Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 228 (2009) 69–84
www.elsevier.com/locate/entcs

Case Analysis of Higher-Order Data

Jana Dunfield and Brigitte Pientka

School of Computer Science, McGill University
3480 rue University, Montréal, QC H3A 2A7, Canada
bp@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 for closed data, we consider open data which may depend on some context. Our work may therefore provide insight 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.

Keywords: higher-order abstract syntax, coverage checking

1 Introduction

Over the past decade, programming and reasoning with and about data structures that contain binders has received widespread attention in programming languages and automated reasoning systems. Higher-order abstract syntax (HOAS) is a simple and elegant technique for handling binders. 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))`. This avoids the need to implement common and tricky machinery 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 (Fig. 1), the hypothetical typing derivation for implication introduction can be elegantly modeled using higher-order functions.

The power of HOAS encodings has been shown within the logical framework LF [5] and its implementation in Twelf [13]. Recently, HOAS encodings are supported in functional programming languages such as Elphin [17], Delphin [14], and Beluga [12]. In these systems, we analyze higher-order data using pattern matching and case

doi:10.1016/j.entcs.2008.12.117

$$\begin{array}{rcl} \text{Numbers} & N, M ::= & x \\ & & \mid 0 \\ & & \mid \mathsf{suc}\ N \\ \text{Propositions} & A ::= & N = M \\ & & \mid A \supset B \\ & & \mid \forall x.A \end{array}$$

$$\text{Natural Deduction} \quad \Gamma \vdash \mathsf{nd}\ A$$

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

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

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

```
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 N).
```

Fig. 1. Natural deduction and its HOAS encoding

expressions. This requires us to validate that the patterns are exhaustive Similarly, proof assistants for HOAS-based reasoning that split a goal into different cases must ensure that the cases are exhaustive. This issue arises in Twelf's induction theorem prover [15], and in systems such as Bedwyr [1] and Abella [4].

In the first-order, simply-typed setting, analyzing data by cases is straightforward. We can just consider all declared constants of a given type. To illustrate, in Figure 1 we define a simple logic with equality on numbers in the usual style of LF [5]. The cases for the proposition A are clear: they are exactly the three proposition forms listed in the grammar. However, for numbers we do not just need cases for 0 and $\mathsf{suc}\ N$, but also a case for a variable x. A similar situation comes up with higher-order data, such as derivations in natural deduction. An encoding based on higher-order abstract syntax does not represent the rule Hyp explicitly. Instead, this base case will be implicit. Thus, generating all cases requires that we consider the context and its possible elements.

Our main contribution is a theoretical framework for generating an exhaustive set of cases for objects that may refer to assumptions, i.e. open objects. Previous work on coverage checking handled closed terms [2,16], or open terms within regular worlds [15, pp. 197–213]. Our work is the first theoretical treatment of coverage in the setting of contextual modal type theory. We believe our theory is a first step toward demystifying coverage checking in Twelf, an operation that is mysterious to many users. More immediately, our work is a foundation for languages such as Beluga [12] that case-analyze open data. We prove a property of *coverage soundness* that is needed to prove progress in Beluga.

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

```
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] V[id_ψ,x]) ⇒ cntV ⌈ψ⌉ box(ψ,x. U[id_ψ,x])
                                    + cntV ⌈ψ⌉ box(ψ,x. V[id_ψ,x])
| box(ψ,x. forall(λy.W[id_ψ,x,y])) ⇒ cntV⌈ψ,y:nat⌉ box(ψ,y,x. W[id_ψ,x,y])
```

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

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 [12] 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` of type `nat`. 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 form is $\mathtt{x}_1\mathtt{:nat,}\ldots\mathtt{,x}_n\mathtt{:nat}$. We use single capital letters `U`, `V`, `W` for contextual variables, which are instantiated via higher-order pattern matching.

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` of 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-level terms (data objects) from computation-level terms. Since formulas are constructed by equality `eq`, implication `imp` and quantification `forall`, we have cases for each of these.

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 id_ψ 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. Thus, in `U[`id_ψ`]` the contextual variable `U` can be instantiated with any formula that either is closed (does not refer to any bound variable in the context ψ) or contains a bound variable from ψ. Since subformulas can refer to all variables in ψ`,x:nat`, we write `U[`id_ψ`, x]`.

In the first case, for `eq`, we call `cntVN` to count the occurrences of `x` in the natural numbers `U[`id_ψ`,x]` and `V[`id_ψ`, x]`, explicitly passing ψ with `cntV` $\lceil\psi\rceil$.

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[`id_ψ`,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 variable x appears last in `box(`ψ`,y,x. ...)`, to match the argument type `o[...,x:nat]`.

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[`id_ψ`])`, 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 data-level variable. The substitution id_ψ 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 `cntVN` 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_ψ`,x] V[`id_ψ`,x], imp U[`id_ψ`,x] V[`id_ψ`,x], forall (`λy`.U[`id_ψ`,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 higher-order pattern matching to enforce variable dependencies, refining the pattern `eq U[`id_ψ`,x] V[`id_ψ`,x]` into the four cases

{`eq U[`id_ψ`] V[`id_ψ`]`, `eq U[`id_ψ`,x] V[`id_ψ`]`, `eq U[`id_ψ`] V[`id_ψ`,x]`, `eq U[`id_ψ`,x] V[`id_ψ`,x]`}

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	::=	$c \mid x \mid u[\sigma] \mid p[\sigma] \mid R\ N \mid \mathsf{proj}_1\ R \mid \mathsf{proj}_2\ R$
Substitutions	σ	::=	$\cdot \mid \sigma\,;\ M \mid \sigma\ ,\ R \mid \mathsf{id}_\psi$
Contexts	Ψ, Φ	::=	$\cdot \mid \psi \mid \Psi, x{:}A$
Meta-contexts	Δ	::=	$\cdot \mid \Delta, u{::}A[\Psi] \mid \Delta, p{::}A[\Psi]$
Schema contexts	Ω	::=	$\cdot \mid \Omega, \psi{::}W$

Fig. 3. The data level

exactly distinguishing (1) `x` occurs in neither $\mathtt{U[id}_\psi\mathtt{]}$ nor $\mathtt{V[id}_\psi\mathtt{]}$, (2) `x` occurs in $\mathtt{U[id}_\psi\mathtt{,x]}$ but not in $\mathtt{V[id}_\psi\mathtt{]}$, (3) `x` occurs in $\mathtt{V[id}_\psi\mathtt{,x]}$ but not in $\mathtt{U[id}_\psi\mathtt{]}$, and (4) `x` occurs in both $\mathtt{U[id}_\psi\mathtt{,x]}$ and $\mathtt{V[id}_\psi\mathtt{,x]}$.

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 emphasize that while we illustrate the problem in the setting of Beluga, where contexts are explicit, the problem is similar in systems such as Delphin and Twelf, where we also must generate all objects of type A in a context Ψ.

3 Background

Since we are interested in testing whether a set of patterns covers a given data object, we concentrate on the data level. For the computation level, see [12].

We support the logical framework LF plus dependent pairs Σ. Our data layer closely follows contextual modal type theory [10], extended with parameter variables and context variables [12], and finally with Σ types. Perhaps most importantly, we formalize schemas, which classify contexts. We only characterize normal terms since only these are meaningful in the logical framework [18,10]. This is achieved by a syntactic distinction between normal terms M and neutral terms R. The syntax guarantees that terms contain no β-redexes, and the typing rules guarantee that all well-typed terms are fully η-expanded.

We distinguish between three[1] kinds of variables (Figure 3): *Ordinary bound variables* x and y are used to represent data-level binders and are bound by λ-abstraction. 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 can be instantiated via pattern matching. They 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

[1] Prior work also considered substitution variables, which we omit here for brevity.

the free variables of that object, and the type system statically guarantees this. Contextual variables are declared in a meta-level context Δ.

Our foundation supports *context variables* ψ which allow us to reason abstractly with contexts, and write recursive computations that manipulate higher-order data. Unlike some other uses of context variables [8], a context may contain at most one context variable [2]. As types classify objects, and kinds classify types, we introduce the notion of *schemas* W that classify contexts Ψ. Context variables' schemas are given in a schema context Ω. We define schemas in Section 3.2.

Substitutions σ are built of normal terms (in $\sigma\,;\,M$) and atomic terms (in $\sigma\,,\,R$). We do not make the domain explicit, which simplifies the theoretical development and avoids having to rename the domain of a given σ. We also have a first-class notion of identity substitution id_ψ. We write $[\sigma]N$ for substitution application.

We assume that type constants and object constants are declared in a signature S as pure LF objects—data of dependent function type. We suppress the signature since it is the same throughout all derivations. As a notational convenience, we generalize pairs to n-ary tuples, writing $\mathsf{proj}_k^{\#}\ R$ for the kth projection of R. For example, the second element of a triple is $\mathsf{proj}_2^{\#}\ R = \mathsf{proj}_1(\mathsf{proj}_2\ R)$.

3.1 Data-level typing

We type data-level terms bidirectionally. Normal objects are checked against a given type in the judgment $\Omega;\Delta;\Psi \vdash M \Leftarrow A$, while neutral objects synthesize their type: $\Omega;\Delta;\Psi \vdash R \Rightarrow A$. Substitutions are checked against their domain: $\Omega;\Delta;\Psi \vdash \sigma \Leftarrow \Phi$. For readability, we omit the schema context Ω in the subsequent development since it is constant, and assume that Δ and Ψ are well-formed.

We give the typing rules for data-level terms in Figure 4. 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 Π 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 neutral terms use *hereditary substitutions* $[\cdots]_A^a$ which preserve canonical forms [10]. 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. Due to lack of space, we relegate the details to [3, appendix]. We omit the subscripts for readability in what follows.

Since hereditary substitution is decidable and the rules in Figure 4 are syntax-directed, data-level typing is decidable.

[2] Lifting this restriction would require tracking dependencies of context variables on each other: in $\psi, x{:}A, \psi'$, the context substituted for ψ' could depend on x or even on variables in ψ. Ensuring that α-renaming holds in the presence of multiple context variables and dependent types appears difficult.

Data-level normal terms

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

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

Data-level neutral terms

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

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

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

Data-level substitutions

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

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

Fig. 4. Data-level typing and substitutions

3.2 Context schemas

As the earlier example illustrated, contexts play an important part in programming with open data objects. In particular, any contexts that are explicitly constructed and passed will belong to a specific context *schema*. In the earlier example, the schema $(\mathtt{nat})^*$ represented contexts of the form $x_1{:}\mathtt{nat}, \ldots, x_n{:}\mathtt{nat}$. But we allow much more expressive contexts. For instance, when reasoning about natural deductions, the rule $\supset I^u$ adds an assumption of the form $u{:}(\mathtt{nd}\ A)$ for some concrete proposition A. The inductive definition $\Gamma' ::= \cdot \mid \Gamma', x{:}\mathtt{nat}, \mid \Gamma', u{:}(\mathtt{nd}\ A)$ corresponds to the schema $(\mathtt{nat} + (\mathsf{all}\ A{:}\mathtt{o}.\,\mathtt{nd}\ A))^*$.

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 5. Schemas are built of elements $F_1, \ldots, F_n$, each of the form $\mathsf{all}\,\widetilde{\Theta}.\,\Sigma y_1{:}\widetilde{B}_1, \ldots, y_j{:}\widetilde{B}_j.\,\widetilde{b}$, where $\widetilde{\Theta} = x_1{:}\widetilde{C}_1, \ldots x_k{:}\widetilde{C}_k$. In other words, for any instantiation of $\widetilde{\Theta}$ (that is, any substitution for $x_1, \ldots, x_k$), the element is of $\Sigma\Pi$-type, where we first introduce some Σs, followed by Πs, with no subsequent Σs. This restriction makes it easier to describe the inhabitants of the type. Twelf has a similar restriction on worlds. In Beluga, computation typing [12] guarantees 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 each element $x{:}A$ in Ψ is an instance of a schema element $F_k = \mathsf{all}\,\widetilde{\Theta}.\,\Sigma y_1{:}\widetilde{B}_1, \ldots, y_j{:}\widetilde{B}_j.\,\widetilde{B}$, with

$$\begin{array}{lrcl} \text{Element types} & \widetilde{A} & ::= & \Pi x{:}A.\widetilde{A} \mid a\ N_1 \dots N_n \\ \text{Schema elements} & F & ::= & \mathsf{all}\, x_1{:}\widetilde{B}_1, \dots x_k{:}\widetilde{B}_k.\, \Sigma y_1{:}\widetilde{A}_1, \dots, y_j{:}\widetilde{A}_j.\widetilde{A} \\ \text{Schemas} & W & ::= & (F_1 + \dots + F_n)^* \end{array}$$

Context Ψ checks against schema W

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

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

$$\frac{\widetilde{\Theta} = x_1{:}\widetilde{C}_1, \dots, x_n{:}\widetilde{C}_n \qquad \sigma = u_1[\mathsf{id}(\Psi)]/x_1, \dots, u_n[\mathsf{id}(\Psi)]/x_n \\ \Omega; \Delta, u_1{::}\widetilde{C}_1[\Psi], \dots, u_n{::}\widetilde{C}_n[\Psi]; \Psi \vdash A \doteq [\sigma]\Sigma\widetilde{\Phi}.\widetilde{B} \;/\; (\theta, \Delta)}{\Omega; \Delta; \Psi \vdash A \in \mathsf{all}\, \widetilde{\Theta}.\, \Sigma\widetilde{\Phi}.\, \widetilde{B}}$$

Fig. 5. Schemas

all variables in $\widetilde{\Theta}$ instantiated such that $x{:}A$ is an instance of F_k. The rule in Figure 5 uses higher-order pattern matching. The judgment $A \doteq B \;/\; (\theta, \Delta)$ means that θ is a substitution such that $[\![\theta]\!]B = A$.

4 Coverage checking

In this section, we present a theory for coverage checking. A derivation of a coverage judgment is a proof that every closed term of a given 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. Any set of patterns covers all terms of an empty type, and emptiness is undecidable [6, p. 179]. In Beluga, empty types should be very rare. In any case, since any algorithm must be incomplete, completeness of the theory is not essential.

Coquand [2] and Schürmann and Pfenning [16] described coverage checking for closed terms, while Schürmann [15, pp. 197–213] formulated coverage for open terms within regular worlds. Our theoretical treatment of coverage is the first in the setting of contextual modal type theory, where objects are closed with respect to explicit contexts that include context variables. This leads to a clean development of coverage.

To see that a set of patterns Z (in Beluga, the guards of a case expression) covers a given type, we usually need to *split* the type into an equivalent set of more precise patterns. To see that $Z = \{\mathtt{Zero}, \mathtt{Suc}\ u\}$ covers all (closed) terms of type $\mathtt{nat}[\cdot]$, we need to split $\mathtt{nat}[\cdot]$ into the pattern set $Z' = \{\mathtt{Zero}, \mathtt{Suc}\ u_1\}$. Now it is obvious that Z covers $\mathtt{nat}[\cdot]$, because Z'—the result of splitting $\mathtt{nat}[\cdot]$—is α-equivalent to Z.

More generally, suppose we want to check that Z covers $\mathtt{nat}[\Psi]$. If $\Psi \neq \cdot$, we are dealing with open data, so when we split, we must consider variables as well as constructors. Suppose the type is $\mathtt{nat}[\psi, x{:}\mathtt{nat}, y{:}\mathtt{o}]$, where ψ represents a context of schema $(\mathtt{o} + \mathtt{nat})^*$. The split then includes the constructors, parameter

variables denoting the generic case for variables from ψ (one variable for each schema element), and the concrete variables x and y:

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

Not all of the variables are actually possible: $p_1[\mathsf{id}_\psi]$ is of type $\texttt{o}$, but we are analyzing type $\texttt{nat}$. The concrete variable y is similarly impossible. This gives the set $\{\texttt{Zero},\ \texttt{Suc}\ u[\mathsf{id}_\psi, x, y],\ p_2[\mathsf{id}_\psi],\ x\}$.

For some sets Z we would also need to split $\texttt{Suc}$'s argument $u[\mathsf{id}_\psi, x, y]$ into *its* constituent constructors and variables. Decisions about when to split are not determined by our theory; such decisions are embodied in a nondeterministic choice between rules Obj-split and Obj-no-split. Our system is thus the *foundation* for a coverage checking algorithm.

After some remarks on substitutions and higher-order pattern unification, we state some key metatheoretical results, and then describe the coverage rules.

We write $[\![\theta]\!]$ for a contextual substitution substituting for u and p variables in Δ. The judgment $\Omega; \Delta' \vdash \theta \Leftarrow \Delta$ says that θ is a contextual substitution with domain Δ and range Δ', under the schema context Ω. We write ρ as an abbreviation for (1) a context substitution on the schema context Ω, substituting for context variables ψ, and (2) a contextual substitution θ. The judgment $\Omega'; \Delta' \vdash \rho : (\Omega; \Delta)$ says that the domain of ρ is $(\Omega; \Delta)$ and its range is $\Omega'; \Delta'$. In the rules, we write data-level substitutions as $[M/x]A$. This is actually hereditary substitution, but we omit the types. See [3, appendix] for details.

We allow higher-order patterns in the sense of Miller [7], in which instantiated meta-variables must be applied to distinct sets of bound variables. Thus, contextual variables are associated with a substitution such as $x_{\Phi(1)}/x_1, \ldots, x_{\Phi(n)}/x_n$. Matching is decidable and efficient [11]. The proof of the following is a simple extension of the one in [11].

Theorem 4.1 (Soundness of higher-order pattern unification) *If P and Q are well-formed types under $\Omega; \Delta; \Psi$, 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 ρ' such that $\rho = [\![\rho']\!]\theta$.*

Lemma 4.2 (Object inversion) *If $\cdot; \cdot; \Psi \vdash R \Leftarrow P$ and $\vdash \Psi : W$ then either*

(1) $R = c\ N_1\ \ldots\ N_k$ where $\mathsf{S}(c) = \Pi x_1{:}A_1. \cdots \Pi x_k{:}A_k.P'$ and $[\sigma]P' = P$, or
(2) $R = x\ N_1\ \ldots\ N_k$ where $(x : \Pi x_1{:}A_1. \cdots \Pi x_k{:}A_k.P') \in \Psi$ and $[\sigma]P' = P$, or
(3) $R = (\mathsf{proj}_l^{\#}\ y)\ N_1\ \ldots\ N_k$ where $(y : \Sigma y_1{:}\widetilde{A}_1, \ldots y_m{:}\widetilde{A}_m.\widetilde{A}_{m+1}) \in \Psi$ and $[\sigma]P' = P$ and $[\mathsf{proj}_1^{\#}\ y/y_1, \ldots, \mathsf{proj}_l^{\#}\ y/y_l]\widetilde{A}_{l+1} = \Pi x_1{:}B_1. \ldots \Pi x_k{:}B_k.P'$ where $1 \leq l \leq m$,

where $\sigma = N_1/x_1, \ldots, N_k/x_k$.

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

4.1 Overview of coverage judgments

Given the set of guards in a case expression, Z, we assume each pattern $\zeta \in Z$ has the form $\Pi\Delta'.\ \mathsf{box}(\hat{\Psi}.\ M) : A[\Psi']$, where Δ' gives the types of contextual variables u and p in M (which will be bound to objects and variables, respectively, when a case expression is evaluated), where M has type $A[\Psi']$. Thus, a pattern in a case expression is not simply $\mathtt{Suc}\ u[\mathsf{id}_\psi, x]$, but $\Pi u{::}\mathtt{nat}[\psi, x{:}\mathtt{nat}].\ \mathsf{box}(\psi, x.\ \mathtt{Suc}\ u[\mathsf{id}_\psi, x]) : \mathtt{nat}[\psi, x{:}\mathtt{nat}]$. In this example, and in many situations, Δ' and $A[\Psi']$ could be omitted in the source program and reconstructed. However, a dependently-typed Δ' such as $u{::}(\mathtt{nd}\,(\mathtt{eq}\,x\,x))[x{:}\mathtt{nat}]$ actually restricts u to match only natural-deduction proofs of $\mathtt{eq}\,x\,x$. Similarly, a dependently-typed A can constrain the entire pattern.

The most essential coverage judgment, $\Omega; \Delta; \Psi \vdash \mathsf{Obj}(A) \ \rhd\ \textsc{covered-by}\, Z$, means that every object of type A is matched by at least one pattern in Z. For example, if we have a derivation of $\Omega; \cdot; \psi, x{:}\mathtt{nat}, y{:}\mathtt{o} \vdash \mathsf{Obj}(\mathtt{nat}) \ \rhd\ \textsc{covered-by}\, Z$ then Z covers the type $\mathtt{nat}[\psi, x{:}\mathtt{nat}, y{:}\mathtt{o}]$.

Such a derivation has subderivations of the general form $\Omega; \Delta; \Psi \vdash \mathsf{Obj}(A) \ \rhd\ \mathcal{J}$, which analyzes A and gives the result as input to $\mathcal{J}$, which is (algorithmically) a kind of continuation. The earlier judgment is an instance of this form: it analyzes A and then "continues with" $\textsc{covered-by}\, Z$.

The splitting operation discussed earlier manifests as subderivations of $\Omega; \Delta; \Psi \vdash M : A \ \rhd\ \mathcal{J}$. Here, M is a term that plays the role of a pattern, with free variables $u[\sigma]$. Omitting contexts for clarity, a derivation where $A = \mathtt{nat}[\cdot]$ would look like

$$\dfrac{\overbrace{\mathtt{Zero}}^{M_1} : \overbrace{\mathtt{nat}[\cdot]}^{A_1} \ \rhd\ \mathcal{J} \qquad \overbrace{\mathtt{Suc}\ u[\cdot]}^{M_2} : \overbrace{\mathtt{nat}[\cdot]}^{A_2} \ \rhd\ \mathcal{J}}{\mathsf{Obj}(\mathtt{nat}[\cdot]) \ \rhd\ \mathcal{J}}$$

In general, $M_1, \ldots, M_n$ collectively cover all possible terms of type A. That is, the subderivations correspond to a split into n patterns. In the example, $n = 2$.

4.2 COVERED-BY*: the leaves of a coverage derivation*

We said that $\mathsf{Obj}(A) \ \rhd\ \textsc{covered-by}\, Z$ means to analyze A and "continue with" $\textsc{covered-by}\, Z$. So, having analyzed A, splitting as necessary, we eventually come to subderivations of $M_k : A_k \ \rhd\ \textsc{covered-by}\, Z$. These are the outermost branches of the derivation tree, and are the only places where Z is examined. Such subderivations all have the same structure: $\mathsf{Covered\text{-}By}\text{-}Z$ picks out one pattern ζ from the

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

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

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

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

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

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

for $0 \le i \le m$:

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

$$\boxed{\Omega; \Delta; \Psi \vdash M : A \;\rhd\; \mathcal{J}} \qquad \frac{\Omega \vdash \Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M) : A[\Psi] \text{ COVERED-BY } \zeta_k}{\Omega; \Delta; \Psi \vdash M : A \rhd \text{COVERED-BY } \{\zeta_1, \ldots, \zeta_n\}}\ \textsf{Covered-By-}Z$$

$$\frac{\Omega; \Delta; \Psi \vdash (\lambda x.\, M) : (\Pi x{:}A_1.A_2) \rhd \mathcal{J}}{\Omega; \Delta; \Psi, x{:}A_1 \vdash M : A_2 \rhd \text{LAM} \rhd \mathcal{J}} \qquad \frac{\Omega; \Delta; \Psi \vdash (M, N) : \Sigma x{:}A_1.A_2 \;\rhd\; \mathcal{J}}{\Omega; \Delta; \Psi \vdash N : [M/x]A_2 \rhd \text{PAIR2}\,(M{:}A_1, x.\bullet) \rhd \mathcal{J}}$$

$$\frac{\Omega; \Delta; \Psi \vdash \mathsf{Obj}([M/x]A_2) \rhd \text{PAIR2}\,(M{:}A_1, x.\bullet) \rhd \mathcal{J}}{\Omega; \Delta; \Psi \vdash M : A_1 \rhd \text{PAIR1}\,(\bullet,\, x.A_2) \rhd \mathcal{J}}$$

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

$$\frac{\Omega; \Delta; \Psi, x{:}A_1 \vdash \mathsf{Obj}(A_2) \rhd \text{LAM} \rhd \mathcal{J}}{\Omega; \Delta; \Psi \vdash \mathsf{Obj}(\Pi x{:}A_1.A_2) \;\rhd\; \mathcal{J}}\textsf{Obj-}\Pi \qquad \frac{\Omega; \Delta; \Psi \vdash \mathsf{Obj}(A_1) \rhd \text{PAIR1}\,(\bullet,\, x.A_2) \rhd \mathcal{J}}{\Omega; \Delta; \Psi \vdash \mathsf{Obj}(\Sigma x{:}A_1.A_2) \;\rhd\; \mathcal{J}}\textsf{Obj-}\Sigma$$

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

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

Fig. 6. Coverage checking rules

set Z, and then Covered-By-ζ checks that M_k is an instance of ζ.[3]

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

We assume that the pattern ζ includes an explicit meta-variable context Δ', explicit data-level names $\hat{\Psi}'$, and an explicit type $A'[\Psi']$. Thus, the premise of Covered-By-ζ is $\Omega \vdash (\Pi\Delta.\mathsf{box}(\hat{\Psi}.\, M_k) : A_k[\Psi]) \doteq (\Pi\Delta'.\mathsf{box}(\hat{\Psi}'.\, M') : A'[\Psi']) \;\; / \; (\theta, \Delta)$. This says that M_k is an instance of M' realized by θ, that is, $M_k = [\![\theta]\!]M'$. If each M_k is an instance of some pattern in Z, then Z covers all inhabitants of A.

4.3 Rules deriving $\mathsf{Obj}(A) \;\rhd\; \mathcal{J}$

Having explained the high-level structure of coverage derivations and the details of the leaves, we can discuss the rules with conclusions of the form $\mathsf{Obj}(A) \rhd \mathcal{J}$. These are the four rules at the bottom of Figure 6.

If $A = \Pi x{:}A_1.A_2$, we use Obj-Π to peel off the Π and analyze A_2. The LAM is added because after analyzing A_2, we need to put back the Π and add a λ:

$$\dfrac{\begin{array}{c}\Omega; \Delta; \Psi \vdash (\lambda x.\, M) : (\Pi x{:}A_1.A_2') \;\rhd\; \mathcal{J} \\ \hline \Omega; \Delta; \Psi \vdash M : A_2' \;\rhd\; \text{LAM} \;\rhd\; \mathcal{J} \\ \vdots \\ \Omega; \Delta; \Psi, x{:}A_1 \vdash \mathsf{Obj}(A_2) \;\rhd\; \text{LAM} \;\rhd\; \mathcal{J}\end{array}}{\Omega; \Delta; \Psi \vdash \mathsf{Obj}(\Pi x{:}A_1.A_2) \;\rhd\; \mathcal{J}}\;\textsf{Obj-}\Pi$$

Note that since splitting A_2 may produce several patterns, we may have more subderivations $(\lambda x.\, \ldots) : (\Pi x{:}A_1.\cdots)$ than just the one shown.

If $A = \Sigma x{:}A_1.A_2$, rule Obj-Σ first analyzes A_1 and then A_2. The rules in Figure 6 are laid out vertically, in the same order as they appear in a derivation.

For base types P, we can either not split (rule Obj-no-split) or split (rule Obj-split). The latter rule is less complicated than it may look. The point is to split into patterns $R\ N_1 \ldots N_m$, where R is a parameter $p[\sigma]$ (left-hand premises), variable x (upper-right-hand premises), or constructor c (lower-right-hand premises),

The simplest of these are the premises $\mathsf{App}\langle c_k\rangle(\mathsf{S}(c_k) > P)$ for constructors c. These cover all constructors c_k, even those for base types that are incompatible with P—those will be discarded further up the derivation.

Deriving premises of the form $\Omega; \Delta; \Psi \vdash \mathsf{App}\langle R\rangle(\mathsf{S}(c_k) > P) \;\rhd\; \mathcal{J}$ is somewhat involved, since we need to generate all spines (lists of arguments) $N_1 \ldots N_m$. Here, the P denotes that we are constructing objects of type P. The constructor type $\mathsf{S}(c_k)$ must have the form $\Pi x_1{:}A_1.\cdots\Pi x_m{:}A_m.Q$, where Q is a base type. In deriving

[3] Note that we need matching, not just equality, in Covered-By-ζ. Suppose $Z = \{(u_1[\cdot], \texttt{Zero}), (\texttt{Zero}, u_2[\cdot])\}$. To show that $(\texttt{Zero}, \texttt{Suc}\ v_2[\cdot])$ is covered (by the second pattern in Z), we need to split the first component, and to show that $(\texttt{Suc}\ v_1[\cdot], \texttt{Zero})$ is covered (by the first pattern in Z), we need to split the second component. This results in a set of patterns including $(\texttt{Zero}, \texttt{Zero})$, which is not equal to any pattern in Z.

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

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

Fig. 7. Coverage checking rules (continued)

this, we use App-Π, which uses $\mathsf{Obj}(A_1)$ to analyze A_1, and (through NEUTRAL) adds the resulting inhabitants M_1 of A_1 to c_k.

Doing this for each $x_i{:}A_i$ yields subderivations of $\mathsf{App}\langle c_k\ N_1 \ldots N_m\rangle(Q > P)$, for various spines $N_1 \ldots N_m$. If Q and P do not unify (written $Q \not\doteq P$ in rule App-$\not\doteq$) we have a trivial coverage subderivation, but if Q and P do unify under some θ, then we can use App-$\doteq$, which has a premise $[\![\theta]\!]R : [\![\theta]\!]P \ \rhd\ [\![\theta]\!]\mathcal{J}$.

Returning to rule Obj-split itself, the premises $\mathsf{App}\langle x_k\rangle(B > P) \rhd \mathcal{J}$ for variables are structurally similar to those for constructors. However, unlike $\mathsf{S}(c_k)$, the variable type B could contain Σs, so we use App-Σ to take projections out of the tuple.

The remaining premises of Obj-split have the form $\mathsf{PVars}\langle \psi : F\rangle > P \ \rhd\ \mathcal{J}$, characterizing the generic variable cases.

4.4 $\mathsf{PVars}\langle \psi : F\rangle > P \ \rhd\ \mathcal{J}$: *Parameter variables*

Exactly one rule concludes $\mathsf{PVars}\langle \ldots\rangle$, the rule PVars in Figure 7. In PVars, we generate a parameter variable for each schema element. We first create a meta-variable for each all-quantified variable in the element. For example, if $F = \mathsf{all}\, A{:}\mathtt{o}.\mathtt{nd}\, A$, then $p[\mathsf{id}_\psi]$ 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 $\mathsf{all}\, \widetilde{\Theta}.\Sigma\widetilde{\Phi}.\widetilde{A}$ by generating a substitution σ' that instantiates all variables in $\widetilde{\Theta}$ with meta-variables, and applying σ' to $\Sigma\widetilde{\Phi}.\widetilde{A}$. Then we use the ideas for concrete variables. Again, since $[\sigma']\Sigma\widetilde{\Phi}.\widetilde{A}$ is inhabited by tuples, we consider all possible projections.

4.5 $\mathsf{MVars}(P) \ \rhd\ \mathcal{J}$: *General case for all ground instances of* P

The premise of rule Obj-no-split is $\mathsf{MVars}(P)$, which is derivable only by rule MVars (Figure 7). This rule does not recursively analyze the given type P. Instead, it

produces patterns $u[\mathsf{id}(\Psi_k)]$, which any object of type $P[\Psi_k]$ matches. [4]

Simply generating $u[\mathsf{id}(\Psi)]$ does not suffice if the user wrote cases with different contexts, as when $\mathtt{eq\ U[id}_\psi, x\mathtt{]\ V[id}_\psi, x\mathtt{]}$ is written as four cases $\{\mathtt{eq\ U[id}_\psi\mathtt{]\ V[id}_\psi\mathtt{]}$, $\mathtt{eq\ U[id}_\psi, x\mathtt{]\ V[id}_\psi\mathtt{]}$, $\mathtt{eq\ U[id}_\psi\mathtt{]\ V[id}_\psi\mathtt{,}x\mathtt{]}$, $\mathtt{eq\ U[id}_\psi, x\mathtt{]\ V[id}_\psi, x\mathtt{]}\}$.

In fact, we generate all valid weakenings of Ψ. A *weakening* $\Psi' \subseteq \Psi$ has zero or more assumptions from Ψ (preserving order). These contexts are weaker because they provide less information. Not all weakenings make sense; for example, removing x:nat from $(x$:nat$, y$:(eq x x)) yields $(y$:(eq x x)), which is dependent on an undeclared x. The *valid* weakenings $\mathsf{ValidWk}(\Omega; \Delta \vdash A[\Psi])$ of a context Ψ with respect to a type A are those that are well-formed and make A well-formed.

4.6 Coverage soundness

Roughly, the soundness result we need is that, if $\cdot;\cdot;\Psi \vdash \mathsf{Obj}(A) \ \triangleright\ \textsc{covered-by}\ Z$, 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, which will guarantee that if $\mathcal{D}$ derives $\mathsf{Obj}(A) \ \triangleright\ \mathcal{J}$ then, for every ground M' of type A, there is within $\mathcal{D}$ a derivation of $M_i : A \ \triangleright\ \mathcal{J}$, where M' is an instance of M_i. Put another way, the lemma states that the illustration from Section 4.1 is accurate.

Once we have this lemma, soundness is straightforward: if $\mathcal{J} = \textsc{covered-by}\ Z$, the lemma gives a subderivation $\mathcal{D}'$ of $\ldots \vdash M : A \ \triangleright\ \textsc{covered-by}\ Z$, and inversions bring us to the premise of $\mathsf{Covered\text{-}By\text{-}}Z$.

To state the lemma precisely, we first observe that the judgment form $\Omega; \Delta; \Psi \vdash \mathsf{Obj}(A) \ \triangleright\ \mathcal{J}$ allows for nonempty Ω and Δ. However, at runtime, we only have concrete contexts, so Ω is empty. Also, objects are ground, containing no contextual variables u and p, so Δ is empty. We can of course have a nonempty Ψ, though since Ω is empty, Ψ will contain no context variables.

Thus, the antecedent that M' has type A can be ground: $\cdot;\cdot;[\![\rho]\!]\Psi \vdash M' \Leftarrow [\![\rho]\!]A$, where Ω and Δ are grounded by $\cdot;\cdot \vdash \rho : (\Omega; \Delta)$. In addition, the domain of $\mathcal{D}'$ need not exactly match the domain of $\mathcal{D}$. In fact, the type in $\mathcal{D}'$ will be $[\![\theta]\!]A$, where θ is a substitution from Δ to Δ'. This is consistent with the intuition that types become more precise as we move into subderivations.

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

In part (2) of the lemma, we reason correspondingly about App derivations.

Lemma 4.3 (Coverage Soundness)

(1) If $\mathcal{D} :: \Omega; \Delta; \Psi \vdash \mathsf{Obj}(A) \ \triangleright\ \mathcal{J}$ *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* $\Omega; \Delta' \vdash \theta \Leftarrow \Delta$
and $\mathcal{D}' :: \Omega; \Delta'; [\![\theta]\!]\Psi \vdash M : [\![\theta]\!]A \ \triangleright\ [\![\theta]\!]\mathcal{J}$ *where* $\mathcal{D}' < \mathcal{D}$
and $\Omega; \Delta'; [\![\theta]\!]\Psi \vdash M \Leftarrow [\![\theta]\!]A$ *and there exists* ρ' *s.t.* $\rho = [\![\rho']\!]\theta$ *and* $M' = [\![\rho']\!]M$.

(2) If $\mathcal{D} :: \Omega; \Delta; \Psi \vdash \mathsf{App}\langle R\rangle(\widetilde{A} > P) \ \triangleright\ \mathcal{J}$ *and* $\Omega; \Delta; \Psi \vdash R \Rightarrow \widetilde{A}$
and $\cdot;\cdot \vdash \rho : (\Omega; \Delta)$ *and for all spines* $N'_1, \ldots, N'_n$ *of some length* n *such that*

[4] The operation $\mathsf{id}(\Psi)$ unrolls Ψ. For example, $\mathsf{id}(\psi, x$:nat$) = \mathsf{id}_\psi, x$. See [3, appendix] for details.

$\cdot;\cdot;[\![\rho]\!]\Psi \vdash ([\![\rho]\!]R)\ N'_1 \ldots N'_n \Leftarrow [\![\rho]\!]P$,
then $\mathcal{D}' :: \Omega;\Delta';[\![\theta]\!]\Psi \vdash [\![\theta]\!](R\ N_1 \ldots N_n) : [\![\theta]\!]P\ \rhd\ [\![\theta]\!]\mathcal{J}$
and for all i *we have* $[\![\rho]\!]N_i = N'_i$ *and there exists* ρ' *s.t.* $\rho = [\![\rho']\!]\theta$.

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

Theorem 4.4 (Coverage Soundness)
If $\cdot;\cdot;\Psi \vdash M' \Leftarrow A$ *and* $\cdot;\cdot;\Psi \vdash \mathsf{Obj}(A)\ \rhd$ COVERED-BY Z
then there exists $\zeta \in Z$ *such that* $\zeta = (\Pi\Delta_k.\mathsf{box}(\hat{\Psi}.\, M_k) : A_k[\Psi_k])$
and $\cdot \vdash (\Pi\Delta'.\mathsf{box}(\hat{\Psi}.\, M) : A[\Psi]) \doteq \zeta\ /\ (\theta_k, \Delta')$ *where* $M' = [\![\rho']\!][\![\theta_k]\!]M_k$.

Proof. By Lemma 4.3, 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 [6,9], dealt with closed data objects. In the setting of logical frameworks, theoretical work on coverage also concentrated on closed objects [16]. In contrast, we have presented a framework for coverage checking terms that depend on assumptions in a given 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 Twelf, contexts are characterized 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 and coverage checking significantly more complicated.

Delphin has no explicit context variables and distinguishes parameters at the type level, rather than the syntax level. Nevertheless, we believe our framework could provide insights into the Delphin coverage checker [14] as well.

We plan to implement a coverage algorithm based on the ideas in this paper within the Beluga prototype.

References

[1] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The Bedwyr system for model checking over syntactic expressions. In Frank Pfenning, editor, *21st Conference on Automated Deduction*, number 4603 in LNAI, pages 391–397. Springer, 2007.

[2] Thierry 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] Jana Dunfield and Brigitte Pientka. Case analysis of higher-order data. Extended online version of this paper. http://complogic.cs.mcgill.ca/beluga, June 2008.

[4] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23rd Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2008.

[5] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[6] Conor McBride. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, University of Edinburgh, 2000. Technical Report ECS-LFCS-00-419.

[7] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[8] Andrew McCreight and Carsten Schürmann. A meta-linear logical framework. In *4th International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, 2004.

[9] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

[10] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3), 2008.

[11] Brigitte Pientka. *Tabled higher-order logic programming*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 2003. CMU-CS-03-185.

[12] Brigitte Pientka. 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)*, pages 371–382. ACM, 2008.

[13] Frank Pfenning and Carsten 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 LNAI 1632, 1999.

[14] Adam Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *Proceedings of the 17th European Symposium on Programming (ESOP '08)*, March 2008.

[15] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 2000. CMU-CS-00-146.

[16] Carsten Schürmann and Frank 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'03)*, volume 2758 of *Lecture Notes in Computer Science*, pages 120–135, Rome, Italy, 2003. Springer.

[17] Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The ∇-calculus. Functional programming with higher-order encodings. In Pawel Urzyczyn, editor, *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA'05)*, volume 3461 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.

[18] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002.