# Covering all bases: design and implementation of case analysis for contextual objects

Brigitte Pientka and Joshua Dunfield

McGill University, Montreal, Canada,
{bpientka,joshua}@cs.mcgill.ca

**Abstract.** We consider the question: Does a set of patterns cover all objects of a given type? This is straightforward in the simply-typed setting, but undecidable in the presence of dependent types. We discuss the question in the setting of Beluga, a dependently-typed programming and reasoning environment which supports programming with contextual objects and contexts. We describe the design and implementation of a coverage algorithm for Beluga programs and provide an in-depth comparison to closely related systems such as Twelf and Delphin. Our experience with coverage checking Beluga programs shows that many problems and difficulties are avoided. Beluga's coverage algorithm has been used on a wide range of examples, from mechanizing the meta-theory of programming languages from Pierce's textbook *Types and Programming Languages* to the examples from the Twelf repository.

## 1 Introduction

Beluga [12–14] is a programming and reasoning environment for formal systems and proofs. Users specify formal systems within the logical framework LF [6] and implement proofs about formal systems as dependently-typed recursive functions that pattern match on LF objects.

Beluga's strengths come from supporting encodings based on higher-order abstract syntax (HOAS), in which binders in the object language are represented as binders in LF's meta-language. As a consequence, users can avoid implementing common and tricky routines dealing with variables, such as capture-avoiding substitution, renaming and fresh name generation. In addition, Beluga provides intrinsic support for contexts and contextual objects—LF objects that depend on assumptions. Consequently, users can avoid the bureaucracy of explicitly managing and reasoning about contexts. Properties about contexts such as well-formedness, uniqueness of each assumption, weakening, and strengthening are directly supported. This allows a direct and elegant implementation of inductive proofs as recursive functions over contextual objects.

In this paper, we present the design and implementation of coverage checking for Beluga. Coverage checking of functional programs ensures that the user wrote an exhaustive set of cases. It is typically an iterative process, splitting a pattern of a given type into an equivalent set of more precise patterns. For example, suppose we have a datatype `tp` with two constructors: `nat : tp` denoting a base

type and `arr : tp → tp → tp` denoting the type of object-level functions. To see that the set of patterns $Z = \{$`nat`, `arr S1 S2`$\}$ covers all objects of type `tp`, we split `T` (the general pattern, which matches anything) into $\{$`nat`, `arr T1 T2`$\}$ where `nat` and `arr T1 T2` are coverage goals. Each coverage goal must be an instance of the specified set of patterns in $Z$. In this small example, $Z$ immediately covers `tp`. But suppose the program has the cases $Z' = \{$`nat`, `arr nat S1`, `arr (arr S1 S2) S3`$\}$. Now the coverage goals `nat` and `arr T1 T2` are not immediately covered. We need to refine the coverage goals again by splitting `T1`. This results in the set $\{$`nat`, `arr nat T2`, `arr (arr T3 T4) T2`$\}$. Each coverage goal in this set is indeed an instance of the specified set of patterns $Z'$, so $Z'$ is exhaustive.

In summary, to check that all objects of type $A$ are covered by a set of user-defined patterns $Z$, we generate a set of coverage goals, called a covering set, containing all constructors of the type $A$. If every coverage goal is matched by one of the user's patterns, coverage succeeds. If not, we iterate and refine the covering set by splitting on a variable. There is a choice of where to split (which variable), and how deeply to split. We could have chosen to refine `T2` instead of `T1`. Coverage is a search problem: to exhaust the search space, we need to split deeply enough so that, if no covering set is found, it is because there is some object not covered; to avoid combinatorial explosion, we also want to avoid splitting more deeply than needed.

Coverage checking in simply-typed functional programming languages is straightforward, and has rarely been described in detail. In particular, the choice of variable to split is less important—it is merely a matter of efficiency. The maximum splitting depth is bounded by the depth of the user's patterns.

In languages like Beluga, coverage faces several challenges: dependent types, HOAS encodings, contextual objects, and contexts. Dependent types make coverage checking undecidable. The main problem is that knowing when a type is empty is undecidable [7, p. 179]; in languages with dependent types, empty types are an essential tool for representing impossibility (contradiction). Moreover, Beluga supports encodings using HOAS, so the splitting operation must rely on higher-order unification, which is in general undecidable [5]. Finally, we support not only patterns on closed objects as described above, but we allow patterns such as `[a:tp] arr a a` describing a function type whose domain and codomain are the type variable `a:tp`. More generally, given a context `g` containing type variables, we allow patterns that are well-typed within the context `g`; for example, `[g] arr (T1 ..) (T2 ..)` stands for a pattern where the pattern variables `T1` and `T2` can refer to the variables declared in the context `g`. This leads to new considerations.

This paper describes two contributions:

*1. Design and implementation of a coverage checker:* We present a sound theoretical foundation for coverage of contextual objects and contexts based on our earlier work [4]. Building on these ideas, we have designed and implemented a coverage algorithm in the Beluga system. Our coverage algorithm splits objects up to a certain depth and also supports absurd patterns which allow the programmer to explicitly state that a given case is impossible. We have used it on a

wide range of examples, from mechanizing proofs about programming languages from Pierce's textbook [16] to examples we translated from the Twelf repository. Our experience shows that the implementation of the coverage checker is transparent and its performance is competitive.

*2. In-depth comparison to other systems:* Dependently-typed systems such as Twelf [11], Delphin [17, 18], and Agda [1, 10] implement coverage, even though the general question is undecidable. To highlight the similarities and differences, we discuss the implementation of the proof that evaluation in a small language with arithmetic expressions is deterministic (see Pierce [16, Ch. 3]). We deliberately chose such a simple example to concentrate on the main ideas and bring out the similarities and differences between these systems. Second, we give a program that translates well-typed terms in HOAS to well-typed terms in de Bruijn format. This example exploits context matching in Beluga and highlights the additional issues in coverage.

Compared to Twelf and Delphin, our experience with coverage checking Beluga programs shows that Beluga avoids many difficulties and requires the user to prove fewer lemmas. Compared to Agda [10], our approach is similar but handles a richer class of patterns. The electronic appendix [15] includes the examples in Twelf, Delphin, Agda and Beluga.

## 2 Example: Determinism of Small-Step Semantics

### 2.1 Specifying Terms, Values and Small-Step Semantics in LF

We first specify this small language in the logical framework LF [6]. We begin by defining a type `tm` of terms and the constructors `z`, `succ`, and `pred` that inhabit the type `tm`. We also define when a term is a value using the type family `value`. The constant `v_z` states that `z` is a value and the constant `v_succ` says that `(succ V)` is a value if `V` is a value.

```
tm    : type .
z     : tm.
succ  : tm → tm.
pred  : tm → tm.

step  : tm → tm → type .
s_succ: step M M'
     → step (succ M) (succ M').
s_pred_zero: step (pred z) z.
```

```
value   : tm → type .
v_z     : value z.
v_succ  : value V → value (succ V).

s_pred     : step M M'
          → step (pred M) (pred M').
s_pred_succ: value V
          → step (pred (succ V)) V.
```

Finally, we specify the small-step semantics in the logical framework LF using the type family `step`. There are four rules specifying the evaluation of terms which are not yet values: two congruence rules `s_succ` and `s_pred` for successor and predecessor, and two rules `s_pred_zero` and `s_pred_succ` for stepping `pred V` where `V` is a value.

### 2.2 Programming Proofs: Determinism of Small-Step Semantics

We discuss the implementation of the proof that the small-step semantics is deterministic in different systems, starting with Beluga.

*Beluga* Beluga [12, 14] is a dependently-typed functional language that supports pattern matching on LF objects. Its unique feature is its intrinsic support for contexts and contextual objects which we will exploit in a later example of translating terms from HOAS to de Bruijn form. A contextual object `[Ψ] M` has the contextual type `A[Ψ]` where `M` has type `A` in the context $\Psi$. Since there are no assumptions in the proofs about the small-step semantics, the context $\Psi$ will be empty: `[ ] M` has type `A[ ]`. This allows us to concentrate on the similarities and differences with related systems. In a later section, we examine the challenges presented by non-empty contexts.

First, we establish a lemma that values do not step, i.e. there is no step rule that applies to a value. In other words, the conjunction of `step M M'` and `value M` leads to a contradiction. We introduce a type `empty` with no constructors. The proof that values do not step is implemented as a recursive function by a case-split on `value M`. If we have derived `value z` using the constant (axiom) `v_z`, then there is in fact no object of type `step z z`: we have a contradiction. This is handled in Beluga by `impossible s in [ ]`, which checks that the type of the object `s` is empty and hence there is no further split possible. If we have derived `value (succ N)` using the object `v_s V` then `V` has type `value N` and we must have used the rule `s_succ D` where `D` is an object of type `step N N'`. But by the induction hypothesis (recursive call) we know that that this is impossible. This proof is realized by the function `values_dont_step`.

```
empty : type .
rec values_dont_step:(step M M')[] → (value M )[] → empty [] =
 fn s ⇒ fn v ⇒ case v of
 | [ ] v_z    ⇒ impossible s in [ ]
 | [ ] v_s V  ⇒ let [ ] s_succ D = s in values_dont_step ([ ] D) ([ ] V) ;

rec det : (step M N1)[] → (step M N2)[] → (equal N1 N2)[] =
 fn s1 ⇒   fn s2 ⇒ case s1 of
 | [] s_succ D      ⇒  let [] s_succ F = s2 in
                       let [] ref = det ([] D)  ([] F) in [] ref
 | [] s_pred_zero   ⇒  let [] s_pred_zero = s2 in [] ref
 | [] s_pred_succ V ⇒ (case s2 of
   | [] s_pred_succ _  ⇒ [] ref
   | [] s_pred D       ⇒ impossible values_dont_step ([] D) ([] v_s V) in [])
 | [] s_pred D      ⇒ (case s2 of
   | [] s_pred F       ⇒ let [] ref = det ([] D)  ([] F) in  [] ref
   | []  s_pred_succ V ⇒ impossible values_dont_step ([] D) ([] v_s V) in []
   | [] s_pred_zero    ⇒ impossible values_dont_step ([] D) ([] v_z)   in []) ;
```

The implementation of the determinacy proof of the small-step semantics follows closely the on-paper proof. We case-analyze the first argument `s1` which derives `(step M N1)[]`, and in each branch case-analyze the second argument. The first case says that `s1` is a proof `s_succ D` that `step (succ M') (succ N1')`, where `D` is a derivation of `step M' N1'`. Hence we know that `s2` is a derivation of `step (succ M') N2`. It must have ended in the rule `s_succ` and can be described by `s_succ E` where the subderivation `E` derives `step M' N2'` and `N2` = `succ N2'`.

By the induction hypothesis we know that `eq N1' N2'`. By inversion, we could only have arrived at `eq N1' N2'` using the rule `ref`, and `N1'` = `N2'`. Therefore, we can establish `eq (succ N1') (succ N2')` by applying `ref`.

In the second case, the `s_pred_zero` rule concludes the derivation `s1` and its conclusion is `step (pred z) z`. Two rules, `s_pred_zero` and `s_pred`, could have been used to derive `step (pred z) N2` in `s2`. For `s_pred_zero`, the two results are obviously equal. For `s_pred`, we have a subderivation `D:step z N2'` and `N2 = pred N2'`. However, there is no rule that can step `z`, so `D` cannot exist. Beluga automatically proves that this case is impossible, so it suffices to write only one case.

The other cases, where `s1` ends in either `s_pred_succ V` or `s_pred D`, follow a similar idea. While it is obvious that there is no step rule for `z`, it is non-trivial for a value `succ V`. Hence, we appeal to the lemma `values_dont_step` and show that we have arrived at a contradiction (empty type).

Beluga's coverage checker accepts the above program. We also note that we did not write out some cases that the coverage checker automatically detects are impossible. For example, if we used `s_pred_zero` for `s1`, we could not have used `s_pred_succ _` for `s2`. Finally, note that we could have obtained a more compact proof by considering `s1` and `s2` simultaneously; the full development can be found in the electronic appendix [15].

*Agda* Agda [10] is based on Martin-Löf's type theory and is well-suited for programming proofs. Unlike Beluga, where we strictly separate the LF language from the recursive programs that analyze and manipulate LF objects, Agda does not separate the language of types from the language of programs. Coverage checking is an important component in the Agda system, since establishing equality between two types may involve executing recursive functions that are used within a type. Agda supports simultaneous patterns, but one can end up writing more cases than in Beluga; Agda splits on a variable only when there is a user-given incentive, meaning a user-written pattern. Consequently, the behavior and the termination of the coverage checker is easy to control. In Beluga, we try to discharge cases by automatically splitting a variable; this process of splitting is guided by heuristics that control how deeply to split. This means that there are some examples where Beluga automatically proves that some cases are impossible while Agda requires one to be more explicit. For example, in Agda we have to explicitly state the case where `s1` uses `s_pred_zero` and `s2` uses `s_pred D`. Since there is no object for `D:step z N2'`, we write an absurd pattern for `D` forcing Agda to check that indeed the type is empty. Beluga automatically discharges this case. Similar to Agda's absurd patterns, which explicitly state that a given type is empty, we support empty patterns using the keyword `impossible`; this is useful when Beluga cannot automatically detect that a case is impossible, and when explicitly including such a case makes the proof more readable.

*Twelf* In Twelf [11], this proof is implemented as a relation. Subsequently, we establish that the relation represents a total function. This is accomplished by first providing a mode declaration stating which arguments of the relation are inputs and which are outputs. A totality declaration will then ensure that all input arguments and all output arguments are covered and that computing the relation terminates. We concentrate here on coverage, and discuss the Twelf implementation in detail, since it highlights some of the subtle difficulties.

We first state the lemma that values do not step.

```
values_dont_step: step M M' → value M → empty → type .
%mode values_dont_step +S +V -I.
nostep_suc : values_dont_step D V I → values_dont_step (s_succ D) (v_s V) I.
```

The simultaneous pattern where we have `value z` and `step z M'` is omitted in Twelf. Its coverage checker proves that this case is indeed impossible. Since we are writing relations, it is not natural to consider the input arguments sequentially.

Next, we translate the Beluga function into a relation in Twelf. One may expect that the resulting Twelf program, given below, should coverage check. Since Twelf does not support absurd patterns, we need a lemma `empty_implies_anything` that, when given an empty type, allows us to conclude anything; in particular, we can conclude that two terms `M` and `N` are equal.

```
det: step M N → step M N' → eq N N' → type .            %mode det +D +E -R.

d_pred_zero        : det s_pred_zero s_pred_zero ref.
d_pred_succ        : det (s_pred_succ V) (s_pred_succ _ ) ref.
d_pred             : det D F ref → det (s_pred D) (s_pred F) ref.
d_succ             : det D F ref → det (s_succ D) (s_succ F) ref.

d_pred_succ_empty_1: empty_implies_anything I Eq → values_dont_step D V I
                 →   det (s_pred (s_succ D)) (s_pred_succ V) Eq.

d_pred_succ_empty_2: empty_implies_anything I Eq → values_dont_step D V I
                →    det (s_pred_succ V) (s_pred (s_succ D)) Eq.
```

Perhaps surprisingly, this proof will not coverage check: Twelf complains about the clauses `d_succ` and `d_pred`. To understand why, we need to inspect the fully reconstructed type. Here is the result of type reconstruction for `d_succ`:

```
d_succ : {M':term} {N1':term} {D:step M' N1'} {F:step M' N1'}
         det D F ref → det (s_succ D) (s_succ F) ref.
```

Universally quantified variables are written using braces: `{M':term}{N1':term}{D:step M' N1'}{F:step M' N1'}` can be read, "For all `M` and `N1'`, and for all derivations `D:step M' N1'` and `F:step M' N1`." Inspecting the types of `D` and `F`, we notice that they describe the same derivation. This happened because we used `ref` in the output position, which constrained some input arguments. The coverage checker now fails because not all possible input combinations are covered. To get around this problem, we explicitly prove congruence lemmas. For `succ`, we prove a lemma `eq_succ` stating that if `eq M' N1'` then `eq (succ M') (succ N1')`. For `pred`, we establish a lemma `eq_pred` which states that if `eq M' N1'` then `eq (pred M') (pred N1')`. We then modify the cases for `d_succ` and `d_pred`:

```
d_succ: eq_succ R R' → det D F R → det (s_succ D) (s_succ F) R'.
d_pred: eq_pred R R' → det D F R → det (s_pred D) (s_pred F) R'.
```

Besides the difficulties in guaranteeing coverage, if the output constrains inputs, Twelf's analysis may run afoul because some variables are free in the output. This issue arises from our specification of the lemma

```
empty_implies_anything: empty → eq M M' → type .
```

As mentioned previously, Twelf's totality checker needs to verify that the relation is a total function and hence the user would typically give the following mode declaration: `%mode empty_implies_anything +I -D.`

We notice that we only assigned modes to the explicit arguments, but not to the implicit arguments `M` and `M'`. By default, Twelf will assign `M` and `M'` a negative (output) mode. But this means they are unconstrained in the output, and if we interpret this relation as a function, the output of the function is not uniquely determined. Twelf reports this as an output freeness error. Luckily, we can force Twelf to treat the implicit arguments `M` and `M'` as inputs by a long-mode declaration.

Some of the difficulties in verifying coverage in Twelf are due to the relational-style of writing proofs. By contrast, in Beluga, we write functions that pattern match on individual arguments, so we can model directly the structure and the information flow of the informal proof; the justification for all cases being covered is more transparent.

*Delphin* Finally, we briefly discuss coverage in Delphin [17], a dependently-typed functional language which supports writing proofs as recursive functions over LF objects. One may hope that, as with Beluga, this would avoid the difficulties we encountered in Twelf. But this is only partially true. Surprisingly, the Delphin program corresponding to our Beluga program `det` fails to pass Delphin's coverage checker; as in Twelf, the Delphin program requires explicit congruence lemmas. The cause is the same as in Twelf: were we to return `ref` directly in the recursive call, type reconstruction constrains the pattern. This design decision was made to avoid having to propagate the constraints generated when pattern matching on dependently typed objects.

*Summary* Our example of proving determinacy of the small-step semantics is very simple and omits higher-order abstract syntax, but illustrates the subtle differences in the behavior of the coverage checker in a variety of systems; the differences come from different approaches to type reconstruction and different design decisions on when to split where. Agda is the most conservative; its coverage checker splits only if there is a user-given incentive, and does not try to prove that a given type is empty unless you claim it is. Compared to Twelf, our functional language avoids pitfalls due to output coverage and free variables, which are caused by having to implement proofs as relations. Beluga's type reconstruction engine constructs the most general type of a pattern together with a refinement substitution. This affects the behavior of the Beluga coverage checker: although the algorithms that generate sets of coverage goals are similar in Twelf, Delphin and Beluga, the set of patterns we derive from the user-written program is more general than in Twelf and Delphin. Hence, Beluga will accept some programs that are rejected by Delphin or Twelf.

## 3 Matching on Contextual Data and Contexts

A unique feature of Beluga is its support for contextual objects and contexts. To illustrate the issues arising we examine a program that translates System F terms from higher-order abstract syntax to de Bruijn form. In the source

language, we distinguish types from terms. But we use a uniform approach for the target language; to distinguish between objects denoting a term and objects denoting a type, we introduce a type `term_or_typ : type` with two inhabitants `typ : term_or_typ` and `term:term_or_typ`. We then define a type family `obj` indexed by elements `term_or_typ` indicating whether the object is a term or a type. We use de Bruijn indices to represent variables in our target language.

```
% Types
tp   : type .
nat  : tp.
arr  : tp → tp → tp.
all  : (tp → tp) → tp.

% Expressions
exp  :  type .

lam  : tp → (exp → exp) → exp.
app  : exp → exp → exp.
tlam : (tp → exp) → exp.
tapp : exp → tp → exp.
```

```
% Uniform target language
obj' : term_or_typ → type .
nat' : obj' typ.
arr' : obj' typ → obj' typ → obj' typ.
all' : obj' typ → obj' typ.

one  : obj' T.
shift: obj' T → obj' T.
lam' : obj' typ  → obj' term → obj' term.
app' : obj' term → obj' term → obj' term.
tlam': obj' term → obj' term.
tapp': obj' term → obj' typ  → obj' term.
```

The target language uses de Bruijn indices, indexing type variables with respect to `tlam`-binders and ordinary variables with respect to `lam`-binders. Finally, we show the program to translate objects `tp` to their respective de Bruijn `obj' typ` representation. The translation of objects of type `exp` to their respective de Bruijn terms of type `obj term` is then similar.

```
schema ctx = exp + tp ;
rec typ2typ' : {g:ctx} tp[g] → (obj' typ)[ ] = fn t ⇒ case t of
| [h, a:tp]  a          ⇒ [] one
| [h, x:exp] #p ..      ⇒ let [] Db = typ2typ' ([h] #p ..) in  [] Db
| [h, a:tp]  #p ..      ⇒ let [] Db = typ2typ' ([h] #p ..) in  [] shift Db
| [h]        nat        ⇒ [] nat'
| [h] arr (T ..) (S ..) ⇒ let [] T' = typ2typ' ([h] T ..) in
                          let [] S' = typ2typ' ([h] S ..) in  [] arr' T' S'
| [h] all λa. T .. a    ⇒ let [] T' = typ2typ' ([h, a:tp] T .. a) in  [] all' T' ;
```

When we recursively analyze objects of type `tp`, we quickly realize that they may contain type variables; moreover, since we will call the function `typ2typ'` from the function which translates expressions to de Bruijn terms, and the context `g` may contain `exp` and `tp`. Hence we first declare the context schema: `schema ctx = exp + tp`.

The type of the function `typ2typ'` can be read as follows: for all context `g` of schema `ctx`, given a contextual object `tp[g]`, the result is a closed de Bruijn term of type `(obj' typ)[]`. This type guarantees that while the input may contain variables declared in `g`, the result is closed: its context is empty.

Three challenges arise in supporting pattern matching on contextual objects and contexts. First, we need a generic case for variables declared in the context `g`. For a concrete bound variable, we can simply refer to it, as in the pattern `[h, a:tp] a`. But we also need a case for some other variables from `h`. We use a *parameter variable* `#q` that stands for *some* declaration in `h` to write the case `[h,a:tp] #p ..` in the function `typ2typ'`.

Second, we exploit context matching to analyze the shape of the context. We need to know the position of a given variable declaration in the context.

Since contexts are ordered, we can peel off one declaration at a time until we find the given variable. When we pattern match on `t` of type `tp[g]` and enter the branch `[h,a:tp] a ⇒ [ ] one`, the context variable `g` is refined to the context `h, a:tp` where `h` is a context variable.

Finally, we distinguish cases on whether we have a type declaration `a:tp` or a term declaration `x:exp` in the context. Since we assign type variables a different index from term variables, the position of a type declaration is determined with respect to type variables only, not term variables.

Matching on the shape of the context allows us to walk through the context and compute the appropriate de Bruijn index. We could modify the implementation by choosing a uniform source language, similar to how the target language indexes its objects by `term` or `typ`. Instead of `a:tp` we would have `a:obj typ`, and instead of the declaration `x:exp`, we say `x:obj term`. The shape of these contexts is given by `schema ctx = some [a:term_or_typ] obj a`, which states that each concrete declaration in a context can be derived by instantiating `a:term_or_typ`. When rewriting the function `typ2typ'`, we then need to pattern match not only on the structure of the context, but also on each declaration in the context. The branch `[h, a:tp] a ⇒ [] one` becomes `[h, a:obj typ] ⇒ [] one`; similarly, the pattern in `[h, x:exp] ⇒ #p ..` becomes `[h, x:obj term]`. For the full implementation, see the electronic appendix [15]. Having generic context schemas and context matching leads to new considerations in coverage. These kinds of programs are not easily translated to Twelf or Delphin because neither system has contexts that programmers can directly manipulate and inspect.

## 4 Foundations of Contextual Coverage

Our central question is: Does a set of patterns cover the type $A[\Psi]$? To answer this question, we present a general way of generating a set of patterns thereby providing a foundation for splitting an object of type $A[\Psi]$ into different cases. For example, in the function `typ2typ'` we ensure that the set of patterns $Z = \{$`[h,a:tp] a`, `[h,a:tp] #p ..`, `[h,x:exp] #p ..`, `[h] nat`, `[h] arr (T ..) (S ..)`, `[h] all λa. T .. a`$\}$ covers all elements of type `tp[g]`, that is, every term of type `tp[g]` is an instance of some pattern in $Z$.

We begin by reviewing the foundation of our pattern language. Patterns are derived from an extension of the logical framework LF where we think of every LF object within a context. Contextual objects were introduced by Nanevski et al. [9]. A contextual object $M$ in a context $\Psi$ is written $[\Psi]M$ and has contextual type $A[\Psi]$. To precisely define holes in contextual objects, we support meta-variables. For example, in the pattern `[h] arr (T ..) (S ..)` the pattern variables `T` and `S` are meta-variables. Meta-variables are associated with a substitution $\sigma$ and are written $u[\sigma]$ in our theoretical foundation. In concrete syntax, we write `T ..` for a meta-variable under the identity substitution. We also support context variables that abstract over concrete contexts, and parameter variables that abstract over variable declarations [12]. We characterize only normal forms, since only these are meaningful in LF. We do this by defining *normal terms* $M$

and *neutral terms* $R$. The syntax guarantees that terms contain no $\beta$-redexes, and the typing rules guarantee that well-typed terms are fully $\eta$-expanded.

Here, we omit block declarations of the form $\Sigma y_1{:}A_1, \ldots, y_k{:}A_k.\ A_{k+1}$, which group multiple assumptions together so that whenever $y_i$ exists, there also exist $y_j$ for all $i$ and $j$ from 1 to $k+1$. This feature is not crucial to the main ideas of coverage, but complicates the description.

| | | | |
|---|---|---|---|
| Atomic types | $P$ | ::= | $a\ M_1 \ldots M_n$ |
| Types | $A, B$ | ::= | $P \mid \Pi x{:}A.B$ |
| Normal terms | $M, N$ | ::= | $\lambda x.\ M \mid R$ |
| Neutral terms | $R$ | ::= | $c \mid x \mid u[\sigma] \mid p[\sigma] \mid R\ N$ |
| Substitutions | $\sigma$ | ::= | $\cdot \mid \sigma\,;\ M \mid \sigma\ ,\ R \mid \mathsf{id}_\psi$ |
| Contexts | $\Psi, \Phi$ | ::= | $\cdot \mid \psi \mid \Psi, x{:}A$ |
| Meta-contexts | $\Delta$ | ::= | $\cdot \mid \Delta, u{::}A[\Psi] \mid \Delta, p{::}A[\Psi] \mid \psi{::}W$ |

We distinguish between three kinds of variables: *Ordinary bound variables* $x$ and $y$ are bound by $\lambda$-abstraction at the LF level; these variables are declared in a context $\Psi$. *Contextual variables* stand for open objects, and include *meta-variables* $u$ and $v$, which represent general open objects, and *parameter variables* $p$ that can only be instantiated with an ordinary bound variable. Contextual variables are introduced in computation-level case expressions, and are instantiated via pattern matching. Contextual variables are associated with a postponed substitution $\sigma$. The intent is to apply $\sigma$ as soon as we know the object the contextual variable should stand for. The domain of $\sigma$ thus includes the free variables of that object, and the type system statically guarantees this.

Our foundation supports *context variables* $\psi$ which allow us to reason abstractly with contexts, and write recursive computations that manipulate higher-order data. Context variables, meta-variables and parameter variables are introduced at the computation level.

As types classify objects, and kinds classify types, we introduce the notion of *schemas* $W$ that classify contexts $\Psi$. Context variables, meta-variables and parameter variables are declared in the meta-context $\Delta$.

Substitutions $\sigma$ are built from normal terms $M$ and atomic terms $R$. This is necessary because when we extend the substitution with a neutral term, we may not always have its type and hence we cannot guarantee that the neutral term is also a well-typed normal term. This issue arises when we push a substitution $\sigma$ under a lambda-abstraction $\lambda x.M$ and need to extend the substitution $\sigma$ with the variable $x$: If $x$ has a functional type, $x$ is not a well-typed normal term and must be $\eta$-expanded. We do not make the domain of a substitution explicit, to simplify the theory and avoid having to rename domains. Finally, we have a first-class notion of identity substitution $\mathsf{id}_\psi$ whose domain is a context variable. We write $[\sigma]N$ for substitution application.

We assume that type constants and object constants are declared in a signature $\mathsf{S}$ as pure LF objects, that is, types not containing meta-variables, parameter variables or context variables. We suppress this signature since it is the same throughout all derivations.

### 4.1 Bidirectional Type System

We type data-level terms bidirectionally. Normal objects are checked against a given type $A$ in the judgment $\Delta;\Psi \vdash M \Leftarrow A$, while neutral objects synthesize their type: $\Delta;\Psi \vdash R \Rightarrow A$. Substitutions are checked against their domain: $\Delta;\Psi \vdash \sigma \Leftarrow \Phi$.

Data-level normal terms

$$\frac{\Delta;\Psi, x{:}A \vdash M \Leftarrow B}{\Delta;\Psi \vdash \lambda x.\, M \Leftarrow \Pi x{:}A.B}\,\underline{\Pi\mathrm{I}} \qquad \frac{\Delta;\Psi \vdash R \Rightarrow P' \quad P' = P}{\Delta;\Psi \vdash R \Leftarrow P}\,\underline{\mathsf{turn}}$$

Data-level neutral terms

$$\frac{x{:}A \in \Psi}{\Delta;\Psi \vdash x \Rightarrow A}\,\underline{\mathsf{var}} \qquad \frac{c{:}A \in \Sigma}{\Delta;\Psi \vdash c \Rightarrow A}\,\underline{\mathsf{con}} \qquad \frac{u{::}A[\Phi] \in \Delta \quad \Delta;\Psi \vdash \sigma \Leftarrow \Phi}{\Delta;\Psi \vdash u[\sigma] \Rightarrow [\sigma]^a_\Phi A}\,\underline{\mathsf{mvar}}$$

$$\frac{p{::}A[\Phi] \in \Delta \quad \Delta;\Psi \vdash \sigma \Leftarrow \Phi}{\Delta;\Psi \vdash p[\sigma] \Rightarrow [\sigma]^a_\Phi A}\,\underline{\mathsf{param}} \qquad \frac{\Delta;\Psi \vdash R \Rightarrow \Pi x{:}A.B \quad \Delta;\Psi \vdash N \Leftarrow A}{\Delta;\Psi \vdash R\ N \Rightarrow [N/x]^a_A B}\,\underline{\Pi\mathrm{E}}$$

Data-level substitutions

$$\frac{}{\Delta;\Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{}{\Delta;\psi,\Psi \vdash \mathsf{id}_\psi \Leftarrow \psi}$$

$$\frac{\Delta;\Psi \vdash \sigma \Leftarrow \Phi \quad \Delta;\Psi \vdash R \Rightarrow A' \quad [\sigma]^a_\Phi A = A'}{\Delta;\Psi \vdash (\sigma\,, R) \Leftarrow (\Phi, x{:}A)} \qquad \frac{\Delta;\Psi \vdash \sigma \Leftarrow \Phi \quad \Delta;\Psi \vdash M \Leftarrow [\sigma]^a_\Phi A}{\Delta;\Psi \vdash (\sigma\,; M) \Leftarrow (\Phi, x{:}A)}$$

We will tacitly rename bound variables, and maintain that contexts and substitutions declare no variable more than once. Note that substitutions $\sigma$ are defined only on ordinary variables $x$, not on contextual variables $u$ or $p$. We also require the usual conditions on bound variables. For example, in $\underline{\Pi\mathrm{I}}$ the bound variable $x$ must be new and cannot already occur in $\Psi$. This can always be achieved by $\alpha$-renaming. The typing rules for neutral terms use *hereditary substitution* $[\cdots]^a_A$, which preserves canonical forms [9]. Hereditary substitution is defined recursively, considering both the structure of the term to which the substitution is applied and the type $A$ of the object being substituted. This operation has been defined and proved to be terminating [9, 12, 13].

For readability, we omit the subscripts. Since hereditary substitution is decidable and the rules above are syntax-directed, data-level typing is decidable.

### 4.2 Context Schemas

Schemas classify contexts. In an earlier example, the schema `ctx = tp + exp` represented a context which consists of type declarations `tp` or of term declarations `exp`. Using dependent types, we can also define a `schema tctx = some[t:term_or_typ] obj t`.

We use + to denote a choice of possible elements in a context, and `some` allows us to describe a generic type where each concrete assumption in a context must be an instance of this generic type. In general, we support grouping multiple declarations into a block (formally a $\Sigma$-type) but we omit it here to concentrate on the essential ideas.

| | | |
|---|---|---|
| Schema elements | $F$ | $::= \mathsf{some}\, x_1{:}A_1, \ldots x_k{:}A_k\, \mathsf{block}\, A$ |
| Schemas | $W$ | $::= (F_1 + \cdots + F_n)^*$ |

Context $\Psi$ checks against schema $W$

$$\frac{}{\Delta \vdash \cdot \Leftarrow W} \qquad \frac{\psi{::}W \in \Delta}{\Delta \vdash \psi \Leftarrow W} \qquad \frac{\Delta;\Psi \vdash A \in F_k \text{ for some } k \qquad \Delta \vdash \Psi \Leftarrow (F_1 + \cdots + F_n)^*}{\Delta \vdash \Psi, x{:}A \Leftarrow (F_1 + \cdots + F_n)^*}$$

To check $A \in F_k$ where $F_k = \mathsf{some}\, x_1{:}A_1, \ldots, x_k{:}A_k\, \mathsf{block}\, B$ we verify that there exists an instantiation $\sigma$ for $x_1{:}A_1, \ldots, x_k{:}A_k$ such that $A = [\sigma]B$.

### 4.3 Meta-substitution

Substitution for meta-variables, parameter variables and context variables has been defined earlier (see [12, 13]). Meta-substitutions provide instantiations for meta-variables $u$, parameter variables $p$ and context variables $\psi$.

We can substitute a normal term $M$ for the meta-variable $u$ of type $A[\Psi]$ if $M$ has type $A$ in the context $\Psi$. Because of $\alpha$-conversion, the variables substituted at different occurrences of $u$ may differ, and we write the contextual substitution as $[\![\hat{\Psi}.M/u]\!](N)$ (and similarly $[\![\hat{\Psi}.M/u]\!]\sigma$ or $[\![\hat{\Psi}.M/u]\!]\Phi$ etc.). Applying $[\![\hat{\Psi}.M/u]\!]$ to the closure $u[\sigma]$ first obtains the simultaneous substitution $\sigma' = [\![\hat{\Psi}.M/u]\!]\sigma$, but instead of returning $M[\sigma']$, it eagerly applies $\sigma'$ to $M$. Similar ideas apply to parameter substitutions. The main difference is that parameter variables can only be instantiated either by bound variables or other parameter variables. Replacing a context variable with a concrete context $\Psi$ is actually straightforward, since context variables can only appear at the left.

## 5 Coverage Checking

A *coverage goal* is a contextual object $[\Psi]M$ that can have meta-variables, parameter variables and context variables, all declared in the meta-context $\Delta$. We write $\Delta;\Psi \vdash M$ for the coverage goal. Intuitively, a coverage goal represents all of its closed instances.

A *coverage problem* consists of the coverage goal and a set of patterns. In Beluga, this set of patterns comes from the program. For example, the case expression in the function `typ2typ'` gives the set of patterns

$$\begin{array}{lllll}
\{\ \mathtt{h{:}ctx} & ;\mathtt{h}, a{:}\mathtt{tp} & \vdash a & : \mathtt{tp} \\
\quad \mathtt{h{:}ctx}, p{::}\mathtt{tp}[\mathtt{h}] & ;\mathtt{h}, a{:}\mathtt{tp} & \vdash p[\mathsf{id}_{\psi_1}] & : \mathtt{tp} \\
\quad \mathtt{h{:}ctx}, p{::}\mathtt{tp}[\mathtt{h}] & ;\mathtt{h}, x{:}\mathtt{exp} & \vdash p[\mathsf{id}_{\psi_1}] & : \mathtt{tp} \\
\quad \mathtt{h{:}ctx} & ;\mathtt{h} & \vdash \mathtt{nat} & : \mathtt{tp} \\
\quad \mathtt{h{:}ctx}, u{::}\mathtt{tp}[\mathtt{h}], v{::}\mathtt{tp}[\mathtt{h}] & ;\mathtt{h} & \vdash \mathtt{arr}\ u[\mathsf{id}_{\mathtt{h}}]\ v[\mathsf{id}_{\mathtt{h}}] & : \mathtt{tp} \\
\quad \mathtt{h{:}ctx}, u{::}\mathtt{tp}[\mathtt{h}, b{:}\mathtt{tp}] & ;\mathtt{h} & \vdash \mathtt{all}\ \lambda b.u[\mathsf{id}_{\mathtt{h}}, b] & : \mathtt{tp}\ \}
\end{array}$$

We explicitly state the type of each meta-variable, parameter variable and context variable; this information is inferred during type reconstruction. In addition, we write meta-variables and parameter variables as closures.

Previous work by Coquand [2] and Schürmann and Pfenning [20] describes coverage checking for closed terms, while Schürmann [19, pp. 197–213] formulated coverage for open terms within regular worlds. However, our setting is

different since we directly support contextual objects, explicit contexts and context matching.

**Definition 1.** *A coverage goal* $\Delta;\Psi \vdash M : A$ *is* immediately covered *by a collection of patterns* $\Delta_i;\Psi_i \vdash M_i : A_i$ *if there exist* $i$ *and a meta-substitution* $\theta$ *such that* $\Delta \vdash \theta : \Delta_i$ *and* $\Delta;[\![\theta]\!]\Psi \vdash [\![\theta]\!]A_i = A$ *and* $\Delta;[\![\theta]\!]\Psi \vdash [\![\theta]\!]M_i = M$.

**Definition 2 (Coverage).** *A coverage goal* $\Delta;\Psi \vdash M : A$ *is* covered *by a set* $Z$ *of patterns* $\Delta_i;\Psi_i \vdash M_i : A_i$ *if every ground instance* $\cdot;[\![\theta]\!]\Psi \vdash [\![\theta]\!]M : [\![\theta]\!]A$, *where* $\cdot \vdash \theta : \Delta$, *is immediately covered by some pattern* $\Delta_i;\Psi_i \vdash M_i : A_i$ *in* $Z$.

To determine if a goal is immediately covered, we have to solve a higher-order matching problem. Higher-order matching in the dependently typed case is undecidable [3] and as in the Twelf system, we concentrate on strict patterns [20]. Informally, a pattern is strict if there exists at least one occurrence of a meta-variable $u[\sigma]$ where $\sigma$ is a *pattern substitution* which maps distinct bound variables to distinct bound variables. This fragment is slightly more liberal than the pure higher-order pattern fragment [8].

In the case where a coverage goal is not immediately covered by a set of patterns, we need to split it and refine it further. Splitting is decomposed into two phases: *splitting inside a context* and *splitting objects*.

### 5.1 Splitting Inside Contexts

There are two ways we can refine a context: refine its general structure according to the context variable's context schema, or refine the types of concrete declarations in the context.

*Splitting a context variable.* First, we can refine the context by splitting on a context variable; in the running example, we split the variable `g` into contexts $\{[\psi_1, x1 : \texttt{exp}], [\psi_1, x2 : \texttt{tp}], [\cdot]\}$, according to the schema of `g`, which is `exp + tp`. This yields three coverage goals, all of which need to be covered:

$$\begin{array}{lllll}
(1) & u{::}\texttt{exp}[\cdot] & ;\cdot & \vdash u[\cdot] & :\texttt{tp} \\
(2) & \psi_1{:}\texttt{ctx}, u{::}\texttt{tp}[\psi_1, a{:}\texttt{tp}] & ;\psi_1, a{:}\texttt{tp} & \vdash u[\mathsf{id}_{\psi_1}, a] & :\texttt{tp} \\
(3) & \psi_1{:}\texttt{ctx}, u{::}\texttt{exp}[\psi_1, x{:}\texttt{exp}] & ;\psi_1, x{:}\texttt{exp} & \vdash u[\mathsf{id}_{\psi_1}, x] & :\texttt{tp}
\end{array}$$

A second example is a modified `typ2typ'` function with a uniform source language: the type `exp` becomes `obj term` and the type `tp` becomes `obj typ`. When we split a context variable `g` of schema `some [a:term_or_typ] obj a`, we get two subproblems:

$$\begin{array}{lll}
(1) & u{::}\texttt{exp}[\cdot] ;\cdot \vdash u[\cdot] & :\texttt{obj typ} \\
(2) & \psi_1{:}\texttt{ctx}, v{::}\texttt{term_or_typ}[\psi_1], & \\
 & u{::}(\texttt{obj typ})[\psi_1, a{:}\texttt{obj}\ v[\mathsf{id}_{\psi_1}]] ;\psi_1, a{:}\texttt{obj}\ v[\mathsf{id}_{\psi_1}] \vdash u[\mathsf{id}_{\psi_1}, a] & :\texttt{obj typ}
\end{array}$$

The splitting depth is controlled by the specified contexts. In `typ2typ'`, the object cased upon is in the context `[g]`, and the longest context in a branch is `[h, a:tp]`, which declares one more bound variable declaration than `[g]`. One split of `[g]`

into $[\psi_1, \ldots]$ suffices; it is useless to split $\psi_1$, since no branch distinguishes its contents. In general,

$$\begin{aligned}\textit{context variable split depth} = &\ (\textit{length of the longest context in a branch}) \\ &- (\textit{length of cased-upon context})\end{aligned}$$

*Splitting declarations in a context.* With dependent types and generic context elements, we may need to refine declarations. Similar to the `typ2typ'` function, a function to translate `obj typ` to `obj' typ` will branch on the patterns

```
{[g, a:obj typ]  a,       [g] nat,
 [g, x:obj term] #p ..,   [g] arr (T ..) (S ..),
 [g, a:obj typ]  #p ..,   [g] all λa. T .. a}
```

Clearly, the second coverage goal, where we refined the context with a generic declaration $\texttt{x:obj}\ v[\mathsf{id}_{\psi_1}]$ is not covered. We need to split and refine $v[\mathsf{id}_{\psi_1}]$. To split a declaration, we use the operation defined in Section 5.2. This refines the second coverage goal into two subcases:

$$\begin{array}{ll}(2.1) & \psi_1{:}\texttt{ctx}, u{::}(\texttt{obj typ})[\psi_1, a{:}\texttt{obj typ}] \ ; \psi_1, a{:}\texttt{obj typ} \ \vdash u[\mathsf{id}_{\psi_1}, a] : \texttt{obj typ} \\ (2.2) & \psi_1{:}\texttt{ctx}, u{::}(\texttt{obj typ})[\psi_1, a{:}\texttt{obj term}] \ ; \psi_1, a{:}\texttt{obj term} \vdash u[\mathsf{id}_{\psi_1}, a] : \texttt{obj typ}\end{array}$$

The user's contexts control how much we need to split declarations. We now need to compute a set of coverage goals for each of these coverage problems.

### 5.2 Splitting an Object

As a guiding example, let us return to the coverage goal $\psi_1, a{:}\texttt{tp} \vdash u[\mathsf{id}_{\psi_1}, a] : \texttt{tp}$. Because of the existence of canonical forms, the objects of a type are generated by constants from the signature, variables from the context, and parameter variables. Splitting will generate the following set of coverage goals:

$$\begin{array}{lll}(G_1)\ \psi_1{:}\texttt{ctx} & ; \psi_1, a{:}\texttt{tp} \vdash a & : \texttt{tp} \\ (G_2)\ \psi_1{:}\texttt{ctx}, p{::}\texttt{tp}[\psi_1] & ; \psi_1, a{:}\texttt{tp} \vdash p[\mathsf{id}_{\psi_1}] & : \texttt{tp} \\ (G_3)\ \psi_1{:}\texttt{ctx} & ; \psi_1, a{:}\texttt{tp} \vdash \texttt{nat} & : \texttt{tp} \\ (G_4)\ \psi_1{:}\texttt{ctx}, u{::}\texttt{tp}[\psi_1, a{:}\texttt{tp}], v{::}\texttt{tp}[\psi_1, a{:}\texttt{tp}] & ; \psi_1, a{:}\texttt{tp} \vdash \texttt{arr}\ u[\mathsf{id}_{\psi_1}, a]\ v[\mathsf{id}_{\psi_1}, a] & : \texttt{tp} \\ (G_5)\ \psi_1{:}\texttt{ctx}, u{::}\texttt{tp}[\psi_1, a{:}\texttt{tp}, b{:}\texttt{tp}] & ; \psi_1, a{:}\texttt{tp} \vdash \texttt{all}\ \lambda b.u[\mathsf{id}_{\psi_1}, a, b] & : \texttt{tp}\end{array}$$

More formally, we define the generation of objects of contextual type $A[\Psi]$ using the rules below and the rules in Figure 1.

$$\boxed{\Delta; \Psi \vdash \mathsf{Obj}(A) \diamond \mathcal{J}}$$

$$\frac{\Delta; \Psi \vdash \mathsf{Vars}\ (\Psi > P) \diamond \mathcal{J} \quad \begin{array}{c}\Delta; \Psi \vdash \mathsf{Neut}\ (c_1 : \mathsf{S}(c_1) > P) \diamond \mathcal{J} \\ \vdots \\ \Delta; \Psi \vdash \mathsf{Neut}\ (c_n : \mathsf{S}(c_n) > P) \diamond \mathcal{J}\end{array}}{\Delta; \Psi \vdash \mathsf{Obj}(P) \diamond \mathcal{J}}\ \mathsf{Obj\text{-}split}$$

$$\frac{\Delta; \Psi, x{:}A_1 \vdash \mathsf{Obj}(A_2) \diamond \textsc{lam}\triangleright \mathcal{J}}{\Delta; \Psi \vdash \mathsf{Obj}(\Pi x{:}A_1.A_2) \diamond \mathcal{J}}\ \mathsf{Obj\text{-}\Pi} \qquad \frac{\Delta, u{::}P[\Psi]; \Psi \vdash u[\mathsf{id}(\Psi)] : P \diamond \mathcal{J}}{\Delta; \Psi \vdash \mathsf{Obj}(P) \diamond \mathcal{J}}\ \mathsf{Obj\text{-}no\text{-}split}$$

$\mathcal{J}$ is a *continuation* of the following form

$$\mathcal{J} ::= \textsc{covered-by}\ Z \mid \textsc{lam} \triangleright \mathcal{J} \mid \textsc{neut}\ (R : x.B > P) \triangleright \mathcal{J}$$

We start the generation of a coverage goal with $\mathcal{J} = \textsc{covered-by}\ Z$. Once we have built a normal term $M$ of type $A$ we check whether it is an instance of one of the patterns in $Z$; if so, $M$ is covered by the set $Z$. As we build the coverage goal and analyze the type $A$, if $A$ is a function type $\Pi x{:}B_1.B_2$, we extend the continuation with the token LAM; after building an object of type $B_2$, we build a $\lambda$-abstraction. The token NEUT $(R : x.B > P)$ waits for some object $N$, to construct an object $R\ N$ of type $[N/x]B$.

In summary, given a meta-context $\Delta$, a context $\Psi$ and a type $A$, we generate a complete set of coverage goals $G_i = \Delta_i; \Psi_i \vdash M_i : A_i$ and verify that each coverage goal is immediately covered by at least one pattern in $Z$.

Two rules could derive $\psi_1{:}\mathtt{ctx}; \psi_1, a{:}\mathtt{tp} \vdash \mathsf{Obj}(\mathtt{tp}) \diamond \textsc{covered-by}\ Z$. If we use the rule Obj-no-split, we generate the premise

$$\psi_1{:}\mathtt{ctx}, u{::}tp[\psi_1, a : \mathtt{tp}\ ;\ \psi_1, a{:}\mathtt{tp} \vdash u[\mathsf{id}_{\psi_1}, a] : \mathtt{tp} \diamond \textsc{covered-by}\ Z$$

This fails since the goal is not immediately covered by a pattern. Using Obj-split, we produce the leaves: $\{G_1 \diamond \textsc{covered-by}\ Z, \ldots, G_5 \diamond \textsc{covered-by}\ Z\}$

The splitting operation expressed by the rule Obj-split considers each constant $c$ declared in the signature $\mathsf{S}$ and each bound variable from the context to generate an object of type $\mathtt{tp}$. If $\Psi$ contains a context variable, it also considers parameter variables. Our formal system is given in a continuation-based style which facilitates building coverage goals. Our rules leave open the choice between Obj-split and Obj-no-split, and we discuss our strategy subsequently. Constructing a complete set of coverage goals of atomic type $P$ relies on the following three mutually recursive judgments (see Figure 1).

$\Delta; \Psi \vdash \mathsf{Norm}\ M : A \diamond \mathcal{J}$: Given a normal term $M$ of type $A$ in the meta-context $\Delta$ and context $\Psi$, continue with $\mathcal{J}$. Depending on the top (leftmost) part of $\mathcal{J}$, we take a different action: if $\mathcal{J} = \textsc{covered-by}\ Z$, we check if the coverage goal is immediately covered by a pattern in $Z$ (rule Covered-By-$Z$). If $\mathcal{J} = \textsc{lam} \triangleright \mathcal{J}'$, then $\Psi = \Psi', x{:}B$ and we have not finished building $M$, but continue building $\mathsf{Norm}\ \lambda x.M : \Pi x{:}B.A$ with the continuation $\mathcal{J}'$ in the context $\Psi'$ (see rule Normal-Lam). If $\mathcal{J} = \textsc{neut}\ (R : x.B > P) \triangleright \mathcal{J}'$, we have finished building a normal term $M$ that was intended as an argument to a neutral term $R$. Rule Normal-Cont continues, building a neutral term $\mathsf{Neut}\ (R\ M : [M/x]B > P)$; eventually we construct all of $R$'s arguments $N_1, \ldots, N_n$ such that $R\ M\ N_1 \ldots N_n$ has type $P$.

$\Delta; \Psi \vdash \mathsf{Neut}\ (R : A > P) \diamond \mathcal{J}$: Given a neutral object $R$ of type $A$, construct an object $R\ N_1\ \ldots\ N_n$ of type $P$ and continue with $\mathcal{J}$. There are three cases. (1) If $A$ is atomic and unifies with $P$, we have built a relevant neutral object and continue with $\mathcal{J}$ (see App-$\doteq$). (2) If $A$ is atomic but cannot be unified with $P$, the case is impossible and we trivially succeed with rule App-$\not\doteq$. (3) If $R$ has type $\Pi x{:}A.B$, we try to build a normal term $M : A$ and extend the continuation

$$\boxed{\Delta;\Psi \vdash \mathsf{Norm}\ M : A \diamond \mathcal{J}} \qquad \frac{\Delta;\Psi \vdash \mathsf{Norm}\ (\lambda x.\, M) : (\Pi x{:}A_1.A_2) \diamond \mathcal{J}}{\Delta;\Psi, x{:}A_1 \vdash \mathsf{Norm}\ M : A_2 \diamond \textsc{lam}\triangleright\mathcal{J}}\ \textsf{Normal-Lam}$$

$$\frac{\Delta;\Psi \vdash \mathsf{Neut}\ (R\ M : [M/x]B > P) \diamond \mathcal{J}}{\Delta;\Psi \vdash \mathsf{Norm}\ M : A \diamond \textsc{neut}\ (R : x.B > P) \triangleright \mathcal{J}}\ \textsf{Normal-Cont}$$

$$\frac{\Delta;\Psi \vdash M : A \text{ immediately covers } \zeta_k \ \text{ where } \ \zeta_k = \Delta_k;\Psi_k \vdash M_k : A_k}{\Delta;\Psi \vdash \mathsf{Norm}\ M : A \diamond \textsc{covered-by}\ \{\zeta_1, \ldots, \zeta_n\}}\ \textsf{Covered-By-}Z$$

$$\boxed{\Delta;\Psi \vdash \mathsf{Neut}\ (R : A > P) \diamond \mathcal{J}}$$

$$\frac{\Delta;\Psi \vdash Q \not\doteq P}{\Delta;\Psi \vdash \mathsf{Neut}\ (R : Q > P) \diamond \mathcal{J}}\ \textsf{App-}\not\doteq \qquad \frac{\Delta;\Psi \vdash Q \doteq P \ /\ (\theta, \Delta') \quad \Delta';[\![\theta]\!]\Psi \vdash [\![\theta]\!]R : [\![\theta]\!]P \diamond [\![\theta]\!]\mathcal{J}}{\Delta;\Psi \vdash \mathsf{Neut}\ (R : Q > P) \diamond \mathcal{J}}\ \textsf{App-}\doteq$$

$$\frac{\Delta;\Psi \vdash \mathsf{Obj}(A) \diamond \textsc{neut}\ (R : x.B > P) \triangleright \mathcal{J}}{\Delta;\Psi \vdash \mathsf{Neut}\ (R : \Pi x{:}A.B > P) \diamond \mathcal{J}}\ \textsf{App-}\Pi$$

$$\boxed{\Delta;\Psi \vdash \mathsf{Vars}\ (\Psi > P) \diamond \mathcal{J}}$$

$$\frac{\Delta(\psi) = F_1 + \cdots + F_m \qquad \begin{array}{c}\Delta;\Psi \vdash \mathsf{PVars}\ (\psi : F_1 > P) \diamond \mathcal{J} \\ \vdots \\ \Delta;\Psi \vdash \mathsf{PVars}\ (\psi : F_m > P) \diamond \mathcal{J}\end{array}}{\Delta;\Psi \vdash \mathsf{Vars}\ (\psi > P) \diamond \mathcal{J}}$$

$$\frac{\Delta;\Psi \vdash \mathsf{Neut}\ (x : A > P) \diamond \mathcal{J} \quad \Delta;\Psi \vdash \mathsf{Vars}\ (\Phi > P) \diamond \mathcal{J}}{\Delta;\Psi \vdash \mathsf{Vars}\ (\Phi, x{:}A > P) \diamond \mathcal{J}} \qquad \frac{}{\Delta;\Psi \vdash \mathsf{Vars}\ (\cdot > P) \diamond \mathcal{J}}$$

$$\boxed{\Delta;\Psi \vdash \mathsf{PVars}\ (\psi : F > P) \diamond \mathcal{J}}$$

$$\frac{\Delta, \Delta_p, p{::}[\sigma]B[\psi];\Psi \vdash \mathsf{Neut}\ (p[\mathsf{id}_\psi] : [\sigma]B > P) \diamond \mathcal{J} \qquad \begin{array}{l}\Delta_p = u_1{::}A_i[\psi], \ldots u_k{::}A_k[\psi] \\ \sigma = u_1[\mathsf{id}_\psi], \ldots, u_k[\mathsf{id}_\psi]\end{array}}{\Delta;\Psi \vdash \mathsf{PVars}\ (\psi : \mathsf{some}\, x_1{:}A_1, \ldots, x_k{:}A_k\ \mathsf{block}\, B > P) \diamond \mathcal{J}}$$

**Fig. 1.** Coverage checking rules

$\mathcal{J}$ with $\textsc{neut}\ (R : x.B > P)$. Once we have completed the term $M$, we will continue to build the neutral term $R\ M$ of type $[M/x]B$.

$\Delta;\Psi \vdash \mathsf{Vars}\ (\Psi > P) \diamond \mathcal{J}$: Since a neutral object of type $P$ may either be built from a constant from the signature or from a variable from $\Psi$, this judgment generates all variable cases. We will iterate through the context $\Psi$, and for each variable declaration $x : A \in \Psi$, we build a neutral term $\mathsf{Neut}\ (x : A > P)$. If the context $\Psi$ contains a context variable $\psi$, we also need to generate generic variable cases: we look up $\psi$'s context schema $F_1 + \ldots + F_n$, and for each schema element $F_i$, we build a parameter variable using $\mathsf{PVars}\ (F_i : \psi)$. The parameter variable's type can be derived from a schema element $F_i = \mathsf{some}\, x_1{:}A_1, \ldots, x_k{:}A_k\ \mathsf{block}\, B$ as follows: Let $\sigma$ be a substitution instantiating $x_1, \ldots, x_n$ with new meta-variables $u_1[\mathsf{id}_\psi], \ldots, u_n[id_\psi]$. Then the type of the new parameter variable $p$ will be $([\sigma]B)[\psi]$, and we continue building the neutral term $\mathsf{Neut}\ (p[\mathsf{id}_\psi] : [\sigma]B > P)$.

In earlier work [4], we proved the object coverage checking rules sound. This proof can be extended to also prove soundness of the overall coverage algorithm, which includes context splitting.

### 5.3 Implementation

The core of the coverage checker in Beluga can be viewed as a function

$$\mathit{check}(Z, A[\Psi], \mathit{maxContextVarSplit}, \mathit{maxDependentSplit}, \mathit{maxTermSplit})$$

where $Z$ is a set of (contextual) patterns—the guards of the case expression—and $A[\Psi]$ is the type of the expression being cased upon. The *max-* parameters limit the depth of splitting for context variables (Section 5.1), dependent arguments in contexts (Section 5.1), and terms (Section 5.2).

We attack each coverage problem (each case- or let-expression) by making a sequence of calls to *check*: we will first split on the context $\Psi$, and then generate a set of coverage goals for $A$. The parameters *maxContextVarSplit* and *maxDependentSplit* control context splitting. The last parameter *maxTermSplit* controls the depth of the coverage goal.

As mentioned, the rules in Figure 1 leave open the choice of whether to use Obj-no-split or Obj-split. In our implementation, the splitting is guided by a function $\mathit{depth}(M)$, which gives an upper bound on how deeply to split.

$$\begin{aligned}
\mathit{depth}(\lambda x.M) &= \mathit{depth}(M) \\
\mathit{depth}(u[\sigma]) &= 0 \\
\mathit{depth}(c\ M_1\ \ldots\ M_n) &= 1 + \max(\mathit{depth}(M_1), \ldots, \mathit{depth}(M_n)) \\
\mathit{depth}(p[\sigma]\ M_1\ \ldots\ M_n) &= 1 + \max(\mathit{depth}(M_1), \ldots, \mathit{depth}(M_n)) \\
\mathit{depth}(x\ M_1\ \ldots\ M_n) &= 1 + \max(\mathit{depth}(M_1), \ldots, \mathit{depth}(M_n))
\end{aligned}$$

If the user stated that a given type is empty using the keyword impossible, then we initialize the splitting depth to 1.

The overall process is as follows: Follow rule Obj-no-split, generating a meta-variable $u{::}P[\Psi]$, and check if the resulting coverage goal is immediately covered. If not, and the current splitting depth exceeds *maxTermSplit*, fail. Otherwise, increment the current splitting depth, refine type $P[\Psi]$ following the rule Obj-split to generate coverage subgoals, and check that each is covered.

We begin with $\mathit{maxTermSplit} = 0$, which forbids term splitting and makes us use Obj-no-split. This is more useful than it seems: many let-expressions have patterns of this form. If this first call fails, we increment *maxTermSplit* to allow slightly deeper splitting. We continue until we found a covering set or *maxTermSplit* exceeds the depth of the deepest pattern in $Z$. If we found a covering set, coverage succeeds for that case- or let-expression. Otherwise, splitting at the depth of the deepest pattern in $Z$ was not enough to find a covering set, and coverage fails.

There are several additional considerations:

*Subordination* When we generate a meta-variable in the rule Obj-no-split, we allow the meta-variable to depend on the current context $\Psi$, but some declarations in $\Psi$ may never be relevant. Hence, we create the meta-variable in a strengthened context $\Psi'$. To put it differently, the current context $\Psi$ can be obtained by weakening $\Psi'$. Following Virga [21, pp. 55–59], we compute a subordination

relation—a dependency graph of all the types in the signature. For example, if `exp` objects cannot appear in terms of `tp` objects, then the declaration $x$:`exp` is irrelevant when analyzing a `tp` object. Hence, when we create a meta-variable of type `tp` in a context `g, x:exp`, we generate the meta-variable $u$ of type `tp[g]`. The same applies to parameter variables.

*Order of splitting* In applying the coverage rules, we need to choose how deeply to split; we also need to choose the order in which to split arguments.

For a constructor $c\ M_1 \ldots\ M_n$, we split the arguments from right to left. In the dependently typed setting, the leftmost arguments are often index arguments, so splitting on $M_n$ will constrain $M_1$. Beluga allows index arguments to be made implicit and inferred during type reconstruction, so the first arguments may not even be visible to the user. By splitting from right to left, we always begin by splitting on arguments the user wrote. Unfortunately, the order in which we split arguments has an impact beyond performance [20]. Some splits cannot be computed because their unification problems lie outside the decidable pattern unification fragment. Our implementation follows in this regard the good practices implemented in Twelf's coverage checker, but there are subtle differences: for example, given multiple arguments the coverage checker can split on, Twelf prefers arguments whose type is non-recursive (such as the `bool` type above). Splitting on non-recursive types seems "safe" because it always yields a finite number of subcases, and is sometimes necessary.

Delphin splits on the argument that does not occur as part of an index of another splittable argument and that yields the smallest number of subgoals. In Beluga, we simply split from right to left. Compared to other systems, users have more control over what arguments to split where: they can always split off multiple arguments sequentially and even control the splitting depth, and hence the performance, by factoring patterns.

## 6 Conclusion and Future Work

We have used the coverage checker on a wide range of examples: from mechanizing proofs from [16] to proofs from the Twelf repository. We require fewer lemmas to work around some of the limitations found in other systems, which makes the development of proofs more straightforward. While we found simultaneous pattern matching on multiple arguments to be convenient, users often prefer to split arguments sequentially and explicitly prove that cases are impossible. Being able to factor patterns into sequential splits gives the user more control and allows users to be very explicit about individual steps.

There are two improvements we plan to explore: (1) Built-in support for simultaneous patterns: At the moment, users need to first define a pair, and subsequently pattern match on this object. This can be awkward. Built-in support for simultaneous patterns would allow examples to be written more simply. This is largely an engineering question. (2) We plan to implement a refined strategy to calculate the depth of a pattern; currently, given a case expression with

one pattern `[a:tp] arr nat (arr (arr S U) a)`, Beluga will potentially split the first argument of `arr` to a depth of 3, even though only the second argument needs to be split that deeply. This will lead to overall improved performance and more meaningful error messages.

## Acknowledgement

To help us gain a better understanding of the issues around coverage, the following colleagues patiently explained the behavior of their systems: A. Poswolsky for Delphin, members of the Twelf mailing list (in particular K. Crary) for Twelf, A. Abel for Agda. We also would like to thank the reviewer of our IJCAR paper whose comments motivated us to investigate the reasons why Beluga proofs / programs required fewer lemmas.

## References

1. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda—a functional language with dependent types. In: 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09). Lecture Notes in Computer Science, vol. 5674, pp. 73–78. Springer (2009)
2. Coquand, T.: Pattern matching with dependent types. In: Informal Proceedings of Workshop on Types for Proofs and Programs, pp. 71–84. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ. (1992), `http://citeseer.ist.psu.edu/310402.html`
3. Dowek, G.: L'indécidabilité du filtrage du troisième ordre dans les calculs avec types dépendants ou constructeurs de types (The undecidability of third order pattern matching in calculi with dependent types or type constructors). Comptes rendus à l'Académie des Sciences, Série I 312(12), 951–956 (1994)
4. Dunfield, J., Pientka, B.: Case analysis of higher-order data. In: International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08). Electronic Notes in Theoretical Computer Science (ENTCS), vol. 228, pp. 69–84. Elsevier (Jun 2009)
5. Goldfarb, W.D.: The undecidability of the second-order unification problem. Theoretical Computer Science 13, 225–230 (1981)
6. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM 40(1), 143–184 (January 1993)
7. McBride, C.: Dependently Typed Functional Programs and Their Proofs. Ph.D. thesis, University of Edinburgh (2000), `http://citeseer.ist.psu.edu/mcbride99dependently.html`, Technical Report ECS-LFCS-00-419
8. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. Journal of Logic and Computation 1(4), 497–536 (1991)
9. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic 9(3), 1–49 (2008)
10. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology (Sep 2007), Technical Report 33D

11. Pfenning, F., Schürmann, C.: System description: Twelf — a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) Proceedings of the 16th International Conference on Automated Deduction (CADE-16). Lecture Notes in Artificial Intelligence, vol. 1632, pp. 202–206. Springer (1999)
12. Pientka, B.: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08). pp. 371–382. ACM Press (2008)
13. Pientka, B., Dunfield, J.: Programming with proofs and explicit contexts. In: ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08). pp. 163–173. ACM Press (Jul 2008)
14. Pientka, B., Dunfield, J.: Beluga: a framework for programming and reasoning with deductive systems (System Description). In: Giesl, J., Haehnle, R. (eds.) 5th International Joint Conference on Automated Reasoning (IJCAR'10). Lecture Notes in Artificial Intelligence (LNAI) (2010)
15. Pientka, B., Dunfield, J.: Electronic appendix to "Covering all bases: design and implementation of case analysis for contextual objects" (2010), `http://complogic.cs.mcgill.ca/beluga/coverage_appendix`
16. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
17. Poswolsky, A., Schürmann, C.: System description: Delphin—a functional programming language for deductive systems. In: International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08). Electronic Notes in Theoretical Computer Science (ENTCS), vol. 228, pp. 135–141. Elsevier (2009)
18. Poswolsky, A.B., Schürmann, C.: Practical programming with higher-order encodings and dependent types. In: Proceedings of the 17th European Symposium on Programming (ESOP '08). vol. 4960, p. 93. Springer (2008)
19. Schürmann, C.: Automating the Meta Theory of Deductive Systems. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University (2000), CMU-CS-00-146
20. Schürmann, C., Pfenning, F.: A coverage checking algorithm for LF. In: Basin, D., Wolff, B. (eds.) Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03). Lecture Notes in Computer Science, vol. 2758, pp. 120–135. Springer, Rome, Italy (2003)
21. Virga, R.: Higher-Order Rewriting with Dependent Types. Ph.D. thesis, Department of Mathematical Sciences, Carnegie Mellon University (1999), CMU-CS-99-167

# Appendix

## 6.1 Proving determinacy of small-step semantics using simultaneous pattern matching in Beluga

While it is often natural to specify case-splits sequentially, i.e. first splitting on one argument and subsequently splitting on another arguments, being able to split simultaneously on both arguments may lead to more compact proofs.

At this point, Beluga does not support built-in tuples; therefore to split on `(step M N1)[]` and `(step M N2)[]` simultaneously, we first define a new type family `steps_to_terms` indexed by three terms, i.e. `M`, `N1` and `N2`. It has one constructor `andalso` which takes as the first argument the object `step M N1` and as a second argument the object `step M N2`. The determinacy proof is then implemented as a function which takes as input an object of type `steps_to_terms M N1 N2`.

```
steps_to_terms: term → term → term → type .
andalso: step M N1 → step M N2 → steps_to_terms M N1 N2.

rec det : (steps_to_terms M N1 N2)[] → (equal N1 N2)[] =
fn s ⇒ case s of
| [] andalso (s_succ D)          (s_succ F)          ⇒
  let [] ref = det ([] andalso D F) in [] ref

| [] andalso (s_pred_zero)       (s_pred_zero)       ⇒ [] ref
| [] andalso (s_pred_succ _ )    (s_pred_succ _ )    ⇒ [] ref
| [] andalso (s_pred D)          (s_pred F)          ⇒
    let [] ref = det ([] andalso D F) in  [] ref

% Empty cases for pred
| [] andalso (s_pred D)          (s_pred_succ V )    ⇒
   impossible values_dont_step ([] D) ([] v_s V) in []
| [] andalso (s_pred_succ V )       (s_pred D)       ⇒
   impossible values_dont_step ([] D) ([] v_s V) in []
;
```

We note that simultaneously pattern matching leads to an even more compact proof, since Beluga can prove automatically that some combinations are impossible. For example, there is no case for the pattern `[] andalso s_pred_zero (s_pred D)` or `[] andalso (s_pred D) s_pred_zero`. Both cases can be proven automatically to be impossible by our coverage checker.

Adding built-in support for simultaneous patterns so the user would not need to define a specific type-family `steps_to_terms`, is largely an engineering question which we plan to address in the future. Our experience shows that simultaneous patterns are sometime useful, in particular for the expert user; for novices, we found sequential patterns often more convenient and easier to understand why all cases are covered. Sequential patterns model very closely what would happen in an interactive theorem prover or even on paper, where we split on one argument after another.

## 6.2 Proving the determinacy of small-step semantics in Agda

We show the implementation of the function `det` in Agda below. The full implementation in Agda is in the electronic appendix [15]. In Agda, we typically

pattern match simultaneously on multiple inputs. To pattern match on the result of a function call we use a `with` clause.

```
det : {m : Tm} → {n1 : Tm} → {n2 : Tm} →
      Step m n1 → Step m n2 → Eq n1 n2
det (s_succ t)       (s_succ t')        with det t t'
... | ref = ref
det (s_pred_zero)    (s_pred_zero)    = ref
det (s_pred_succ _)  (s_pred_succ _)  = ref
det (s_pred t)       (s_pred t')        with det t t'
... | ref = ref
-- Impossible cases for pred
det (s_pred t)       (s_pred_succ nv)   with values_dont_step t (v_s nv)
... | ()
det (s_pred_succ nv) (s_pred t)         with values_dont_step t (v_s nv)
... | ()
det (s_pred ())      (s_pred_zero)
det (s_pred_zero)    (s_pred ())
```

Absurd patterns, i.e. objects whose type is empty and hence it is impossible to derive such an object, are written using `()`. So for example,

```
det (s_pred_zero)    (s_pred ())
```

describes the case where the first argument matches `s_pred_zero` and the second argument matches `s_pred ()`. The use of an absurd pattern as an argument to the constructor `s_pred` allows us to state explicitly that it is impossible to derive an object of type `step z N2'`.

Let us compare this program with the cases we write in Beluga. We note that some cases are proven automatically in Beluga. For example, in Beluga, we accept the case

```
fn s1 ⇒ fn s2 ⇒ case s1 of
...
| [] s_pred_zero   ⇒  let [] s_pred_zero = s2 in [] ref
```

The reason is the following: to prove that we have covered all cases, coverage will generate possible objects of the type `(step z N2)[]`, called coverage goals, and then check whether each coverage goal is an instance of the patterns specified by the programmer.

Coverage will start with the constructors which could have been used to construct an object `s2` of type `(step z N2)` There are two and it will generate two coverage goals: `[] s_pred_zero` and `[] s_pred F`.

Since we only specified one branch for `s2`, namely the pattern `[] s_pred_zero` , coverage will fail, because not both coverage goals match this single branch. Hence, Beluga will then try to refine the coverage goals further, by splitting `F` which has type `step z N2'`. Coverage will now realize that there is no possible way to ever construct an object `F` of type `step z N2'`, and hence it will remove `[] s_pred F` from the set of coverage goals. It now only needs to show whether the cover goal `[] e_pred_zero` is matched by one of the branches which of course is true.

Comparing the Agda program to the program in Beluga where we write simultaneous patterns, the difference in the approach to coverage and proving cases automatically impossible becomes even clearer.

### 6.3 Proving the determinacy of small-step semantics in Delphin

In this section, we show the proof of determinacy in Delphin.

```
fun det : <step M N1> → <step M N2> → <eq N1 N2> =
fn <s_succ D>  <s_succ F>                ⇒ eq_succ (det <D> <F>)
| <s_pred_zero>       <s_pred_zero>      ⇒ <ref>
| <s_pred_succ V>     <s_pred_succ _ >   ⇒ <ref>
| <s_pred D>          <s_pred F>         ⇒
   eq_pred (det <D> <F>)
| <s_pred D>          <s_pred_succ V >   ⇒
   empty_implies_anything (values_dont_step <D> <v_s V>)
| <s_pred_succ V >    <s_pred D>         ⇒
   empty_implies_anything (values_dont_step <D> <v_s V>)
;
```

Delphin is closely following Twelf. As in Twelf, we observe that we need to prove explicitly congruence lemmas. This has the same cause as in Twelf: were we to return `<ref>` directly, type reconstruction constrains the pattern. This is visible from the type of the offending branch:

```
WARNING:  Match Non-Exhaustive Warning:
[<M : term>]  [<N1 : term>] [<D : step M N1>]
[<N2 : term>] [<F : step M N2>]
((<succ M> and <succ N> and
  <s_succ M N D> and <succ N2> and <s_succ M N2 F>) ⇒    ...)
```

Delphin claims that this more general branch is missing. The reason lies in Delphin's type reconstruction engine. The branch

```
   <s_succ D>  <s_succ F> ⇒ let val <ref> = det <D> <F> in  <ref>  end
```

will be reconstructed such that `D` has type `step M N1` and `F` will have the same type `step M N1`. It may be a bit surprising, since we declared a more general pattern where `D` has type `step M N1` and `F` has type `step M N2`. Only in the body of the branch do we learn that indeed `N1` must be identical to `N2`. Typically, dependently typed functional languages do not allow the pattern to learn more information from the body of a branch, since this goes against the flow of information in a functional program. The motivation behind this design decision has to do with the desire to avoid propagating constraints which are typically generated when pattern matching on dependently typed objects.

This has some unexpected consequences. Functions defined by patterns will refine types and in fact the whole branch, i.e. patterns together with the body is viewed together as a type-reconstruction problem. On the other hand, case-analyzing objects in Delphin will not refine the types. Hence, the following would work:

```
fun values_dont_step : <step M M'> → <value M> → <empty> =
fn <S> <v_s V'> ⇒
    (case <S> of <s_succ S'> ⇒ values_dont_step <S'> <V'>)
| <S> <v_z>    ⇒ step_zero_impossible <S>
;
```

But rewriting this program into a sequential pattern match using case-expression will not type-check.

```
fun step_zero_impossible : <step z M'> → <empty> =
(fn . ) ;

fun values_dont_step : <step M M'> → <value M> → <empty> =
fn <S> <V> ⇒
case <V> of
  <v_s V'> ⇒ (case <S> of <s_succ S'> ⇒ values_dont_step <S'> <V'>)
| <v_z>    ⇒ step_zero_impossible <S>
;
```

However, this will give a type error:

```
Incompatible types: Delphin Unification Failed: Constant clash
   Expected Type: <value (succ X1)>
   Actual   Type: <value z>
```

Once we pattern match `V` which has type `value M` against `v_s V'` which has type `value (succ M')`, the object `M` is fixed to be `succ M'` and since the second pattern `v_z` will have type `value z` instead of `value (succ M')`, Delphin produces a type error.