Coverage checking contextual objects

Brigitte Pientka

McGill University, Montreal, Canada,
bpientka@cs.mcgill.ca

Abstract. We reconsider 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 revisit this question in the setting of contextual objects where objects are closed with respect to a context and contexts are intensional, i.e. their structure can be observed by pattern matching. Our algorithm generalizes and extends prior work on coverage checking of closed LF objects by Pfenning and Schürmann to contextual objects and more generally to datatypes which are indexed by contextual objects. In particular, we describe the design and implementation of a coverage algorithm for Beluga programs, a rich dependently typed programming and proof environment which supports contextual objects and contexts, and prove its correctness. Beluga's coverage algorithm has been used on a wide range of examples, including all the examples from the Twelf repository. Most recently, we have also used the presented algorithm to check coverage of data-types indexed by contextual objects.

1 Introduction

Rich programming and proof environments based on dependent types play an important role when verifying and certifying properties about software systems because they allow us to statically guarantee powerful invariants about the runtime behaviour of the software. To specify rich safety policies, often given by axioms and inference rules, intrinsic support for variable binding, fresh name generation, renaming, capture-avoiding substitution, and representing and manipulating a context of assumptions is convenient and can substantially lower the burden on programmers. Contextual LF [7] which extends the logical framework LF [5] with the power of contexts and contextual objects provides an elegant foundation for these tasks. It supports higher-order abstract syntax encodings as in LF; however, unlike LF, it supports contexts to represent assumptions and packs an "open" LF object M of type A together with the context Ψ in which it is meaningful thereby obtaining a closed contextual object $\hat{\Psi}.M$ of type $A[\Psi]$ where $\hat{\Psi}$ describes the bound variables which M may refer to. $\hat{\Psi}.M$ has the contextual type $A[\Psi]$.

In this paper, we consider the question: Does a set of contextual patterns cover all objects of a given contextual type? The question of coverage is straightforward in the simply-typed setting, but undecidable in the presence of dependent types. The main problem is that knowing when a type is empty is undecidable [6]. Coverage for dependently type objects has been first investigated by

Coquand [3]. Subsequently, coverage of closed LF objects has been investigated by Pfenning and Schürmann [18]. This paper generalizes and extends their work on closed LF objects to contextual objects where objects are closed with respect to a context and contexts are intensional, i.e. their structure can be observed by pattern matching.

Building on earlier work [4] we present a sound theoretical foundation for coverage of contextual objects and contexts. We have designed and implemented a coverage algorithm as part of Beluga [9, 11], a dependently-typed programming and proof environment which supports programming with contextual LF objects. Our coverage algorithm splits incrementally objects 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 [13] to examples we translated from the Twelf repository such as the Church-Rosser theorem for the simply-typed lambda-calculus and cut elimination. Comparing our experience with these examples to other related systems such as Twelf [8] and Delphin [14, 15] shows that many issues are avoided (see [12]). The implementation of the coverage checker is transparent and its performance is competitive. Most recently, we have also employed the presented algorithm to check coverage of data-types indexed by contextual LF objects [2] and we can inspect such indices via pattern matching.

The paper is organized as follows: We show the translation between two different representations for terms both using higher-order abstract syntax in Beluga. Our example exploits pattern matching on contexts, contextual objects and on context relations. Based on this examples, we describe the high-level idea behind coverage. We then introduce the foundation of contextual objects and give a sound theoretical foundation for coverage in this setting. Finally, we describe some of the features of our coverage implementation.

2 Matching on Contexts and Contextual Data

To illustrate the issues arising when modelling formal systems, we consider translating between two different representations for polymorphic lambda-terms. The first representation is our standard specification using `tp` to define types and `tm` to describe terms. We use higher-order abstract syntax to model the binders in our object language, the polymorphic lambda calculus, with binders in the meta-language. The second representation uses a uniform approach for representing the lambda-calculus; 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.

To translate between the two representations, we need to translate objects `obj term` to `tm` and objects `obj typ` to `tp`. As we analyze terms, we traverse lambda-terms and the body of a lambda-term is not necessarily closed any more. Hence our translation in fact translates objects `obj term` in a context ψ containing `obj T`

```
% Types                          % Uniform language
tp  : type .                     obj : term_or_typ → type.
nat' : tp.                       nat: obj typ.
arr' : tp → tp → tp.             arr: obj typ → obj typ → obj typ.
all' : (tp → tp) → tp.           all: (obj typ → obj typ) → obj typ.

% Terms
tm :  type .
lam' : tp → (tm → tm) → tm.      lam : obj typ → (obj term → obj term) → obj term.
app' : tm → tm → tm.             app : obj term → obj term → obj term.
tlam': (tp → tm) → tm.           tlam: (obj typ → obj term) → obj term.
tapp': tm → tp → tm.             tapp: obj term → obj typ → obj term.
```

Fig. 1. Two representations for the polymorphic lambda calculus

declarations where `T` is either `term` or `typ` to `tm` objects in a context ϕ containing `tm` and `tp` declarations. To translate variables in the context ψ to their corresponding variable in ϕ, we also must know that ψ and ϕ are related, i.e. the i-th declaration in ψ corresponds to the i-th declaration in ϕ.

We implement this translation in Beluga [9, 11] is a dependently-typed functional language that supports pattern matching on contexts and contextual LF objects. We call a object `M` in a context ψ a contextual object and write it as `[`ψ `. M]` in our concrete syntax. It has the contextual type `[`ψ`.A]` where `M` has type `A` in the context ψ.

Beluga also provides support for stating the relationship between contexts as an indexed datatype [2] which we use to state the relationship between ψ and ϕ in our running example.

We begin by stating the schema for the two different contexts. `tctx` is the schema for the context ψ, a context containing `tm` and `tp` declarations. `ctx` is the schema for the context ϕ, a context containing declarations of type `obj T` where `T` is either `term` or `typ`.

```
schema tctx = tm + tp ;
schema ctx = some [t:term_or_typ] obj t;
```

We state how a context of schema `tctx` is related to a context `ctx` using an indexed data-type `CtxRel` next. The keyword `ctype` indicates that we are defining an indexed data-type, in contrast to an LF type.

```
datatype CtxRel : {ψ:ctx}{ϕ:tctx} ctype =
| CtxRel_nil : CtxRel [ ] [ ]
| CtxRel_tm : CtxRel [ψ] [ϕ] → CtxRel [ψ, x:obj term] [ϕ, x:tm]
| CtxRel_tp : CtxRel [ψ] [ϕ] → CtxRel [ψ, x:obj typ] [ϕ, x:tp]
;
```

In Figure 2, we implement a recursive function `copy_tp` which states: Given `CtxRel [`ψ`] [`ϕ`]` and `[`ψ`.obj typ]` we produce a corresponding object `[`ϕ`.tp]`. A similar function can be implemented for translating terms.

Several challenges arise in supporting contexts, contextual objects and indexed data-types. To translate variables from the context ψ to their corresponding variables in ϕ, we must observe their position in ψ and map them to their corresponding position in ϕ. The first pattern `[`ψ`, x:obj typ. x]` analyzes

```
rec copy_tp : CtxRel [ψ] [φ]→ [ψ. obj typ] → [φ. tp]=
fn r ⇒ fn m ⇒ case m of
| [ψ . nat] ⇒ let(r' : CtxRel [ψ] [φ]) = r in [φ . nat']

| [ψ, x: obj typ . x]⇒
  (let CtxRel_tp (r:CtxRel [ψ] [φ])= r in [φ,x:tp . x])

| [ψ, x: obj typ . #p ..] ⇒
  let CtxRel_tp  r' = r in
  let [φ . R ..] = copy_tp r' [ψ . #p ..] in
    [φ, x:tp . R ..]

| [ψ, x: obj term . #p ..] ⇒
  let CtxRel_tm r' = r in
  let [φ . R ..] = copy_tp r' [ψ . #p ..] in
    [φ, x:tm . R ..]

| [ψ . arr (T ..) (S ..) ]⇒
  let [φ . R ..] = copy_tp r [ψ . T ..] in
  let [φ . Q ..] = copy_tp r [ψ . S ..] in
    [φ . arr' (R ..) (Q ..)]

| [ψ . all λx. T .. x] ⇒
  let [φ, x:tp . R .. x] = copy_tp (CtxRel_tp r) [ψ, x:obj typ. T .. x] in
    [ φ . all' λx.R .. x]
;
```

Fig. 2. Translating between two HOAS representations for the polymorphic lambda-calculus

the structure of the context. If the variable is the last one in the context ψ, `x:obj typ`, then we know that `r` has type `CtxRel [ψ,x:obj typ] [φ]`. By further pattern matching on `r`, we know that ϕ is indeed of the shape ϕ,`x:tp`, since `r` can only be `CtxRel_tp (r:CtxRel [ψ] [φ])`. By giving a type annotation to the argument `r`, we obtain the name ϕ, the context related to ψ, and return `[φ,x:tp. x]`.

While for a concrete bound variable, we can simply refer to it, as in the pattern `[ψ, x:obj typ. x]`, we also need a case for some other variables from ψ. We use a *parameter variable* `#p` that stands for *some* declaration in ψ to write the case `[ψ,x:obj typ. #p ..]`. We associate the parameter variable `#p` with the identity substitution, written as `..` in the concrete syntax. As a consequence, `#p` has type `[ψ. obj typ]` and we can only instantiate `#p` with a variable declared in the context ψ. Were we to associate `#p` with the identity substitution `.. x`, the type of `#p` would be `[ψ,x:obj typ. obj typ]` and we can instantiate `#p` with any variable declared in the context ψ, `x:obj typ`.

By pattern matching on the context relation `r` we know that ψ, `x:obj typ` is related to ϕ, `x:tp`. We then recursively translate `[ψ.#p ..]` using `CtxRel [ψ] [φ]`.

The remaining cases are straightforward. To translate `[ψ,x:obj tm. λ#p ..]` we proceed as above. To translate `[ψ. arr (T ..) (S ..) ]`, we recursively translate `( T ..)` and `(S ..)`. To translate `[ψ. all λx. T .. x]`, we recursively translate `T .. x` in the extended context ψ,`x:obj typ` and extend the context relation.

To summarize the challenges: we exploit context matching to analyze the shape of the context. Contexts are ordered, we can peel off one declaration at a time until we find the given variable. We pattern match on declarations occurring in contexts. We pattern match on contextual objects, i.e. objects which are closed

with respect to a context which leads us to consider generic variable cases. We rely on higher-order unification. The given example demonstrate that context matching and contextual objects leads to new considerations in coverage.

3 Background: Contextual LF and contexts

Our central question is: Does a set of patterns cover the type `[ψ. A]`? 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 `[ψ.A]` into different cases. For example, in the function `copy_tp` we ensure that the set of patterns

$Z = \{$ `[ψ,x:obj typ. x]`, `[ψ,x:obj typ. #p ..]`, `[ψ,x:obj term. #p ..]`, `[ψ. nat]`, `[ψ. arr (T ..) (S ..)]`, `[ψ. all λx. T .. x]` $\}$

covers all elements of type `[ψ.obj typ]`, i.e. every term of type `[ψ.obj typ]` is an instance of some pattern in Z. To check coverage of a contextual object, we also need to be able to decide when a set of patterns covers a given context schema. More generally, coverage for contextual objects also is at the heart of checking coverage of indexed datatypes such as `CtxRel [ψ] [ϕ]`.

3.1 Contextual LF

Since patterns are derived from an extension of the logical framework LF where we think of every LF object within a context, we first review contextual LF. Contextual types were introduced by Nanevski et al. [7] and subsequently extended and used in [9].

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	σ	::=	$\cdot \mid \sigma\,;\, M \mid \sigma\,,\, R \mid \mathsf{id}_\psi$
Contexts	Ψ, Φ	::=	$\cdot \mid \psi \mid \Psi, x{:}A$

In our theoretical foundation, a contextual object M in a context Ψ is written $[\Psi]M$ and has contextual type $A[\Psi]$. This is in contrast to the concrete syntax used in the examples where we write `[Ψ.M]` which has type `[Ψ.A]`. To precisely define holes in contextual objects, we support meta-variables. For example, in the pattern `[ψ. arr (T ..) (S ..)]` the pattern variables `T` and `S` are meta-variables. Meta-variables are associated with a substitution σ 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 [9]. We characterize only normal forms, since only these are meaningful in LF. We do this by defining *normal terms* M and *neutral terms* R and employing a normalizing hereditary substitution. 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. It is written as

$[*]_A$ where $*$ stands for types, terms, etc. This operation has been defined and proved to be terminating [7, 9, 10]. The syntax guarantees that terms contain no β-redexes, and the typing rules guarantee that well-typed terms are fully η-expanded.

We distinguish between three kinds of variables: *Ordinary bound variables* x and y are bound by λ-abstraction at the LF level; these variables are declared in a context Ψ. *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 σ. The intent is to apply σ as soon as we know the object the contextual variable should stand for. The domain of σ thus includes the free variables of that object, and the type system statically guarantees this.

Substitutions σ 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 σ under a lambda-abstraction $\lambda x.M$ and need to extend the substitution σ with the variable x: If x has a functional type, x is not a well-typed normal term and must be η-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 id_ψ whose domain is a context variable. We write $[\sigma]N$ for substitution application.

Our foundation supports *context variables* ψ 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.

3.2 Meta-objects, Meta-types, Meta-Substitutions, Meta-Contexts

We lift contextual LF objects to meta-types and meta-objects which are embedded into computation-level expressions. This allows a uniform treatment of all meta-objects. Meta-objects are either contextual objects written as $\hat{\Psi}.R$ or contexts Ψ.

Context schemas	G ::=	$\exists(\overrightarrow{x{:}A}).B \mid G + \exists(\overrightarrow{x{:}A}).B$
Meta Types	U ::=	$P[\Psi] \mid \#A[\Psi] \mid G$
Meta Objects	C ::=	$\hat{\Psi}.R \mid \Psi$
Meta substitutions	θ ::=	$\cdot \mid \theta, C/X$
Meta-context	Δ ::=	$\cdot \mid \Delta, X{:}U$

There are three different meta-types: $P[\Psi]$ denotes the type of a meta-variable u and stands for a general contextual object $\hat{\Psi}.R$. $\#A[\Psi]$ denotes the type of a parameter variable p and it stands for a variable object, i.e. either $\hat{\Psi}.x$ or $\hat{\Psi}.p[\pi]$ where π is a variable substitution. A variable substitution π is a special case

for general substitutions σ; however unlike $p[\sigma]$ which can produce a general LF object, $p[\pi]$ guarantees we are producing a variable. Finally, G describes the schema (i.e. type) of a context.

The tag # on the type of parameter variables is a simple syntactic device to distinguish between the type of meta-variables and parameter variables. It does not introduce a subtyping relationship between the type $\#A[\Psi]$ and the type $A[\Psi]$ (as for example in [15]). The meta-context in which an LF object appears uniquely determines if X denotes a meta-variable, parameter variable or context variable. We use the following convention: if X denotes a meta-variable we usually write u or v; if it stands for a parameter-variable, we write p and for context variables we use ψ.

As types classify objects, and kinds classify types, we introduce the notion of *schemas* G that classify contexts Ψ. Context schemas consist of different elements $\exists(\overrightarrow{x{:}A}).B$ which are built using $+$. Intuitively, this means a concrete declaration in a context must be an instance of one of the elements specified in the schema. For example, a context $x{:}\texttt{obj term}, y{:}\texttt{obj typ}$ will check against the schema $\exists T{:}\texttt{term\_or\_typ.obj}\ T$ which was stated as `some [t:term_or_typ] obj t` in the concrete syntax.

The uniform treatment of meta-terms, called C, and meta-types, called U, allows us to give a compact, uniform definition of meta-substitutions θ and meta-contexts Δ. This also simplifies the definitions for coverage.

3.3 Bidirectional Type System

We type contextual 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: $\Delta; \Psi \vdash \sigma \Leftarrow \Phi$. We assume that type constants and object constants are declared in a signature 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.

The typing rules are given in Figure 3.We will tacitly rename bound variables, and maintain that contexts and substitutions declare no variable more than once. Note that substitutions σ are defined only on ordinary variables x, not on contextual variables u or p. We also require the usual conditions on bound variables. This can always be achieved by α-renaming. The typing rules for neutral terms use hereditary substitution $[\cdots]_A$, which preserves canonical forms [7]. Since hereditary substitution is decidable and the rules above are syntax-directed, data-level typing is decidable.

The typing rules for meta-objects and meta-substitutions are straightforward and revert to the contextual LF typing rules. We omit here the rules stating when meta-types and meta-contexts are well-formed and show only the typing rules for meta-term and meta-substitutions.

Neutral Terms $\boxed{\Delta; \Psi \vdash R \Rightarrow A}$

$$\frac{\Psi(x) = A}{\Delta; \Psi \vdash x \Rightarrow A} \qquad \frac{\Delta(p) = \#A[\Phi] \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash p[\sigma] \Rightarrow [\sigma]_\Phi A} \qquad \frac{\Sigma(\mathbf{c}) = A}{\Delta; \Psi \vdash \mathbf{c} \Rightarrow A}$$

$$\frac{\Delta(u) = P[\Phi] \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash u[\sigma] \Rightarrow [\sigma]_\Phi P} \qquad \frac{\Delta; \Psi \vdash R \Rightarrow \Pi x{:}A.B \quad \Delta; \Psi \vdash M \Leftarrow A}{\Delta; \Psi \vdash R\, M \Rightarrow [M/x]_A B}$$

Normal Terms $\boxed{\Delta; \Psi \vdash M \Leftarrow A}$

$$\frac{\Delta; \Psi \vdash R \Rightarrow P \quad P = Q}{\Delta; \Psi \vdash R \Leftarrow Q} \qquad \frac{\Delta; \Psi, x{:}A \vdash M \Leftarrow B}{\Delta; \Psi \vdash \lambda x.\, M \Leftarrow \Pi x{:}A.B}$$

Substitutions $\boxed{\Delta; \Psi \vdash \sigma \Leftarrow \Psi'}$

$$\frac{}{\Delta; \Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \Delta; \Psi \vdash R \Rightarrow B \quad B = [\sigma]_\Phi A}{\Delta; \Psi \vdash \sigma; R \Leftarrow \Phi, x{:}A}$$

$$\frac{}{\Delta; \psi, \Psi \vdash \mathsf{id}_\psi \Leftarrow \psi} \qquad \frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \Delta; \Psi \vdash M \Leftarrow [\sigma]_\Phi A}{\Delta; \Psi \vdash \sigma, M \Leftarrow \Phi, x{:}A}$$

Meta Terms $\boxed{\Delta \vdash C \Leftarrow U}$

$$\frac{}{\Delta \vdash \cdot \Leftarrow G} \qquad \frac{\Delta(\psi) = G}{\Delta \vdash \psi \Leftarrow G} \qquad \frac{\Delta \vdash \Psi \Leftarrow G \quad \exists(\overrightarrow{x : B'}).B \in G \quad \Delta; \Psi \vdash \sigma \Leftarrow \overrightarrow{(x{:}B')} \quad A = [\sigma]_{\overrightarrow{(x:B')}} B}{\Delta \vdash \Psi, x{:}A \Leftarrow G}$$

$$\frac{\Delta; \Psi \vdash R \Leftarrow P}{\Delta \vdash \hat{\Psi}.R \Leftarrow P[\Psi]} \qquad \frac{\Psi(x) = A}{\Delta \vdash \hat{\Psi}.x \Leftarrow \#A[\Psi]} \qquad \frac{\Delta(p) = \#A[\Phi] \qquad \Delta; \Psi \vdash \pi \Leftarrow \Phi \quad [\pi]_\Phi(A) = B}{\Delta \vdash \hat{\Psi}.p[\pi] \Leftarrow \#B[\Psi]}$$

Fig. 3. Typing for contextual LF and meta-objects

3.4 Meta-contexts and meta-substitutions

We adopt here the notion that meta-contexts are a set of meta-declarations, since it simplifies the subsequent theoretical development of coverage (see also [16, 1]. For the typing rules to be meaningful, the set cannot be cyclic.

Meta-Substitutions $\boxed{\Delta \vdash \theta : \Delta'}$

$$\frac{\text{for all } X{:}U \in \Delta' \text{ and } C/X \in \theta \quad \Delta \vdash C \Leftarrow [\![\theta]\!]U}{\Delta \vdash \theta : \Delta'}$$

The single meta-substitution, written as $[\![C/X]\!]_U(*)$ where $*$ stands for A, M, R, σ, Ψ, is defined inductively on the structure of the given object. The typing rules ensure that the type of the instantiation C and the type of X agree. We note that we can always appropriately rename bound variables in C, if necessary. We similarly write $[\![\theta]\!](*)$ for the simultaneous meta-substitution.

We only discuss briefly here some of the fundamental ideas. Let us first consider the case where X stands for a meta-variable u and C is a meta-object $\hat{\Psi}.R$. We note that there is no capture issues when we push $[\![\hat{\Psi}.R/u]\!]$ through a lambda-expression and the only interesting issue arises when we encounter an object $u[\sigma]$. In this case, we apply $[\![\hat{\Psi}.R/u]\!]$ to σ to obtain σ'. Subsequently, we apply σ' to R to obtain the final result.

Next, we consider the case where X stands for a parameter variable p and C is a meta-object $\hat{\Psi}.x$ or $\hat{\Psi}.q[\pi]$. The only interesting case is when we encounter $p[\sigma]$. Similar to the case for meta-variables, we apply the meta-substitution to σ to obtain σ' and subsequently apply σ' to x or $q[\pi]$. There is however a small caveat: since σ' is an arbitrary substitution, applying it to x, may yield a normal object M. Hence, simply returning M may produce a non-normal term which is not meaningful in our grammar. The solution to this problem is to define meta-substitutions hereditarily (see [10]).

Finally, the case where X stands for a context variable ψ and C is a meta-object Ψ. There are two interesting cases: 1) when we encounter the identity substitution id_ψ, we unroll Ψ and create at the same time a concrete identity substitution which maps all variables from Ψ to themselves. 2) when we encounter a context variable ψ in a context, then we simply replace it with the concrete context Ψ. The full definition of meta-substitutions has been previously been described in [7, 9].

4 Coverage Checking

A *coverage goal* is a meta-object C that can stand for a contextual object $\hat{\Psi}.R$ or a context Ψ. We write $\Delta \vdash C : U$ for the coverage goal. Intuitively, a coverage goal represents all of its closed instances which have type U.

A *coverage problem* consists of the coverage goal and a set of patterns. In Beluga, this set of patterns comes from the program. 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 [3] and Schürmann and Pfenning [18] describes coverage checking for closed terms. While Schürmann [17, pp. 197–213] formulated coverage for open terms within regular worlds, the foundation lacks a general theoretical foundation for context variables and context matching. We generalize the previous notion of coverage to directly support contextual objects, parameter variables, explicit contexts and context matching.

Definition 1 (Immediate coverage). *A coverage goal $\Delta \vdash C : U$ is* immediately covered *by a collection of patterns $\Delta_i \vdash C_i : U_i$ if there exist i and a meta-substitution θ such that $\Delta \vdash \theta : \Delta_i$ and $[\![\theta]\!]C_i = C$ and $[\![\theta]\!]U_i = U$.*

Definition 2 (Coverage). *A coverage goal $\Delta \vdash C : U$ is* covered *by a set of patterns $\Delta_i \vdash C_i : U_i$ if every ground instance $\cdot \vdash [\![\theta]\!]C : [\![\theta]\!]U$, where $\cdot \vdash \theta : \Delta$, is immediately covered by some pattern $\Delta_i \vdash C_i : U_i$.*

Definition 3 (Non-redundant complete set of meta-substitutions). *Let $\Delta \vdash C : U$ be a coverage goal. We say a finite collection $\Delta_i \vdash \rho_i : \Delta$ is a non-redundant set of meta-substitutions if for every $\cdot \vdash \rho : \Delta$ there exists a unique i and a unique $\cdot \vdash \theta_i : \Delta_i$ s.t. $\rho = [\![\theta_i]\!]\rho_i$.*

The idea of coverage checking can then be described as follows: We begin by generating a general coverage goal $\Delta \vdash X : U$ and check whether the coverage goals is immediately covered by a set of patterns. If it is not, we choose a variable from Δ (typically $X : U$ in the first step) and compute a set of splitting (refinement) substitutions ρ_i s.t. $\Delta_i \vdash \rho_i : \Delta$. For each i, we check whether $\Delta_i \vdash [\![\rho_i]\!]X : [\![\rho_i]\!]U$ is immediately covered. If it is not, we again choose a variable $X : U$ from Δ and compute a set of splitting (refinement) substitutions and proceed as previously. Unlike our previous formulation [4] coverage checking becomes an iterative process and a natural extension of previous algorithms [18].

Theorem 1. *Let $\Delta \vdash C : U$ be a coverage goal and $\Delta_i \vdash \rho_i : \Delta$ be a non-redundant complete collection of meta-substitutions. All $\Delta_i \vdash [\![\rho_i]\!]C : [\![\rho_i]\!]U$ are covered by a given set of patterns if and only if $\Delta \vdash C : U$ is covered.*

Proof. Coverage depends only on ground instances - but the collection of ground instances is exactly the same.

5 Splitting

At the heart of the coverage is the splitting operation. It defines how to refine a meta-object $X : U$ based on the meta-type U. It can not only be used as part of a coverage algorithm to ensure that a given set of patterns covers, but also to generate cases automatically in an interactive fashion. The three different meta-types give rise to three different splitting operations.

5.1 Splitting a context variable

Given a context variable ψ : `ctx` where `schema ctx = some [t:term_or_typ] obj t`, we generate two coverage goals:

$$\begin{array}{rl} [\ \psi : \texttt{ctx}, u : \texttt{term\_or\_typ}[\psi] \vdash [\phi, x : \texttt{obj}\ u[\mathsf{id}_\psi]\ ] & : \texttt{ctx}, \\ \vdash [\quad \cdot \quad] & : \texttt{ctx}\ ] \end{array}$$

We generate the most general declaration from the given schema `ctx`. The context can then be refined further by for example splitting on `u`.

Definition 4 (Splitting a context variable $\psi : G$). *Let $\Delta \vdash C : U$ be a coverage goal and $\Delta = \Delta_1, \psi : G, \Delta_2$. The set $\mathcal{R}$ of splitting substitutions is generated as follows.*

1. *$[\![\cdot/\psi]\!](\Delta_1, \Delta_2) \vdash [\![\cdot/\psi]\!]\mathsf{id}(\Delta) : \Delta$ is in $\mathcal{R}$.*
2. *For each $\exists(\overrightarrow{x{:}A}).B \in G$, let $\rho = \phi, x : [\overrightarrow{u[\mathsf{id}_\phi]/x}]B/\psi$ and $\Delta_i = \phi : G, \overrightarrow{u : A[\phi]}, [\![\rho]\!](\Delta_1, \Delta_2)$. Then $\Delta_i \vdash [\![\rho]\!]\mathsf{id}(\Delta) : \Delta$ is in $\mathcal{R}$.*

5.2 Splitting a meta-variable

Given a meta-variable $u : P[\Psi]$, splitting needs to generate all possible meta-objects which have type P in the context Ψ. Since P is atomic, we concentrate on generating neutral objects. We first synthesize a set $\mathcal{H}$ of all possible heads together with their type. Intuitively the set $\mathcal{H}$ contains all constants defined in the signature and all variables occurring in Ψ. If Ψ contains a context variable ψ of schema G, then we generate also parameter variables using the schema G.

Definition 5 (Generating a set of heads). *Given meta-context Δ and a local context Ψ, we generate a set $\mathcal{H}$ as follows:*

1. *For each constant $c : A$ in the signature, $\boxed{\Delta;\Psi \vdash c : A}$ is in $\mathcal{H}$.*
2. *For each bound variable $x : A$ in Ψ, $\boxed{\Delta;\Psi \vdash x : A}$ is in $\mathcal{H}$.*
3. *If $\Psi = \psi, \overrightarrow{x{:}A}$ and $\psi : G \in \Delta$, then for each $\exists\overrightarrow{x{:}B'}.B \in G$,*

   $\boxed{\Delta, \overrightarrow{u{:}B'[\psi]}, \#p{:}([\overrightarrow{u[\mathsf{id}_\psi]/x}]B)[\psi] \ ; \ \Psi \ \vdash \ p[\mathsf{id}_\psi] : [\overrightarrow{u[\mathsf{id}_\psi]/x}]B}$ *is in $\mathcal{H}$.*

Using a head h of type A from the set $\mathcal{H}$, we then generate the most general neutral term whose type is unifiable with P in the context Ψ. We describe unification using the judgment $\Delta;\Psi \vdash Q \doteq P \ / \ (\Delta' \ , \ \theta)$. If unification succeeds then $[\![\theta]\!]Q = [\![\theta]\!]P$ and $\Delta' \vdash \theta : \Delta$. The generation of a neutral term R' is accomplished by the judgement $\boxed{\Delta;\Psi \vdash \mathsf{neut}\ (R) : A \Leftarrow P \ / \ (\Delta' \ , \ \theta \ , \ R')}$ where all the elements on the left side of $/$ are inputs and the right side is the output. The following holds: $\Delta' \vdash \theta : \Delta$ and $\Delta'; [\![\theta]\!]\Psi \vdash R' \Leftarrow [\![\theta]\!]P$.

$$\frac{\Delta, u{:}A[\Psi];\Psi \vdash \mathsf{neut}\ (R\ u[\pi]) : [u[\pi]/x]B \Leftarrow P \ / \ (\Delta' \ , \ \theta \ , \ R')}{\Delta;\Psi \vdash \mathsf{neut}\ (R) : \Pi x{:}A.B \Leftarrow P \ / \ (\Delta' \ , \ \theta \ , \ R')} \text{ where } \pi = \mathsf{id}(\Psi)$$

$$\frac{\Delta;\Psi \vdash Q \doteq P/(\Delta', \theta)}{\Delta;\Psi \vdash \mathsf{neut}\ (R) : Q \Leftarrow P \ / \ (\Delta' \ , \ \theta \ , \ [\![\theta]\!]R)}$$

Intuitively, we start generating a neutral term with $h : A$. As we recursively analyze A, we generate all the arguments h is applied to until we reach an atomic type Q. If Q unifies with the expected type P, then generating a most general neutral term with head h succeeds.

Definition 6 (Splitting a meta-variable). *. Let $\Delta \vdash C : U$ be a coverage goal and $\Delta = \Delta_1, u : P[\Psi], \Delta_2$. The set $\mathcal{R}$ of splitting substitutions is generated as follows.*

1. *Given the meta-context Δ and the local context Ψ, generate a set $\mathcal{H}$ using Definition 5.*
2. *For each head $\Delta_i;\Psi \vdash h_i : A_i \ \in \ \mathcal{H}$,*
   if $\Delta_i;\Psi \vdash \mathit{neut}\ (h) : A \Leftarrow P \ / \ (\Delta' \ , \ \theta_i \ , \ R_i)$ and $\Delta' = \Delta'_1, u{:}[\![\theta_i]\!](P[\Psi]), \Delta'_2$
   then $\boxed{[\![\hat{\Psi}.R_i/u]\!](\Delta'_1, \Delta'_2) \vdash [\![\hat{\Psi}'.R_i/u]\!]\theta_i : \Delta_i}$ is a splitting substitution in $\mathcal{R}$.

5.3 Splitting a parameter-variable

We show how to generate all variables of type $\#A[\Psi]$. Intuitively, only bound variables $x : B$ from Ψ whose type is unifiable with A inhabit this type and if the context Ψ contains a context variable $\psi : G$ we also include all parameter variables of the appropriate type synthesized from G. We only need to be careful to generate an object which is size-decreasing such that coverage will eventually terminate. We should only generate parameter variables of type $[\![\theta]\!](B[\psi])$ where $[\![\theta]\!]B = [\![\theta]\!]A$, if Ψ contains in addition to $\psi : G$ also other bound variable declarations. This guarantees that the type of the new parameter variable is smaller than the old one.

Definition 7 (Splitting a parameter-variable). . *Let $\Delta \vdash C : U$ be a coverage goal and $\Delta = \Delta_1, \#p : A[\Psi], \Delta_2$. The set $\mathcal{R}$ of splitting substitutions is generated as follows.*

Case: $\Psi = \psi, \overrightarrow{x{:}B}$ *where* $\overrightarrow{x{:}B}$ *is non-empty and* $\psi : G \in \Delta_1$

i For each $\exists(\overrightarrow{x{:}B'}).B \in G$. *If* $\Delta, \overrightarrow{u : B'[\psi]} \vdash [\overrightarrow{u[\mathsf{id}_\psi]/x}]B \doteqdot A \;/\; (\Delta' \;,\; \theta)$ *where* $\Delta' = \Delta'_1, \#p{:}[\![\theta]\!](A[\Psi]), \Delta'_2$ *and* $\Delta' \vdash \theta_0 : \Delta$ *s.t.* $\theta_0 \subset \theta$, *then* $\boxed{[\![\psi.q[\mathsf{id}_\psi]/p]\!](\Delta'_1, \Delta'_2), \#q : [\![\theta_0]\!](A[\psi]) \vdash [\![\psi.q[\mathsf{id}_\psi]/p]\!]\theta_0 : \Delta}$ *is in the set* $\mathcal{R}$.

ii For each declaration $x{:}B$. *If* $\Delta; \Psi \vdash B \doteqdot A \;/\; (\Delta' \;,\; \theta)$ *where* $\Delta' = \Delta'_1, \#p{:}[\![\theta]\!](A[\Psi]), \Delta'_2$ *and* $\Delta' \vdash \theta : \Delta$ *then* $\boxed{[\![\hat{\Psi}.x/p]\!](\Delta'_1, \Delta'_2) \vdash [\![\hat{\Psi}.x/p]\!]\theta : \Delta}$ *is in the set* $\mathcal{R}$.

Case : $\Psi = \overrightarrow{x{:}B}$ *where* $\overrightarrow{x{:}B}$ *is possibly empty For each declaration* $x{:}B$. *If* $\Delta; \Psi \vdash B \doteqdot A \;/\; (\Delta' \;,\; \theta)$ *where* $\Delta' = \Delta'_1, \#p{:}([\![\theta]\!]A[\Psi]), \Delta'_2$ *and* $\Delta' \vdash \theta : \Delta$ *then* $\boxed{[\![\hat{\Psi}.x/p]\!](\Delta'_1, \Delta'_2) \vdash [\![\hat{\Psi}.x/p]\!]\theta : \Delta}$ *is in the set* $\mathcal{R}$.

5.4 Theoretical properties of splitting

Theorem 2 (Splitting). *The set of meta-substitutions generated by splitting is non-redundant and complete.*

Proof. From the properties of unification and our definition of contextual objects. The sets generated are also complete (see the appendix for the proof).

6 Implementation of coverage algorithm

The coverage algorithm proceeds as follows:

1. Check that a coverage goal $\Delta \vdash C : U$ is immediately covered, i.e. there exists an i and a meta-substitution $\Delta \vdash \theta_i : \Delta_i$ s.t. $\Delta \vdash C = [\![\theta_i]\!]C_i : U$ and $U = [\![\theta_i]\!]U_i$ where $\Delta_i \vdash C_i : U_i$ is a given pattern

2. If immediate coverage fails, pick a variable $X : U$ from Δ and split it. Splitting returns a collection of splitting substitutions $\Delta_k \vdash \rho_k : \Delta$ which applied to C give us a a collection of new coverage goals $\Delta_k \vdash [\![\rho_k]\!]C : [\![\rho_k]\!]U$ all of which must be immediately covered.

Given that splitting generates a non-redundant most general set of refinement substitutions, coverage is sound. Following Pfenning and Schürmann [18] we record why coverage fails to generate counter examples. To accomplish this we employ a pre-matching algorithm between C (the coverage goal) and C_i (the pattern) which is inspired by Huet's simplification phase in higher-order unification algorithm and is similar to first-order matching. Our algorithm handles matching of normal terms and contexts and produces two sets of equations. The set $\mathcal{E}$ contains equations of the form $C = \hat{\Psi}.u[\sigma]$, $C = \psi$ or $C = \hat{\Psi}.\#p[\pi]$ where u, ψ and $\#p$ are variables occurring in the pattern and C is the coverage goal. The set $\mathcal{S}$ contains equations of the form $\hat{\Psi}.u[\sigma] = C$, $\psi =$ or $\hat{\Psi}.\#p[\sigma] = C$ where u, ψ, $\#p$ are variables occurring in the coverage goal and C is a sub-expression of the given pattern. Splitting equations $\mathcal{S}$ arise from the failure of the pre-matching algorithm but we can possibly make further progress by splitting on u, ψ, or $\#p$. The algorithm is a natural extension of [1].

For coverage to succeed, the set of splitting equations $\mathcal{S}$ must be empty and all equations in $\mathcal{E}$ must be solved by higher-order unification. There are several additional considerations:

Subordination When we generate a neutral term, we allow the meta-variable to depend on the current context Ψ, but some declarations in Ψ may never be relevant. Hence, we create the meta-variable in a strengthened context Ψ'. To put it differently, the current context Ψ can be obtained by weakening Ψ'. Following Virga [19, pp. 55–59], we compute a subordination relation—a dependency graph of all the types in the signature. For example, if `tm` objects cannot appear in terms of `tp` objects, then the declaration x:`tm` is irrelevant when analyzing a `tp` object. Hence, when we create a meta-variable of type `tp` in a context `g, x:tm`, we generate the meta-variable u of type `[g.tp]`. The same applies to parameter variables.

Order of splitting During coverage, we need to choose a variable from Δ to split on. Unfortunately, the order in which we split arguments has an impact beyond performance [18]. Some splits cannot be computed because their unification problems lie outside the decidable pattern unification fragment. We first split on context variables, if any context variable occurs in a splitting equation. If there are none, we choose the meta-variable with the lowest de Bruijn index. 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.

Proving trivially impossible cases In Beluga, we decided to try and prove that some generated counter-examples are impossible. After we are done checking for coverage, we revisit the open coverage goals which were not so far covered, and attempt to show that there is at least one variable for which no element exists, i.e. its type is empty. Checking in general whether a type is empty is undecidable, but we use a simple heuristic where we try to generate possible splits for a given variable. If this fails, then we know that there is no inhabitant for it. The same strategy is used in Twelf. For some examples, this strategy is very important in practice. In proving soundness of a evaluation under continuations, proving cases to be impossible, is vital for a compact specification. We prove 20-21 cases to be impossible 6-times! We typically write out only two relevant cases, while 20 others are proven trivially impossible. Similarly in proving determinacy of a small-step semantics containing arithmetic (TAPL Ch3), we can prove 8 cases to be trivially impossible.

In our implementation, the user can write out these cases explicitly, but the programmer does not have to. For convenience we support proving some cases trivially.

7 Conclusion and Future Work

We have used the presented coverage algorithm on a wide range of examples: from mechanizing proofs from [13] to proofs from the Twelf repository. Compared to other systems such as Twelf and Delphin, our ability to describe contextual objects and contexts explicitely requires fewer lemmas to work around some of the limitations of coverage checkers found in other systems. This makes the development of proofs more straightforward (for a deeper analysis see [12]). Most recently, we have also extended Beluga with indexed data-types and incorporated the coverage algorithm for contextual objects into a more general coverage algorithm for data-types. In the future, we plan to use the algorithm to generate case-splits interactively when the user desires and hence support interactive program development

References

1. Abel, A., Pientka, B.: Higher-order dynamic pattern unification for dependent types and records. In: Ong, L. (ed.) 10th International Conference on Typed Lambda Calculi and Applications (TLCA'11). pp. 10–26. Lecture Notes in Computer Science (LNCS 6690), Springer (2011)
2. Cave, A., Pientka, B.: Programming with binders and indexed data-types. In: 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12). p. to appear. ACM Press (2012)

3. 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)
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. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM 40(1), 143–184 (January 1993)
6. McBride, C.: Dependently Typed Functional Programs and Their Proofs. Ph.D. thesis, University of Edinburgh (2000), Technical Report ECS-LFCS-00-419
7. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic 9(3), 1–49 (2008)
8. Pfenning, F., Schürmann, C.: System description: Twelf — a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) 16th International Conference on Automated Deduction (CADE-16). Lecture Notes in Artificial Intelligence, vol. 1632, pp. 202–206. Springer (1999)
9. 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)
10. 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)
11. 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). pp. 15–21. Lecture Notes in Artificial Intelligence (LNAI 6173), Springer-Verlag (2010)
12. Pientka, B., Dunfield, J.: Covering all bases: design and implementation of case analysis for contextual objects. Tech. rep., McGill University (2010)
13. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
14. 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)
15. Poswolsky, A.B., Schürmann, C.: Practical programming with higher-order encodings and dependent types. In: 17th European Symposium on Programming (ESOP '08). vol. 4960, pp. 93–107. Springer (2008)
16. Reed, J.: Higher-order constraint simplification in dependent type theory. In: Felty, A., Cheney, J. (eds.) International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'09) (2009)
17. Schürmann, C.: Automating the Meta Theory of Deductive Systems. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University (2000), CMU-CS-00-146
18. 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). pp. 120–135. Springer (2003)
19. Virga, R.: Higher-Order Rewriting with Dependent Types. Ph.D. thesis, Department of Mathematical Sciences, Carnegie Mellon University (1999), CMU-CS-99-167

Appendix

Theorem 3 (Splitting). *The set of meta-substitutions generated by splitting is non-redundant and complete.*

Proof. From the properties of unification and our definition of contextual objects. The set generated from splitting a context variable is obviously non-redundant. The set generated from splitting a meta-variable is non-redundant since all generated neutral terms have distinct heads. The set generated from splitting a parameter variable is non-redundant, since all declarations in a context Ψ are distinct.

The sets generated are also complete (see the appendix for the proof). We consider here the three cases of meta-objects.

Case Splitting on a meta-variable $u{:}P[\Psi] \in \Delta$. We need to show that all closed canonical objects C of type $P[\Psi]$ are covered by the generated splits. Since C is normal, we know $C = \hat{\Psi}.R$ and $R = h\ M_1 \dots M_n$ s.t. $\cdot \vdash \theta : \Delta$ and $\cdot; [\![\theta]\!]\Psi \vdash R \Leftarrow [\![\theta]\!]P$. The set $\mathcal{H}$ is complete and for all heads h we have $h{:}A \in \mathcal{H}$. Moreover, by the properties of unification, $\Delta; \Psi \vdash \mathsf{neut}\ (h) : A \Leftarrow P\ /\ (\Delta'\ ,\ \theta'\ ,\ R')$ generates the most general R' s.t. $\Delta'; [\![\theta']\!]\Psi \vdash R' \Leftarrow [\![\theta']\!]P$. Therefore there exists a meta-substitution ρ s.t. $\cdot \vdash \rho : \Delta'$ and $[\![\rho]\!]([\![\theta']\!](P[\Psi]) = [\![\theta]\!](P[\Psi])$ and $[\![\rho]\!]R' = R$.

Case Splitting on a context variable $\psi{:}G \in \Delta$. We need to show that all closed canonical objects C of type G are covered by the generated splits. Since C is normal, it stands for a concrete context which is either empty or $\Psi = x_1{:}B_1, \dots, x_n{:}B_n$ s.t. $\cdot \vdash \Psi : G$. Our splitting definition generates the most general declarations which are instances of the schema G, i.e. for all $\exists \overrightarrow{x{:}A}.B \in G$, we generate $\Psi' = \psi, x : [\overrightarrow{u[\mathsf{id}_\psi]/x}]B$ s.t. $\psi : G, \overrightarrow{u{:}A[\psi]} \vdash \Psi' : G$. Since it is most general, there exists a meta-substitution ρ s.t. $\cdot \vdash \rho\ :\ \psi{:}G, \overrightarrow{u{:}A[\psi]}$ s.t. $[\![\rho]\!]\Psi' = \Psi$.

Case Splitting on a parameter variable $\#p : A[\Psi]$. We need to show that all closed canonical objects C of type $\#[\![\theta]\!](A[\Psi])$ where $\cdot \vdash \theta : \Delta$ are covered by the generated splits. Since C is canonical, it must be of the form $C = x_1, \dots, x_n.x_i$ where $1 \leq i \leq n$ and $\cdot; [\![\theta]\!]\Psi \vdash x_i \Rightarrow A'$ s.t. $[\![\theta]\!]A_i = A'$. We distinguish two cases. If $\Psi = x_1{:}A_1, \dots x_n{:}A_n$, then x_i has type A_i and our splitting algorithm guarantees that there exists a most general meta-substitution $\Delta' \vdash \theta' : \Delta$ s.t. $[\![\theta']\!]A_i = [\![\theta']\!]A$. Since θ' is most general, there exists a grounding meta-substitution ρ s.t. $\cdot \vdash \rho : \Delta'$ and $[\![\rho]\!]([\![\theta']\!]A_i) = A' = [\![\theta]\!]A$.

If $\Psi = \psi, x_i{:}A_i, \dots, x_n{:}A_n$, we also need to consider the case where our algorithm generates for all $\exists \overrightarrow{x{:}A}.B \in G$, $\psi : G, \overrightarrow{u{:}A[\psi]}, \#p : B[\psi] \vdash \psi.p[\mathsf{id}_\psi] \Rightarrow B$, if B unifies with A. As a consequence there is a most general meta-substitution $\Delta' \vdash \theta : \Delta, \psi : G, \overrightarrow{u{:}A[\psi]}, \#p : B[\psi]$. To generate closed instances of the form $x_1, \dots x_n.x_k$ where $1 \leq k < i$, we instantiate ψ with $x_1{:}A_1, \dots x_k{:}A_k$ and p with $x_1, \dots x_{i-1}.x_k$.