Case analysis of higher-order data

Joshua Dunfield and Brigitte Pientka

School of Computer Science, McGill University, Montreal, Canada
{joshua,bpientka}@cs.mcgill.ca

Abstract. We discuss coverage checking for data that is dependently typed and is defined using higher-order abstract syntax. Unlike previous work on coverage checking that required objects to be closed, we consider open data objects, i.e. objects that may depend on some context. Our work may therefore provide insights into coverage checking in Twelf, and serve as a basis for coverage checking in functional languages such as Delphin and Beluga. More generally, our work is a foundation for proofs by case analysis in systems that reason about higher-order abstract syntax.

1 Introduction

Over the past decade, programming and reasoning with and about data structures that contain binders has received widespread attention in programming languages as well as automated reasoning systems. One simple and elegant technique for handling binders is higher-order abstract syntax. The central idea is easily explained: instead of representing object variables explicitly, we use meta-language variables. For example, the object-level formula $\forall x.\,(x = 1) \supset \neg(x = 0)$ can be represented as `forall λx. (eq x (Suc Zero)) imp (not (eq x Zero))`. One of the key benefits is that one can avoid implementing common and tricky routines dealing with variables, such as capture-avoiding substitution, renaming and fresh name generation. When we implement proofs, higher-order abstract syntax allows us to think of hypothetical derivations, i.e. derivations that depend on assumptions as higher-order functions, where the application of a substitution lemma corresponds to a function application. For example, in natural deduction, the hypothetical typing derivation for implication introduction can be elegantly modeled using higher-order functions (see Fig. 1).

The power of higher-order abstract syntax encodings has been demonstrated within the logical framework LF [4] and its implementation in the Twelf system [9]. Most recently, HOAS encodings are supported in functional programming languages such as Elphin [15], Delphin [12], and Beluga [11]. In these systems, we typically analyze higher-order data using pattern matching and case expressions. This requires us to validate that all possible cases are covered, i.e. the patterns are exhaustive. A closely related question arises in proof assistants that support reasoning about HOAS specifications when we split a goal into different cases. In this situation we also must generate an exhaustive set of cases. This issue arises in the Twelf system's induction theorem prover [13], and in systems such as Bedwyr [1] and Abella [3].

$$\begin{array}{rll} \text{Numbers} & N, M ::= & x \\ & & \mid 0 \\ & & \mid \mathsf{suc}\ N \end{array}$$

$$\begin{array}{rll} \text{Propositions} & A ::= & N = M \\ & & \mid A \supset B \\ & & \mid \forall x.A \end{array}$$

Natural Deduction : $\Gamma \vdash \mathsf{nd}\ A$

$$\frac{\Gamma, u : \mathsf{nd}\ A \vdash \mathsf{nd}\ B}{\Gamma \vdash \mathsf{nd}\ A \supset B}\ \supset I^u$$

$$\frac{\Gamma \vdash \mathsf{nd}\ A \supset B \qquad \Gamma \vdash \mathsf{nd}\ A}{\Gamma \vdash \mathsf{nd}\ B}\ \supset E$$

$$\frac{\Gamma \vdash \mathsf{nd}\ [a/x]A}{\Gamma \vdash \mathsf{nd}\ (\forall x.A)}\ \forall I^a$$

$$\frac{\Gamma \vdash (\forall x.A)}{\Gamma \vdash \mathsf{nd}\ [T/x]A]}\ \forall E$$

$$\frac{(u : \mathsf{nd}\ A) \in \Gamma}{\Gamma \vdash \mathsf{nd}\ A}\ \text{Hyp}$$

```
nat : type.

Zero: nat.
Suc : nat → nat.

o   : type.
eq  : nat → nat → o.
imp : o → o → o.
forall: (nat → o) → o.

nd: o → type.

impi:(nd A → nd B)
   → nd (A imp B).

impe: nd (A imp B) → nd A
   → nd B.

alli: (Π a:nat. nd (A a))
   → nd (forall λx. A x).

alle: nd (forall λx. A x)
   → nd (A T).
```

Fig. 1. Natural deduction and its HOAS encoding

Analyzing data by cases is a basic programming and proof technique. In the first-order setting, it is straightforward. We simply consider all declared constants of a given type. To illustrate, in Figure 1 we define a simple logic with equality on numbers following typical encodings in the logical framework LF [4]. The cases arising when splitting on the proposition A are clear: they are exactly the three possible proposition forms specified in the grammar. However, already when we consider analyzing numbers we potentially will not only need to consider cases for 0 and $\mathsf{suc}\ N$, but also the case where we encounter a variable. A similar situation comes up with higher-order data, such as derivations in natural deduction. An encoding using higher-order abstract syntax does not represent the last rule Hyp explicitly. Instead, this base case will be implicit. Therefore, generating all cases exhaustively is not a simple matter. We must think of an object within a context, and we need to reason about the possible elements of the context.

Our main contribution is a theoretical framework for generating an exhaustive and complete set of cases for objects that may refer to assumptions, i.e. open objects. While some systems like Twelf provide implementations that support case analysis on open objects, previous theoretical work on coverage dealt with closed objects [14] and a clean theoretical analysis has so far been lacking. For many Twelf users, this operation remains mysterious, and we hope our rational reconstruction is a step towards demystifying this operation. Moreover,

we believe our work lays a solid foundation for languages such as Beluga [11] that support case analysis of open data, and is an important step towards understanding how to reason about HOAS encodings. In this paper, we describe a theoretical framework for checking whether a set of patterns is exhaustive, that is, whether it covers, and we prove soundness of this coverage checking framework.

We will begin with an example in the language Beluga, which supports programming with LF encodings in a functional setting. To emphasize the issues due to open terms, we will concentrate on the simply typed setting in this example. However, our formal framework treats dependently typed terms, which makes the problem harder. The structure of types can be observed, and this makes coverage checking undecidable, since any set of patterns will cover all terms of an empty type and emptiness is undecidable.

2 Motivation

To motivate the problem, we consider a simple program in the Beluga language [11] that counts the free occurrences of some variable x in a formula. For example, $\forall y.(x = y) \supset (\mathsf{suc}\ y = \mathsf{suc}\ x)$ has two free occurrences of x. The data language here is first-order logic with quantification over natural numbers, as defined in Figure 1, and we analyzes HOAS data via pattern matching. Using this example, we then discuss in more detail the problem of coverage.

We will write two functions to solve this problem. The function `cntV` will recursively analyze formulas. When it reaches a natural number expression, it will call a second function `cntVN`. We use modal types such as `o[x:nat, y:nat]`, which describes a formula that can refer to the variables `x` and `y`. The formula `((eq x y) imp (eq (Suc x) (Suc y))` has this type.

When `cntV` recursively reaches a formula with a universal quantifier, the set of free variables grows. Hence, we need to abstract over the contexts in which the formula makes sense. Context variables ψ provide this ability.

The function `cntV` (Fig. 2) takes in a context ψ of natural numbers, a formula `f`, and returns an integer. Just as types classify data objects and kinds classify types, we introduce *schemas* to classify contexts. In the type declaration for the function `cntV` we say that the context variable ψ has the schema `(nat)`*, meaning that ψ stands for a data-level context whose the form is $\mathtt{x}_1\mathtt{:nat},\ldots,\mathtt{x}_n\mathtt{:nat}$. We represent contextual variables which are instantiated via higher-order pattern matching with capital letters.

We examine the second function, `cntV`, first. It is built by a context abstraction $\Lambda\,\psi$ that introduces the context variable ψ and binds every occurrence of ψ in the body. Next, we introduce the computation-level variable `f` which has type `o[`ψ`,x:nat]`. In the body of the function `cntV` we case-analyze objects of type `o[`ψ`,x:nat]`. The **box** construct separates data from computations. Since formulas are constructed by equality `eq`, implication `imp` and quantification `forall`, we have cases for each of these.

```
rec cntVN : Π ψ:(nat)*.nat[ψ,x:nat] → int =
Λ ψ ⇒ fn n ⇒ case n of
  box(ψ,x. x]) ⇒ 1
| box(ψ,x. p[id_ψ]) ⇒ 0
| box(ψ,x. Zero) ⇒ 0
| box(ψ,x. Suc U[id_ψ,x]) ⇒ cntVN ⌈ψ⌉ box(ψ,x. U[id_ψ,x])

rec cntV : Π ψ:(nat)*.o[ψ,x:nat] → int =
Λ ψ ⇒ fn f ⇒ case f of
  box(ψ,x. eq U[id_ψ,x] V[id_ψ, x]) ⇒ cntVN ⌈ψ⌉ box(ψ,x.U[id_ψ,x])
                                    + cntVN ⌈ψ⌉ box(ψ,x.V[id_ψ,x])
| box(ψ,x. imp U[id_ψ,x] W[id_ψ,x]) ⇒ cntV ⌈ψ⌉ box(ψ,x. U[id_ψ,x])
                                    + cntV ⌈ψ⌉ box(ψ,x. W[id_ψ,x])
| box(ψ,x. forall(λy.U[id_ψ,x,y]))⇒ cntV⌈ψ,y:nat⌉ box(ψ,y,x. U[id_ψ,x,y])
```

Fig. 2. Counting free variables using pattern matching and HOAS

When we encounter an object built from a constructor `eq`, `imp`, or `forall`, we must extract the subexpression(s) underneath. Pattern variables are characterized by a closure `U[`σ`]` consisting of a contextual variable `U` and a *postponed substitution* σ. As soon as we know what the contextual variable stands for, we apply the substitution σ. In the example, the postponed substitution associated with `U` is the identity substitution which essentially corresponds to α-renaming. We write $\mathbf{id}_\psi$ for the identity substitution with domain ψ. Intuitively, one may think of the substitution associated with contextual variables which occur in patterns as a list of variables which may occur in the hole. In the data object `U[`$\mathbf{id}_\psi$`]`, for example, the contextual variable `U` can be instantiated with any formula which either is closed (does not refer to any bound variable listed in the context ψ) or contains a bound variable from the context ψ. Since we want to ensure that subformulas refer to all variables in ψ`, x:nat`, we write `U[`$\mathbf{id}_\psi$`, x]`. We use capital letters for meta-variables.

In the first case, for `eq`, we call `cntVN` to count the occurrences of `x` in the natural numbers `U[`$\mathbf{id}_\psi$`,x]` and `V[`$\mathbf{id}_\psi$`, x]`, explicitly passing the context ψ.

The second case for `imp` is similarly structured, calling `cntV` instead of `cntVN`.

In the third case, for `box(`ψ`,x. forall (`λ`y.W[`$\mathbf{id}_\psi$`,x,y]))`, we analyze the quantified formula under the assumption that `y` is a natural number. To do this, we pass an extended context `(`ψ`,y:nat)` to `cntV`.

The function `cntVN` counts the occurrences of a variable `x` in an object of type `nat[`ψ`,x:nat]`, considering four cases. The first case, `box(`ψ`,x. x)`, matches an occurrence of `x`. The second case, `box(`ψ`,x. p[`$\lambda\mathbf{id}_\psi$`])`, matches a variable that is not `x` and occurs in ψ. For this case, we use a *parameter variable* `p` (using a small letter to distinguish it from a meta-variable). This represents a bound object-level variable. The substitution $\mathbf{id}_\psi$ associated with `p` characterizes the possible instantiations of `p`. The remaining cases are straightforward.

2.1 Basic idea of coverage on open data

In this paper, we provide the foundation for ensuring that case expressions which analyze elements of type $A[\Psi]$ via pattern matching cover all possible elements of this type. For example, in the function `cntV` we ensure that the set of patterns {`x, p[id_`ψ`]`, `Zero`, `Suc U[id_`ψ`, x]`} covers the type `nat[`ψ`,x:nat]`. In `cntV`, the set {`eq U[id`$_\psi$`,x] V[id`$_\psi$`,x]`, `imp U[id`$_\psi$`,x] W[id`$_\psi$`,x]`, `forall (`λ`y.U[ id`$_\psi$`,x,y])`} covers all elements of type `o[`ψ`,x:nat]`.

This set of patterns for covering the type `o[`ψ`,x:nat]` is by no means the only one. Instead of explicitly counting the occurrences of `x` in a natural number of type `nat[`ψ`,x:nat]`, we could have used the power of higher-order pattern matching to enforce variable dependencies. In other words, we could have refined the pattern `eq U[id`$_\psi$`,x] V[id`$_\psi$`,x]` into the following four cases: {`eq U[id`$_\psi$`] V[id`$_\psi$`]`, `eq U[id`$_\psi$`,x] V[id`$_\psi$`]`, `eq U[id`$_\psi$`] V[id`$_\psi$`,x]`, `eq U[id`$_\psi$`,x] V[id`$_\psi$`,x]`}, exactly distinguishing between (1) no variable `x` occurs in `U[id`$_\psi$`]` and `V[id`$_\psi$`]`, (2) the variable `x` occurs in `U[id`$_\psi$`,x]` but not in `V[id`$_\psi$`]`, (3) the variable `x` occurs in `V[id`$_\psi$`,x]` but not in `U[id`$_\psi$`]`, and (4) the variable `x` occurs in both `U[id`$_\psi$`,x]` and `V[id`$_\psi$`,x]`.

More generally, we provide a formal framework for answering the following question: Does a set of patterns cover the type $A[\Psi]$? Alternatively, our framework provides a general way of generating a set of patterns thereby providing a foundation for splitting an object of type $A[\Psi]$ into different cases.

We would like to emphasize that while we illustrate the problem here in the setting of Beluga, in which contexts are explicit, similar situations arise when contexts are implicit, as in other systems such as Delphin and Twelf.

3 Background

We will concentrate here on the data level, since we are mainly interested in testing whether a set of patterns covers a given data object. For a discussion of the computation level, see [11].

We essentially support the full logical framework LF plus Σ-types. Our data layer closely follows contextual modal type theory [7], extended with parameter variables, substitution variables, and context variables [11], and finally with dependent pairs and projections. Perhaps most importantly, we formalize schemas, which classify contexts. We only characterize normal terms since only these are meaningful in the logical framework, following Watkins et al.[17] and Nanevski et al.[7]. This is achieved by distinguishing between normal terms M and neutral terms R. While the syntax only guarantees that terms N contain no β-redexes, the typing rules will also guarantee that all well-typed terms are fully η-expanded.

We distinguish between four kinds of variables in our theory: *Ordinary bound variables* are used to represent data-level binders and are bound by λ-abstraction. *Contextual variables* stand for open objects, and include *meta-variables* u, which

Kinds	K	::=	$\mathsf{type} \mid \Pi x{:}A.K$
Atomic types	P	::=	$a\ M_1 \ldots M_n$
Types	A, B	::=	$P \mid \Pi x{:}A.B \mid \Sigma x{:}A.B$
Normal terms	M, N	::=	$\lambda x.\, M \mid (M, N) \mid R$
Neutral terms	R	::=	$c \mid x \mid u[\sigma] \mid p[\sigma] \mid R\ N \mid \mathsf{proj}_k R$
Substitutions	σ, ρ	::=	$\cdot \mid \sigma\,;\, M \mid \sigma\,,\, R \mid s[\sigma] \mid \mathsf{id}_\psi$
Context variables	ψ, ϕ		
Contexts	Ψ, Φ	::=	$\cdot \mid \psi \mid \Psi, x{:}A$
Meta-contexts	Δ	::=	$\cdot \mid \Delta, u{::}A[\Psi] \mid \Delta, p{::}A[\Psi] \mid \Delta, s{::}\Psi[\Phi]$
Schema contexts	Ω	::=	$\cdot \mid \Omega, \psi{::}W$

Fig. 3. The data level

represent general open objects, *parameter variables* p that can only be instantiated with an ordinary bound variable, and *substitution variables* s, which represent a mapping from one context to another. Contextual variables are introduced in computation-level case expressions, and can be instantiated via pattern matching. They are associated with a postponed substitution σ thereby representing a closure. Our intention is to apply σ as soon as we know which term the contextual variable should stand for. The domain of σ thus describes the free variables that can possibly occur in the object which represents the contextual variable, and the type system statically guarantees this.

Substitutions σ are built of either normal terms (in $\sigma\,;\, M$) or atomic terms (in $\sigma\,,\, R$). We do not make the domain of the substitutions explicit, which simplifies the theoretical development and avoids having to rename the domain of a given substitution σ. Similar to meta-variables, substitution variables are closures with a postponed substitution. We also have a first-class notion of identity substitution id_ψ. Our convention is that data-level substitutions, as defined operations on data-level terms, are written $[\sigma]N$.

Contextual variables such as meta-variables u, parameter variables p, and substitution variables s are declared in a meta-level context Δ, while ordinary bound variables are declared in a context Ψ.

Finally, our foundation supports *context variables* ψ which allow us to reason abstractly with contexts. Abstracting over contexts is an interesting and essential next step to allow recursion over higher-order abstract syntax. Context variables are declared in Ω. Unlike previous uses of context variables [6], a context may contain at most one context variable. In the same way that types classify objects, and kinds classify types, we introduce the notion of a schema W that classifies contexts Ψ. We will return to the definition of schemas on page 8.

We assume that type constants and object constants are declared in a signature Σ as pure LF objects, i.e. data objects of Π-type. We suppress the signature since it is the same throughout a typing derivation, but we will keep in mind that all typing judgments have access to a well-formed signature. As a notational convenience, we write $\mathsf{proj}_k^{\#}\ R$ for the kth projection of R.

Data-level typing We present a bidirectional type system for data-level terms. Typing is defined via the following judgments:

$\Omega; \Delta; \Psi \vdash M \Leftarrow A$	Check normal object M against A
$\Omega; \Delta; \Psi \vdash R \Rightarrow A$	Synthesize A for neutral object R
$\Omega; \Delta; \Phi \vdash \sigma \Leftarrow \Psi$	Check σ against context Ψ

For readability, we omit Ω in the subsequent development since it is constant; we also assume that Δ and Ψ are well-formed. First, the typing rules for objects.

Data-level normal terms

$$\frac{\Delta; \Psi, x{:}A \vdash M \Leftarrow B}{\Delta; \Psi \vdash \lambda x.\, M \Leftarrow \Pi x{:}A.B}\,\underline{\Pi\mathrm{I}} \qquad \frac{\Delta; \Psi \vdash M_1 \Leftarrow A_1 \quad \Delta; \Psi \vdash M_2 \Leftarrow [M_1/x]^a_{A_1} A_2}{\Delta; \Psi \vdash (M_1, M_2) \Leftarrow \Sigma x{:}A_1.A_2}\,\underline{\Sigma\mathrm{I}}$$

$$\frac{\Delta; \Psi \vdash R \Rightarrow P' \qquad P' = P}{\Delta; \Psi \vdash R \Leftarrow P}\ \underline{\mathsf{turn}}$$

Data-level neutral terms

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

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

$$\frac{\Delta; \Psi \vdash R \Rightarrow \Sigma x{:}A_1.A_2}{\Delta; \Psi \vdash \mathsf{proj}_1 R \Rightarrow A_1}\ \underline{\Sigma\mathrm{E}_1} \qquad \frac{\Delta; \Psi \vdash R \Rightarrow \Sigma x{:}A_1.A_2}{\Delta; \Psi \vdash \mathsf{proj}_2 R \Rightarrow [\mathsf{proj}_1 R/x]^a_{A_1} A_2}\ \underline{\Sigma\mathrm{E}_2}$$

Data-level substitutions

$$\frac{}{\Delta; \Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{}{\Delta; \psi, \Psi \vdash \mathsf{id}_\psi \Leftarrow \psi} \qquad \frac{s{::}\Phi_1[\Phi_2] \in \Delta \qquad \Delta; \Psi \vdash \rho \Leftarrow \Phi_2 \qquad \Phi \stackrel{\alpha}{=} \Phi_1}{\Delta; \Psi \vdash (s[\rho]) \Leftarrow \Phi}$$

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

We assume that data level type constants a together with constants c have been declared in a signature. 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 modal variables u. 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 Ψ. This can always be achieved via α-renaming. The typing rules for data-level neutral terms rely on *hereditary substitutions* which ensure that canonical forms are preserved [7].

The idea is to define a primitive recursive functional that always returns a canonical object. In places where the ordinary substitution would construct a redex $(\lambda y.\, M)\, N$ we must continue, substituting N for y in M. Since this could again create a redex, we must continue and hereditarily substitute and eliminate potential redexes. Hereditary substitution can be defined recursively,

$$
\begin{array}{lrcl}
\text{Element types} & A^* & ::= & \Pi x{:}A.A^* \mid a\ N_1 \ldots N_n \\
\text{Schema elements} & F & ::= & \mathsf{all}\, x_1{:}B_1^*, \ldots x_k{:}B_k^*.\Sigma y_1{:}A_1^*, \ldots, y_j{:}A_j^*.A^* \\
\text{Schemas} & W & ::= & (F_1 + \cdots + F_n)^*
\end{array}
$$

Context Ψ checks against a schema W

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

$$\frac{\psi{::}W \in \Omega}{\Omega;\Delta \vdash \psi \Leftarrow W} \qquad \frac{}{\Omega;\Delta \vdash \cdot \Leftarrow W}$$

Type A is an instance of schema element $F = \mathsf{all}\,\Theta.\Sigma\Phi^*.\,B^*$

$$\frac{\Theta^* = x_1{:}C_1^*, \ldots, x_n{:}C_n^* \qquad \sigma = u_1[\mathsf{id}(\Psi)]/x_1, \ldots, u_n[\mathsf{id}(\Psi)]/x_n \qquad \Omega;\Delta;u_1{::}C_1^*[\Psi], \ldots, u_n{::}C_n^*[\Psi];\Psi \vdash A \doteq [\sigma]\Sigma\Phi^*.B^*\ /\ (\theta,\Delta)}{\Omega;\Delta;\Psi \vdash A \in \mathsf{all}\,\Theta^*.\Sigma\Phi^*.\,B^*}$$

Fig. 4. Schemas

considering both the structure of the term to which the substitution is applied and the type of the object being substituted. We omit the definition of ordinary hereditary substitutions and refer the reader to Nanevski et al. [7] for details. For the subsequent development we omit the subscripts for better readability.

Context schemas As the earlier example illustrated, contexts play an important part in programming with open data objects, and in particular contexts which are explicitly constructed passed will belong to a specific context schema. In the earlier example, the context schema $(\mathsf{nat})^*$ represented contexts of the form $\mathtt{x}_1\mathtt{:nat, \ldots, x}_n\mathtt{:nat}$. In general, we allow much more expressive contexts. For instance, when reasoning about natural deductions, the implication introduction rule adds an assumption of the form `u:nd A` for some concrete proposition `A`.

Inductive definition for contexts	Representation of the context
$\Gamma' ::= \cdot \mid \Gamma'$, `x:nat` $\mid \Gamma'$, `u:nd` A	$\mathsf{nat} + (\mathsf{all}\, A : \mathtt{o.nd}\ A)$

We use + to denote a choice of possible elements in a context, and all allows us to describe an assumption for all possible propositions A. One concrete instance of this schema is `x:nat, u:nd (eq x x)`, which arises when describing the derivation of `forall λx. (eq x x) imp (eq (Suc x) (Suc x))`.

We give the grammar of schemas in Figure 4. Schemas are built of elements $F_1, \ldots, F_n$, each of which must have the form $\mathsf{all}\,\Phi^*.\Sigma y_1{:}A_1^*, \ldots, y_j{:}A_j^*.\,A^*$ where $\Phi^* = x_1{:}B_1^*, \ldots x_k{:}B_k^*$. In other words, the element is of $\Sigma\Pi$-type, where we first introduce some Σ-types, followed by pure Π-types. We disallow arbitrary mixing of Σ and Π. This restriction will make it easier to describe the possible terms of this type, which is a crucial step towards ensuring coverage. It also seems sufficient for many practical situations, since the Twelf system employs

a similar restriction on worlds. In Beluga, the computation typing rules [11] guarantee that contexts matching this grammar are the only contexts created during computation.

To check a context Ψ against a schema $(F_1 + \cdots + F_n)$, we check that Ψ is an instance of a schema element $F_k = \mathsf{all}\, \Phi^*.\Sigma y_1{:}A_1^*, \ldots, y_j{:}A_j^*.\, A^*$, with all variables in Φ^* instantiated such that Ψ is an instance of F_k. In Figure 4 we show how to check that a given context Ψ is an instance of a schema element F_i using higher-order pattern matching.

4 Coverage Checking

In this section, we present a theory for coverage checking. The task is to decide whether every closed term of type $A[\Psi]$ is an instance of at least one of a given set of patterns; in Beluga, this is the set of patterns guarding the branches of a case expression. This is undecidable in general, since any set of patterns covers all terms of an empty type, and emptiness is undecidable [5, p. 179]. In the setting of Beluga, empty types are useless, so there is no reason to worry about how coverage checking behaves with empty types.

The work of Coquand [2] and of Schürmann and Pfenning [14] described coverage checking for closed terms. A theoretical treatment of coverage over open data objects has been left open.

Coverage checking always requires some notion of *splitting*. In the setting of closed terms, to see that the set of patterns $Z = \{\mathtt{Zero}, \mathtt{Suc}\ u\}$ covers the type `nat`, one splits `nat` into its constituent constructors. In our setting, the question becomes: does Z cover $\mathtt{nat}[\Psi]$ for some context Ψ?

In fact, Z only covers $\mathtt{nat}[\cdot]$, the *closed* terms of type `nat`. With open data, the type is also inhabited by variables. Thus, $\mathtt{nat}[\Psi]$ should be split into the constituent constructors *plus* the possible variables. For example, to see that $Z' = \{\mathtt{Zero}, \mathtt{Suc}\ u[\mathsf{id}_\psi], x, p_2[\mathsf{id}_\psi]\}$ covers $\mathtt{nat}[\psi, x{:}\mathtt{nat}, y{:}\mathtt{o}]$, we split that type into the constructors along with the parameter variables $p[\mathsf{id}_\psi]$ denoting the generic case that we may encounter a variable from ψ and concrete variables `x` and `y` which occur in the context:

$$\overbrace{\mathtt{Zero},\ \mathtt{Suc}\ u[\mathsf{id}_\psi, x, y]}^{\text{constructors of } \mathtt{nat}},\ \overbrace{p_1[\mathsf{id}_\psi],\ p_2[\mathsf{id}_\psi],\ x,\ y}^{\text{variables of } \psi, x{:}\mathtt{nat}, y{:}\mathtt{o}}$$

Next, we throw out variables that are not of the covering type `nat`. To do this, we need to know the types of the parameter variables p_1 and p_2. These come from the schema of context ψ. Suppose that ψ represents a context of the form `(o)*+ (nat)*`. We can let $p_1[\mathsf{id}_\psi]$ stand for any variable in ψ of type `o`, and $p_2[\mathsf{id}_\psi]$ stand for any variable in ψ of type `nat`. Then we can discard $p_1[\mathsf{id}_\psi]$ and y—both of which are variables of type `o`, not of type `nat`—giving a direct correspondence to Z', from which coverage is obvious.

In practice, we can rely on techniques based on subordination [16] to eliminate the variable y in $\mathtt{Suc}\ u[\mathsf{id}_\psi, x, y]$, since natural numbers do not include subterms of type `o`.

Note that we could also refine $u[\mathsf{id}_\psi, x, y]$ further by splitting it into *its* constituent constructors and variables to obtain `Suc Zero` and `Suc` (`Suc` $u[\mathsf{id}_\psi, x, y]$), but there was no need. On the other hand, showing that Z' covers required the "outermost" split. Decisions about when and where to split are not determined by our theory; such choices are embodied in a nondeterministic choice between two rules. Our system is thus the *foundation* for a coverage checking algorithm.

We state some key metatheoretical results, and then describe how the coverage checking rules work. We write $[\![\theta]\!]$ for meta-level substitution.

Theorem 1. *All data-level typing judgments are decidable.*

Proof. The typing rules are syntax-directed, and therefore clearly decidable assuming hereditary substitution is decidable.

Theorem 2 (Soundness of higher-order pattern unification).
If $\Omega; \Delta; \Psi \vdash Q \Leftarrow \mathsf{type}$ *and* $\Omega; \Delta; \Psi \vdash P \Leftarrow \mathsf{type}$ *and* $\Omega; \Delta; \Psi \vdash Q \doteq P \;/\; (\theta, \Delta')$
then $\Omega; \Delta' \vdash \theta : \Delta$ *and* $\Omega; \Delta'; [\![\theta]\!]\Psi \vdash [\![\theta]\!]P = [\![\theta]\!]Q$
and θ *is the most general unifier, that is,*
for all $\cdot; \cdot \vdash \rho : (\Omega; \Delta)$ *there exists* θ' *such that* $\rho = [\![\theta']\!]\theta$.

Proof. The proof is a simple extension of the one in [10].

Lemma 1 (Object inversion).
If $\cdot; \cdot; \Psi \vdash R \Leftarrow P$ *and* $\vdash \Psi : W$ *then*

(1) $R = c\ N_1\ \ldots\ N_k$ *iff* $\Sigma(c) = \Pi x_1{:}A_1. \cdots \Pi x_k{:}A_k.P'$
and $[N_1/x_1, \ldots, N_k/x_k]P' = P$.

(2) $R = x\ N_1\ \ldots\ N_k$ *iff* $\Psi(x) = \Pi x_1{:}A_1. \cdots \Pi x_k{:}A_k.P'$
and $[N_1/x_1, \ldots, N_k/x_k]P' = P$.

(3) $R = (\mathsf{proj}_l^{\#}\ y)\ N_1\ \ldots\ N_k$ *iff* $\Psi(y) = \Sigma y_1{:}A_1^*, \ldots y_m{:}A_m^*.A_{m+1}^*$
and $[\mathsf{proj}_1^{\#}\ y/y_1, \ldots, \mathsf{proj}_l^{\#}\ y/y_l]A_{l+1}^* = \Pi x_1{:}B_1. \ldots \Pi x_k{:}B_k.P'$
where $1 \le l \le m$ *and* $[N_1/x_1, \ldots N_k/x_k]P' = P$.

Proof. By case analysis and inversion on the derivation of $\cdot; \cdot; \Psi \vdash R \Leftarrow P$.

4.1 Overview of coverage judgments

The most essential judgment has the form $\Omega; \Delta; \Psi \vdash \mathsf{Obj}(A) \rhd \textsc{covered-by}\ Z$, meaning that every object of type A is matched by at least one pattern in Z. A derivation of this judgment has subderivations of a more general form, $\Omega; \Delta; \Psi \vdash \mathsf{Obj}(A) \rhd \mathcal{J}$, which analyzes A and "continues" with $\mathcal{J}$. Thus, the first judgment $\Omega; \Delta; \Psi \vdash \mathsf{Obj}(A) \rhd \textsc{covered-by}\ Z$ says to analyze A and then check whether Z covers it.

Within the derivation, the splits produced are represented by subderivations of $\Omega; \Delta; \Psi \vdash M : A \rhd \mathcal{J}$, saying that M—which can include "pattern variables"

$u[\sigma]$, very roughly corresponding to wildcard patterns—has type A. The shape of a coverage derivation (omitting contexts for clarity) is:

$$\frac{\begin{array}{ccc} M_1 : A_1 \ \rhd\ \mathcal{J} & \ldots & M_n : A_n \ \rhd\ \mathcal{J} \\ \vdots & & \vdots \end{array}}{\mathsf{Obj}(A) \ \rhd\ \mathcal{J}}$$

where $M_1, \ldots, M_n$ collectively cover all possible terms of type A, and $A_1, \ldots, A_n$ may be more precise than A.

In a derivation of $\mathsf{Obj}(A) \rhd \text{COVERED-BY}\ Z$, we examine Z only at the leaves. Following the diagram above, with $\mathcal{J} = \text{COVERED-BY}\ Z$, each subderivation rooted at $M_k : A_k \ \rhd\ \text{COVERED-BY}\ Z$ must look like this:

$$\dfrac{\dfrac{\Omega \vdash (\Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M_k) : A_k[\Psi]) \ \doteq\ \zeta \ \ / \ (\theta, \Delta'')}{\Omega \vdash \Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M_k) : A_k[\Psi] \ \text{COVERED-BY}\ \zeta}\ \textsf{Covered-By-}\zeta}{\Omega; \Delta; \Psi \vdash M_k : A_k \ \rhd\ \text{COVERED-BY}\ \{\ldots, \zeta, \ldots\}}\ \textsf{Covered-By-}Z$$

We assume that the pattern ζ includes an explicit meta-variable context Δ', explicit object-level names $\hat{\Psi}'$, and an explicit type $A'[\Psi']$. The expressions on each side of the $\doteq$ thus have the same shape, so the $\doteq$ premise is

$$\Omega \vdash \underbrace{(\Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M_k) : A_k[\Psi])}_{\text{result of coverage analysis}} \doteq \underbrace{(\Pi\Delta'.\mathsf{box}(\hat{\Psi}'.\, M') : A'[\Psi'])}_{\text{guard from case expression}} \ / \ (\theta, \Delta'')$$

Ignoring contexts for a moment, this says that M_k is an instance of M', realized by the substitution θ, that is, $M_k = [\![\theta]\!]M'$. If all the M_k resulting from analysis of A are thus covered, then all terms of type A are covered.

4.2 Obj(P): analyzing base types

When deriving $\mathsf{Obj}(P) \ \rhd\ \mathcal{J}$, we can choose not to split (rule Obj-no-split), in which case the M_k shown above will have the form $u[\mathsf{id}(\Psi_k)]$ for various Ψ_k (discussed below). Alternatively, we can apply the rule Obj-split (bottom of Figure 5) to split it into several terms. Each of these will look like $R\ N_1 \ldots N_m$, where R is a variable x, constructor c or parameter $p[\sigma]$—or some projection of x or $p[\sigma]$.

The simplest of these are the constructors c. The rule Obj-split has a premise $\mathsf{App}\langle c_k\rangle(\Sigma(c_k) > P) \ \rhd\ \mathcal{J}$ for every object-level constructor c_k. This even includes constructors for completely different base types, which will be discarded further up the derivation. In general, we must generate many spines $N_1 \ldots N_m$ in derivations of $\Omega; \Delta; \Psi \vdash \mathsf{App}\langle R\rangle(A^* > P) \ \rhd\ \mathcal{J}$. The P denotes that we are constructing objects of type P. The rules for such judgments are defined on the structure of $A^* = \Pi x_1{:}A_1 \cdots \Pi x_m{:}A_m.Q$. Eventually, after going through this for each argument, we will have a fully-applied $c_k\ N_1 \ldots N_m$ where the A^* is some atomic type Q. To ensure that the type Q of the constructed object is equal to our target type P, we unify them (in rule App-$\doteq$). If they do not unify

$\boxed{\Omega \vdash \Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M) : A[\Psi] \ \textsc{covered-by}\, \zeta}$

$$\frac{\Omega \vdash (\Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M) : A[\Psi]) \doteq (\Pi\Delta'.\mathsf{box}(\hat{\Psi}'.\, M') : A'[\Psi']) \ /\ (\theta, \Delta'')}{\Omega \vdash \Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M) : A[\Psi] \ \textsc{covered-by}\, (\Pi\Delta'.\mathsf{box}(\hat{\Psi}.\, M') : A'[\Psi'])}\ \mathsf{Covered\text{-}By\text{-}}\zeta$$

$\boxed{\Omega; \Delta; \Psi \vdash \mathsf{App}\langle R\rangle(A > P) \ \rhd\ \mathcal{J}}$

$$\frac{\Omega; \Delta; \Psi \vdash Q \Rightarrow\Leftarrow P}{\Omega; \Delta; \Psi \vdash \mathsf{App}\langle R\rangle(Q > P) \ \rhd\ \mathcal{J}}\ \mathsf{App\text{-}}\Rightarrow\Leftarrow \qquad \frac{\Omega; \Delta; \Psi \vdash Q \doteq P \ /\ (\theta, \Delta') \quad \Omega; \Delta'; [\![\theta]\!]\Psi \vdash [\![\theta]\!]R : [\![\theta]\!]P \ \rhd\ [\![\theta]\!]\mathcal{J}}{\Omega; \Delta; \Psi \vdash \mathsf{App}\langle R\rangle(Q > P) \ \rhd\ \mathcal{J}}\ \mathsf{App\text{-}}\doteq$$

$$\frac{\Omega; \Delta; \Psi \vdash \mathsf{App}\langle R\ M\rangle([M/x]B > P) \ \rhd\ \mathcal{J}}{\Omega; \Delta; \Psi \vdash M : A \ \rhd\ \textsc{neutral}\langle R\rangle(x.B > P) \ \rhd\ \mathcal{J}}$$

$$\frac{\Omega; \Delta; \Psi \vdash \mathsf{Obj}(A) \ \rhd\ \textsc{neutral}\langle R\rangle(x.B > P) \ \rhd\ \mathcal{J}}{\Omega; \Delta; \Psi \vdash \mathsf{App}\langle R\rangle(\Pi x{:}A.B > P) \ \rhd\ \mathcal{J}}\ \mathsf{App\text{-}}\Pi$$

for $0 \le i \le m$:

$$\frac{\Omega; \Delta; \Psi \vdash \mathsf{App}\langle \mathsf{proj}_i^{\#}\, R\rangle([\mathsf{proj}_1^{\#} R/x_1, \ldots, \mathsf{proj}_i^{\#} R/x_i]A^*_{i+1} > P) \ \rhd\ \mathcal{J}}{\Omega; \Delta; \Psi \vdash \mathsf{App}\langle R\rangle(\Sigma x_1{:}A^*_1, \ldots, x_m{:}A^*_m.A^*_{m+1} > P) \ \rhd\ \mathcal{J}}\ \mathsf{App\text{-}}\Sigma$$

$\boxed{\Omega; \Delta; \Psi \vdash M : A \ \rhd\ \mathcal{J}}$

$$\frac{\Omega \vdash \Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M) : A[\Psi] \ \textsc{covered-by}\, \zeta_k}{\Omega; \Delta; \Psi \vdash M : A \ \rhd\ \textsc{covered-by}\, \{\zeta_1, \ldots, \zeta_n\}}\ \mathsf{Covered\text{-}By\text{-}}Z$$

$$\frac{\Omega; \Delta; \Psi \vdash (\lambda x.\, M) : (\Pi x{:}A_1.A_2) \rhd \mathcal{J}}{\Omega; \Delta; \Psi, x{:}A_1 \vdash M : A_2 \rhd \textsc{lam} \rhd \mathcal{J}} \qquad \frac{\Omega; \Delta; \Psi \vdash (M, N) : \Sigma x{:}A_1.A_2 \ \rhd\ \mathcal{J}}{\Omega; \Delta; \Psi \vdash N : [M/x]A_2 \rhd \textsc{pair2}\,(M{:}A_1, x.\bullet) \rhd \mathcal{J}}$$

$$\frac{\Omega; \Delta; \Psi \vdash \mathsf{Obj}([M/x]A_2) \ \rhd\ \textsc{pair2}\,(M{:}A_1, x.\bullet) \ \rhd\ \mathcal{J}}{\Omega; \Delta; \Psi \vdash M : A_1 \ \rhd\ \textsc{pair1}\,(\bullet,\ x.A_2) \ \rhd\ \mathcal{J}}$$

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

$$\frac{\Omega; \Delta; \Psi, x{:}A_1 \vdash \mathsf{Obj}(A_2) \ \rhd\ \textsc{lam} \ \rhd\ \mathcal{J}}{\Omega; \Delta; \Psi \vdash \mathsf{Obj}(\Pi x{:}A_1.A_2) \ \rhd\ \mathcal{J}} \qquad \frac{\Omega; \Delta; \Psi \vdash \mathsf{Obj}(A_1) \ \rhd\ \textsc{pair1}\,(\bullet,\ x.A_2) \ \rhd\ \mathcal{J}}{\Omega; \Delta; \Psi \vdash \mathsf{Obj}(\Sigma x{:}A_1.A_2) \ \rhd\ \mathcal{J}}$$

$$\frac{\Omega; \Delta; \Psi \vdash \mathsf{MVars}(P) \ \rhd\ \mathcal{J}}{\Omega; \Delta; \Psi \vdash \mathsf{Obj}(P) \ \rhd\ \mathcal{J}}\ \mathsf{Obj\text{-}no\text{-}split}$$

$$\frac{\begin{array}{ll} & \Omega; \Delta; \Psi \vdash \mathsf{App}\langle x_1\rangle(\Sigma\Psi^*_1.A^*_1 > P) \rhd \mathcal{J} \\ \Psi = \psi, x_1{:}\Sigma\Psi^*_1.A^*_1, \ldots, x_k{:}\Sigma\Psi^*_k.A^*_k & \vdots \\ \Omega(\psi) = F_1 + \cdots + F_m & \Omega; \Delta; \Psi \vdash \mathsf{App}\langle x_k\rangle(\Sigma\Psi^*_k.A^*_k > P) \rhd \mathcal{J} \\ \Omega; \Delta; \Psi \vdash \mathsf{PVars}\langle\psi : F_1\rangle > P \ \rhd\ \mathcal{J} & \Omega; \Delta; \Psi \vdash \mathsf{App}\langle c_1\rangle(\quad \Sigma(c_1) > P) \rhd \mathcal{J} \\ \vdots & \vdots \\ \Omega; \Delta; \Psi \vdash \mathsf{PVars}\langle\psi : F_m\rangle > P \ \rhd\ \mathcal{J} & \Omega; \Delta; \Psi \vdash \mathsf{App}\langle c_n\rangle(\quad \Sigma(c_n) > P) \rhd \mathcal{J} \end{array}}{\Omega; \Delta; \Psi \vdash \mathsf{Obj}(P) \ \rhd\ \mathcal{J}}\ \mathsf{Obj\text{-}split}$$

Fig. 5. Coverage checking rules

$$\boxed{\Omega; \Delta; \Psi \vdash \mathsf{MVars}(P) \ \rhd\ \mathcal{J}}$$

$$\frac{\begin{array}{ll} & \Omega; \Delta, u{::}P[\Psi_1]; \Psi \vdash (u[\mathsf{id}(\Psi_1)] : P) \ \rhd\ \mathcal{J} \\ \mathsf{ValidWk}(\cdot; \cdot \vdash P[\Psi]) & \vdots \\ = \{\Psi_1, \ldots, \Psi_n\} & \Omega; \Delta, u{::}P[\Psi_n]; \Psi \vdash (u[\mathsf{id}(\Psi_n)] : P) \ \rhd\ \mathcal{J} \end{array}}{\Omega; \Delta; \Psi \vdash \mathsf{MVars}(P) \ \rhd\ \mathcal{J}}\ \mathsf{MVars}$$

$$\boxed{\Omega; \Delta; \Psi \vdash \mathsf{PVars}\langle \psi : \mathsf{all}\, \Theta^*.\Sigma\Phi^*.\, A^*_{j+1}\rangle > P \ \rhd\ \mathcal{J}}$$

$$\frac{\begin{array}{l} \Theta^* = y_1{:}B^*_1, \ldots, y_n{:}B^*_n \ \text{ and } \ \Phi^* = x_1{:}A^*_1, \ldots, x_j{:}A^*_j \\ \sigma = u_1[\mathsf{id}_\psi]/y_1, \ldots, u_n[\mathsf{id}_\psi]/y_n \\ \Delta_\Theta = u_1{::}B^*_1[\psi], \ldots, u_n{::}B^*_n[\psi] \\ \text{for } 0 \le i \le j\text{:} \\ \quad \sigma' = (\mathsf{proj}^{\#}_1\ p[\mathsf{id}_\psi])/x_1, \ldots, (\mathsf{proj}^{\#}_i\ p[\mathsf{id}_\psi])/x_i \\ \quad \Omega; \Delta, \Delta_\Theta, p{::}[\sigma]((\Sigma\Phi^*.A^*_{j+1})[\psi]); \Psi \vdash \mathsf{App}\langle \mathsf{proj}^{\#}_i\ p[\mathsf{id}_\psi]\rangle([\sigma'][\sigma]A^*_{i+1} > P) \ \rhd\ \mathcal{J} \end{array}}{\Omega; \Delta; \Psi \vdash \mathsf{PVars}\langle \psi : \mathsf{all}\, \Theta^*.\Sigma\Phi^*.A^*_{j+1}\rangle > P \ \rhd\ \mathcal{J}}\ \mathsf{PVars}$$

Fig. 6. Coverage checking rules (continued)

(written $Q \Rightarrow\!\Leftarrow P$ in rule App-$\Rightarrow\!\Leftarrow$) we have a trivial coverage derivation, and the term is discarded.

Next, we turn to the premises for variables in the rule Obj-split: $\mathsf{App}\langle x_k\rangle(B > P) \ \rhd\ \mathcal{J}$ where $x_k{:}B$ is in Ψ. If B is a sequence of Πs around a base type, the system behaves as it does for the constructors. However, since B can have one or more Σs around some B^*, we need to take all appropriate projections to get components of the dependent tuple of type Σ.

Finally, we discuss the remaining premises of Obj-split, which have the form $\mathsf{PVars}\langle \psi : F\rangle > P \ \rhd\ \mathcal{J}$. These characterize the generic variable case by constructing parameter variables p of the appropriate type, and hinge on our notion of context schemas. For each possible schema element, we generate a parameter variable. The type of the parameter variable is given by the schema element F. Recall the schema element $\mathsf{all}\, A{:}\mathtt{o.nd}\, A$ describing assumptions about natural deductions. To describe the type of a parameter p generically, we first create a meta-variable for each all-quantified variable in the element. So, in this example, $p[\mathsf{id}_\psi]$ has type $\mathtt{nd}\ u[\mathsf{id}_\psi]$ where u is a (fresh) meta-variable. In general, we get the type of a parameter from the element $\mathsf{all}\, \Theta^*.\Sigma\Phi^*.A^*$ by generating a substitution σ' that instantiates all variables in Θ^* with meta-variables, and applying σ' to $\Sigma\Phi^*.A^*$. Then we closely follow the ideas for concrete variables. Again, since $[\sigma']\Sigma\Phi^*.A^*$ is inhabited by tuples, we consider all possible projections.

4.3 $\mathsf{MVars}(P) \ \rhd\ \mathcal{J}$: General case for all ground instances of P

The rule Obj-split refined an element of type P. In contrast, the rule Obj-no-split creates a meta-variable $u[\mathsf{id}(\Psi)]$ of type $P[\Psi]$. Since any object of type $P[\Psi]$ matches $u[\mathsf{id}(\Psi)]$, it covers all possible ground instances of the type $P[\Psi]$.

We also need to refine pattern sets Z based on variable dependencies. Recall our earlier example where we refined the pattern `eq U[id`$_\psi$`,x] V[id`$_\psi$`,x]` into the four cases {`eq U[id`$_\psi$`] V[id`$_\psi$`]`, `eq U[id`$_\psi$`,x] V[id`$_\psi$`]`, `eq U[id`$_\psi$`] V[id`$_\psi$`,x]`, `eq U[id`$_\psi$`,x] V[id`$_\psi$`,x]`}. Thus, instead of simply generating $u[\mathsf{id}(\Psi)]$ we generate all sensible weakenings $\{\Psi_1, \ldots, \Psi_n\}$ of Ψ, and for each Ψ_i we generate $u[\mathsf{id}(\Psi_i)]$.

4.4 Coverage soundness

Roughly, the soundness result we need is that, if $\cdot;\cdot;\Psi \vdash \mathsf{Obj}(A) \rhd \text{COVERED-BY}\ Z$, then for every M of type A there is a pattern in Z that matches M. That theorem will not be difficult once we have a key lemma. This lemma will guarantee that if we have a derivation for $\mathcal{S} :: \ldots \vdash \mathsf{Obj}(A) \rhd \mathcal{J}$ then for every ground object M' of type A there is some subderivation $\mathcal{S}' :: \ldots \vdash M_i : A \rhd \mathcal{J}$ within $\mathcal{S}$ where M' is an instance of M_i. Once we have this lemma, the path to soundness is easy: if $\mathcal{J} = \text{COVERED-BY}\ Z$, the subderivation $\mathcal{S}'$ is of $\ldots \vdash M : A \rhd \text{COVERED-BY}\ Z$, and inversions bring us to the premise of $\mathsf{Covered\text{-}By}\text{-}Z$.

To state the lemma precisely, we first note that while the coverage judgment $\Omega;\Delta;\Psi \vdash \mathsf{Obj}(A) \rhd \mathcal{J}$ generically covers some Ω and Δ, when we run a program we only consider closed data, so these contexts will be empty. Thus, the assumption $M' \Leftarrow A$ should be ground: $\cdot;\cdot;[\![\rho]\!]\Psi \vdash M' \Leftarrow [\![\rho]\!]A$, where ρ is a grounding substitution: $\cdot;\cdot \vdash \rho : (\Omega;\Delta)$.

In addition, the domain of $\mathcal{S}'$ need not exactly match the domain of $\mathcal{S}$. For instance, the type in $\mathcal{S}'$ will actually be $[\![\theta]\!]A$, where θ is a substitution from Δ to Δ'. Likewise, "$M \Leftarrow A$" must actually be $\Omega;\Delta';[\![\theta]\!]\Psi \vdash M \Leftarrow [\![\theta]\!]A$.

As we have θ from Δ to Δ', and ρ from Δ to ground, the lemma also asserts the existence of a θ' from Δ' to ground, so that $\rho = [\![\theta']\!]\theta$.

We must also reason about App derivations. The rough statement is that if $\mathcal{S} :: \ldots \vdash \mathsf{App}\langle R\rangle(A > P) \rhd \mathcal{J}$ and $R \Rightarrow A$ and $R\ N_1' \ldots N_n' \Leftarrow P$, then $\mathcal{S}' :: \ldots \vdash (R\ N_1' \ldots N_n') : P \rhd \mathcal{J}$. Laying out contexts and substitutions as before, we will actually have some N_1 such that $[\![\rho]\!]N_1 = N_1'$, etc.

Lemma 2 (Coverage Soundness).

(1) If $\mathcal{S} :: \Omega;\Delta;\Psi \vdash \mathsf{Obj}(A) \rhd \mathcal{J}$ *and* $\cdot;\cdot;[\![\rho]\!]\Psi \vdash M' \Leftarrow [\![\rho]\!]A$ *and* $\cdot;\cdot \vdash \rho : (\Omega;\Delta)$
then there exist θ *and* M *such that* $\Delta' \vdash \theta : \Delta$
and $\mathcal{S}' :: \Omega;\Delta';[\![\theta]\!]\Psi \vdash M : [\![\theta]\!]A \rhd [\![\theta]\!]\mathcal{J}$ *where* $\mathcal{S}' < \mathcal{S}$
and $\Omega;\Delta';[\![\theta]\!]\Psi \vdash M \Leftarrow [\![\theta]\!]A$
and there exists θ' *s.t.* $\rho = [\![\theta']\!]\theta$ *and* $M' = [\![\theta']\!]M$.

(2) If $\mathcal{S} :: \Omega;\Delta;\Psi \vdash \mathsf{App}\langle R\rangle(A^* > P) \rhd \mathcal{J}$ *and* $\Omega;\Delta;\Psi \vdash R \Rightarrow A^*$
and $\cdot;\cdot \vdash \rho : \Delta$
and for all spines $N_1', \ldots, N_n'$ *of some length* n *such that*
$\cdot;\cdot;[\![\rho]\!]\Psi \vdash ([\![\rho]\!]R)\ N_1' \ldots N_n' \Leftarrow [\![\rho]\!]P$,
then $\mathcal{S}' :: \Omega;\Delta';[\![\theta]\!]\Psi \vdash [\![\theta]\!](R\ N_1 \ldots N_n) : [\![\theta]\!]P \rhd [\![\theta]\!]\mathcal{J}$
and for all i *we have* $[\![\rho]\!]N_i = N_i'$
and there exists θ' *s.t.* $\rho = [\![\theta']\!]\theta$.

Proof. By complete induction on the height of $\mathcal{S}$.

Theorem 3 (Coverage Soundness).
If $Z = \{\zeta_1, \ldots, \zeta_n\}$ *and for all* j $\cdot;\cdot;\cdot \vdash \zeta_j \Leftarrow_{A[\Psi]} \tau$ *and*
$\cdot;\cdot;\Psi \vdash \mathsf{Obj}(A) \rhd \textsc{covered-by}\ Z$ *and for all* M' *s.t.* $\cdot;\cdot;\Psi \vdash M' \Leftarrow A$ *then there exists* k *such that* $\cdot;\cdot;\cdot \vdash \mathsf{box}(\hat{\Psi}.\, M) \doteq \zeta_i \ /\ (\theta, \cdot)$.

Proof. By Lemma 2, inversion, and correctness of higher order matching.

5 Conclusion

Most previous work on coverage checking, such as Coquand's work [2] in the setting of Agda and later refinements of this approach [5,8], has dealt with closed data objects. In the setting of logical frameworks, theoretical work on coverage also concentrated on closed objects [14]. In contrast, we have presented a framework for coverage checking terms that depend on assumptions in a given context, and proved soundness. Context schemas and parameter variables allow us to analyze generic cases for all objects represented by a context variable.

We have concentrated on the Beluga language, but systems like Delphin and Twelf have to address a very similar issue. In fact, Schürmann [13] presented early work for open data in Twelf. In Twelf, we characterize context schemas by world declarations. However, there is an important difference between worlds and schemas. In Twelf, to count free occurrences of a variable, we would write a relation. But there is no way to write a generic base case for all possible variables occurring in a context represented by ψ. Instead, we must introduce dynamic extensions for each variable encountered when we traverse a binder. Thus, the world declaration not only captures the bound variables introduced when we traverse a binder, but also a base case for each binder. Consequently, some of the base cases are scattered, and world declarations tend to be more complicated than our schema declarations. It also makes world checking significantly more complicated.

The treatment of contexts in Delphin is closely related to ours, though Delphin has no explicit context variables, and parameter variables are missing from the theoretical framework. Nevertheless, we believe this framework is also a foundation for coverage in Delphin.

We plan to implement a coverage algorithm based on the ideas in this paper within the Beluga prototype. We also plan to extend coverage checking to substitution patterns (a feature unique to Beluga), which we expect will be straightforward.

References

1. D. Baelde, A. Gacek, D. Miller, G. Nadathur, and A. Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *21st Conference on Automated Deduction*, number 4603 in LNAI, pages 391–397. Springer, 2007.
2. T. Coquand. Pattern matching with dependent types. In *Informal Proceedings of Workshop on Types for Proofs and Programs*, pages 71–84. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992.

3. A. Gacek, D. Miller, and G. Nadathur. Combining generic judgments with recursive definitions. Draft, arXiv:0802.0865v1 [cs.LO], Jan. 2008.
4. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.
5. C. McBride. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, University of Edinburgh, 2000. Technical Report ECS-LFCS-00-419.
6. A. McCreight and C. Schürmann. A meta-linear logical framework. In *4th International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, 2004.
7. A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 2008. Accepted, to appear.
8. U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.
9. F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer-Verlag LNAI 1632, 1999.
10. B. Pientka. *Tabled higher-order logic programming*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 2003. CMU-CS-03-185.
11. B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and explicit substitutions. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM, 2008.
12. A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *Proceedings of the 17th European Symposium on Programming (ESOP '08)*, Mar. 2008.
13. C. Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 2000. CMU-CS-00-146.
14. C. Schürmann and F. Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science(LNCS)*, pages 120–135, Rome, Italy, 2003. Springer.
15. C. Schürmann, A. Poswolsky, and J. Sarnat. The ∇-calculus. Functional programming with higher-order encodings. In P. Urzyczyn, editor, *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA'05), Nara, Japan*, volume 3461 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.
16. R. Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999. CMU-CS-99-167.
17. K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002.