract syntax, where the arrow -> is taken as the simply-typed function space, overloading the notation for the LF function space. Universal quantification is denoted between braces {} by an indexed variable. Context variables are indexed by schemas, meta-variables by a contextual object, and substitutions by domain and range.

Explicit arguments from the function signature correspond to computation variables in the function body. An object comprising the left-hand side of an arrow function is introduced as fn id whereas quantifiers are constructed as mlam id. Multiple variables may be introduced at once by writing fn id, id, ... or mlam id, id, ... though the two keywords must be separated by the double arrow =>. If a term from the signature does not require a variable representation in the body, it is surrounded by parentheses ( ) to indicate it is implicit.

Example: The type signature below reads: "for all contexts g with schema xaG, for terms M which are expressions in the context of g, there is a proof that M is equal to itself." The computation-level variables are identified as h and N.

```
rec refl : {g : xaG} {M : [g |- term]} [g |- eq M M] =
FN h => mlam N => ...
```

Recall that by default the meta-variable M is associated with the identity substitution which may be omitted. Hence the definition is equivalent to writing.

```
rec refl : {g : xaG} {M : [g |- term]} [g |- eq M[..] M[..] ] =
FN h => mlam N => ...
```

Example: In this type signature, the context `g` is implicit, not used anywhere within the function body. Variables need only be introduced for the input types `[g |- oft M T[^]]` and `[g |- oft M T1[^]]`, where `oft` stands for "of type". The meta-variables `M`, `T`, and `T1` are implicitly bound at the outside. By default `M` is associated with the identity substitution; its type is hence `[g |- term]`. The meta-variables `T` and `T'` are associated with a weakening substitution, written as `[^]`. Their type is hence `[ |- tp]` and the weakening substitution `^` transports an object of type `tp` that is closed to an object in the context `g`.

```
rec trans : (g:xaG) [g |- type_of M T[^] ] -> [g |-  M T1[^] ] ->
[g |- eq T T1] = fn d1, d2 => ?;
```

We use the question mark to denote an incomplete program.

3.9 Pattern-matching

Beluga provides powerful ML-style pattern-matching to analyze contextual object language, supporting matching on:

- the shape of contexts
- specific bound variables
- meta-variables and substitutions
- constructors
- inside λ expressions

Pattern matching can be nested.

Inductive proofs in Beluga are represented as recursive functions about contextual LF objects using pattern matching. Each case of the proof corresponds to one branch in the function.

4 Type Reconstruction

Dependently typed systems can be extremely verbose since dependently typed objects must carry type information. It would be incredibly tedious to provide annotations on every variable when programming yet it can be very difficult to keep track of types without them as a reader. Type reconstruction relieves the user of the burden of providing type information while retaining the benefits of strongly-typed languages. Beluga infers types, reconstructs implicit variables with fresh identifiers, and outputs annotations. Features like holes, context subsumption and meta-variable strengthening improve usability whereas name preferences customize the output.

4.1 LF

Type reconstruction is, in general, undecidable for LF. Beluga's algorithm reports a principal type, a type error, or that the source term needs more type information. Every LF declaration is processed one at a time. Given a constant declaration, the types of the free variables and any omitted index arguments are inferred when necessary, constituting implicit arguments. As a consequence, users need only provide type annotations where no type can be inferred.

Note that Beluga only outputs type information about the LF objects embedded within computations, not the actual LF signature. By default, implicit arguments are not printed beside their constructors. The `+implicit` command argument overrides this behavior.

Example: Say for instance that terms are indexed by type `tp`, corresponding to the LF declaration `term: tp -> type`. Type reconstruction can infer that any free variable following the `term` constructor must be of type `tp`. A contextual object `M: [term T]` takes an implicit quantifier: `{T : tp} M : [term T]`. Note that `T` may itself contain implicit arguments depending on the type of the term. If `T` is an arrow type `arr A B`, then `A` and `B` will be implicit to `T`.

4.2 Functions

Type reconstruction for a Beluga function begins by reconstructing the type signature. Similar to LF reconstruction, an index argument is implicit to a computation-level type if it occurs either as a free meta-variable or index argument in the computation-level type; it need not be passed to functions if it was implicit when the type of the function was declared. Note that implicit meta-variables are not associated with a substitution. Once the full type of the function is known, type reconstruction introduces the implicit variables and inserts implicit arguments at recursive calls. These implicit arguments are not outputted unless the program was executed with the `+implicit` command line option.

Pattern-matching on an object often refines its type. Case-analysis on a meta-variable `M: [g |- term]`, for instance, may yield a case in which `M` is a lambda-abstraction and another in which it is an application. Beluga reconstructs the types of free variables occurring in patterns themselves, inserting any omitted arguments. Once determined, the refinement substitution is stored with the pattern to be applied when the complete type is known.

4.3 Context Subsumption

Beluga automatically detects weakening relationships. A contextual object in the context `g` can be provided in place of a contextual object in the context `h` if `g` can be obtained by weakening `h`.

Example: Consider the following code segment.

```
schema eqCtx = block x:term, eq x x;
schema eCtx = block x:term, u:eq x x, equal x x;

rec ref : {h : eqCtx} {M : [g |- term]} [g |- eq M M] = ? ;
```

The recursive function `ref` expects a context `h` satisfying the schema `eqCtx`. Beluga would allow passing some context `g:eCtx` to `ref` since `g:eqCtx` can be obtained by adding `equal x x` from `h`.

4.4 Meta-variable Strengthening

Automatic strengthening can be disabled by putting the `--nostrengthen` pragma at the beginning of the program.

4.5 Holes

Beluga supports ***holes*** for bound variables as well as contextual objects and variables. A bound variable in an abstraction which does not appear in the body can be replaced with the underscore _.

4.6 LF and Computation Holes

Holes which take the place of contextual objects and variables are denoted with the question mark `?`. They can appear within functions either embedded within LF boxes or placed directly into the computation to represent normal terms or function outputs, respectively. During reconstruction, Beluga determines the expected type, printing all objects declared in the appropriate scope that could fit.

Note that holes do not synthesize objects; they can only be used in instances where type-checking is possible. Objects of the form `[g |- ?]`, for instance, often cannot be reconstructed because there are no terms against which to type check the hole.

Holes of this nature can be filled using Interactive Mode.

4.7 Name Preference

Beluga assigns names to anonymous variables during reconstruction. These names are generated randomly but appending a type family constructor declarations the `--name` pragma sets a preferred naming choice for anonymous variables of that type.

```
name_prg ::=  lf_decl --name id ID "."
```

The first identifier must match the name of the constructor to which it is being applied. The second identifier is the preferred name itself. Note that name preference declarations have no affect on parsing.

5 Logic Programming with Queries

A query is posed as a `--query` declaration.

```
qdecl ::= --query int int [ID]:lf_typ.
```

The first integer specifies the expected number of solutions whereas the second specifies the maximum number of tries. Beluga will attempt to solve the query to determine whether it has the expected number of solutions while never exceeding the user-defined limit. Supplying either integer field with an asterisk * will set the quantity to unlimited.

Beluga's logic programming engine is active by default. It can be disabled with the `-logic` switch from the command line.

```
# bin/beluga -logic path/to/file.bel
```

6 Coverage

Coverage checking ensures that every closed object of a given type is an instance of a case outlined in the program, ruling out partial functions. Beluga does not perform coverage checks by default. To initiate a coverage check on a given program, include the `--coverage` pragma at the beginning of the program. The program will not compile unless the coverage check is successful. The `--warncoverage` alerts the user of missed cases without failures. Should a coverage check fail, Beluga will output the location of the incomplete pattern matching along with the cases not covered, similar to ML-style exhaustiveness checking.

7 Modules

Modules provide namespace separation within a program. They are declared with ML-style syntax with the `module` keyword followed by an upper-case identifier. The user can specify an abbreviation with the `--abbrev` pragma, followed by the module's actual identifier and the desired abbreviation. Structural components are listed after the `struct` keyword in the usual Beluga syntax. The module declaration is terminated with the `end;` keyword.

Example: A module defining natural numbers could be declared as follows:

```
module Nats = struct
  nat : type.
  z : nat.
  s : nat -> nat.

  rec suc : [ |- nat ] -> [ |- nat] =
  fn n => let [ |- N ] = n in [ |- s N];

  let two = [ |- s (s z)];
  let three = suc two;
end;
--abbrev Nats N
```

Module components are accessed via fields as `Id.id`. The constructor `z` from the example above is called with `Nats.z` or `N.z` using the abbreviation. A module can be opened within a program using the `--open` pragma followed by the module identifier. The components of open modules can be access directly, ie: `z` instead of `Nat.z`. Note that if module `B` is opened in the body of module `A`, then opening `A` within a program also opens `B`.

8 Interactive Programming with Emacs

Beluga supports interactive proof development, with an interactive mode integrated with the Emacs editor. The programmer can leave "holes" for terms in the program using the `?` symbol, and then use the following commands to query and partially fill the holes.

8.1 Keybindings

C-c C-x command	Execute shell command
C-c C-c	Compile
C-c C-l filename.bel	Load (necessary before the next commands)
C-c C-e	Erase hole highlights
C-c C-s num id	Split on variable `id` at hole `num`
C-c C-i num	Introduce variable `num`
C-c C-t	Get type of the term under the cursor

Note that the hole numbers `num` count the holes in order of occurrence, starting at 0.

8.2 Type Information

The user can query the type of the term represented by a hole. This feature is not yet implemented in all cases.

8.3 Variable Introduction

Beluga can generate variable introductions based on the type of a function. The user enters the hole directly after the equal sign.

```
rec fun : {g : sCtx} {M : [g |- term]} [g |- oft M T[^] ] ->
[g |- oft M T'[^] ] -> [ |- eq T T'] = ?;
```

Enter `C-c C-i 0` and Interactive Mode introduces variables:

```
rec fun : {g : sCtx} {M : [g |- term]} [g |- oft M T[^]] ->
[g |- oft M T'[^]] -> [ |- eq T T'] =
FN g => mlam M => fn d => fn f => ? ;
```

8.4 Variable Splitting

Interactive mode supports variable splitting to fill computation-level holes with pattern matching. Use the split command with the hole number and variable name, and Beluga will elaborate the hole with case analysis, generating cases to cover all possible instances of the particular variable. After splitting one must recompile and reload the program, to continue splitting.

9 HTML mode

Beluga source code can be outputted to HTML complete with linking, highlighting, and unicode turnstiles. From the command line, compile the program with the `+HTML` option.

```
# bin/beluga +HTML path/to/source.bel
```

An HTML file called `<source>.html` is created in the directory containing the source file. The type reconstruction information appears on the web page as preformatted text in code boxes between `<pre>` tags. Beluga keywords along with user-defined types, constructors, and functions are enclosed between `<keyword>` and `<code>` tags, respectively, stylized with CSS embedded directly into the HTML. To forgo the default style, use the `-css` option to output the HTML body only.

```
# bin/beluga +HTML -css path/to/source.bel
```

Alternatively, the user can specify a particular CSS stylesheet for the HTML page with the `+cssfile` option. Note that filepaths must be relative to the directory in which the Beluga program is located, not the terminal current directory.

```
# bin/beluga +HTML +cssfile path/to/style.css path/to/source.bel
```

HTML mode includes a markup language to add text to the web page. Three backticks ‘‘‘ introduces and concludes a block of text to be rendered onto the page. In addition to the syntax described in the table below, pure HTML can embedded directly into text blocks. In fact, the special characters listed at the end are merely mnemonics for HTML encodings.

‘‘‘	Begin/end text block
# Level 1 ## Level 2 ### Level 3	Headers
text	Italicize text
text	Bold text
’code’	Inline code

1.
2.
3 .
..
	Ordered list

- - - ..	Unordered list
-----	Horizontal line
&rarr;	Right-arrow
&Gamma;	Upper-case gamma
&delta;	Lower-case delta
&forall;	Turned A
&exist;	Turned E