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
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 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 
\[
\text{forall } \lambda x. (\text{eq } x \text{ (Suc Zero)) imp (not (eq x Zero))}
\]
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 [HHP93] and its implementation in Twelf [PS99]. Recently, HOAS encodings are supported in functional programming languages such as Elphin [SPS05], Delphin [PS08], and Beluga [Pie08]. In these systems, we analyze higher-order data using pattern matching and case expressions. This requires us to validate that the patterns are exhaustive. Similarly, proof assistants for HOAS-based reasoning that split a goal...
Dunfield and Pientka

Fig. 1. Natural deduction and its HOAS encoding

Numbers $N, M ::=$ $x$
$| 0$
$| \text{succ } N$

Propositions $A ::= N = M$
$| A \supset B$
$| \forall x . A$

Natural Deduction

$$
\Gamma \vdash \text{nd } A
\quad
\frac{\Gamma, u : \text{nd } A \vdash \text{nd } B}{\Gamma \vdash \text{nd } A \supset B}
\quad
\frac{\Gamma \vdash \text{nd } A \supset B \quad \Gamma \vdash \text{nd } A}{\Gamma \vdash \text{nd } B}
\quad
\frac{\Gamma \vdash \text{nd } A \supset B \quad \Gamma \vdash \text{nd } A}{\Gamma \vdash \text{nd } B}
$$

$$
\frac{\text{nd} : \text{type} .}{\text{impi} : (\text{nd } A \rightarrow \text{nd } B) \quad \frac{\Gamma \vdash \text{nd } A \rightarrow \text{nd } B}{\Gamma \vdash \text{nd } A \rightarrow \text{nd } B}}
$$

$$
\frac{\text{impe} : \text{nd } (A \supset B) \rightarrow \text{nd } A \rightarrow \text{nd } B .}{\text{alli} : (\Pi a : \text{nat} . \text{nd } (A a)) \rightarrow\frac{\Gamma \vdash \text{nd } (\forall x . A)}{\Gamma \vdash \text{nd } (\forall x . A)}}
$$

$$
\frac{(u : \text{nd } A) \in \Gamma}{\frac{\Gamma \vdash (\forall x . A)}{\Gamma \vdash \text{nd } [N/x] A}}
$$

$$
\frac{\text{Hyp} : \text{nd} : \text{type} .}{\text{alle} : \text{nd } (\forall x . A)}
\quad
\frac{\Gamma \vdash \text{nd } [N/x] A}{\Gamma \vdash \text{nd } (A N)}
$$

2

into different cases must ensure that the cases are exhaustive. This issue arises in Twelf’s induction theorem prover [Sch00], and in systems such as Bedwyr [BGM+07] and Abella [GMN08].

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 [HHP93]. 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 $\text{succ } N$, 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 [Coq92,SP03], or open terms within regular worlds [Sch00, 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 [Pie08] 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 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.
Motivation

To motivate the problem, we consider a simple program in the Beluga language [Pie08] that counts the free occurrences of some variable \( x \) in a formula. For example, \( \forall y. (x = y) \supset (\text{suc } y = \text{suc } x) \) 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 analyze 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 \( \text{cntV} \) will recursively analyze formulas. When it reaches a natural number expression, it will call a second function \( \text{cntVN} \). We use modal types such as \( o[\psi, x : \text{nat}] \), which describes a formula that can refer to the variables \( x \) and \( y \) of type \( \text{nat} \). We use single capital letters \( U, V, W \) for contextual variables, which are instantiated via higher-order pattern matching.

We examine the second function, \( \text{cntV} \), first. It is built by a context abstraction \( \Lambda \psi \Rightarrow \text{fn } f \Rightarrow \text{case } f \) of

\[
\begin{align*}
\text{box}(\psi, x. \text{eq } U[\psi, x] \text{ V}[\psi, x]) & \Rightarrow \text{cntV}[\psi] \text{ box}(\psi, x. \text{ U}[\psi, x]) \\
& \quad + \text{cntV}[\psi] \text{ box}(\psi, x. \text{ V}[\psi, x]) \\
\text{box}(\psi, x. \text{imp } U[\psi, x] \text{ V}[\psi, x]) & \Rightarrow \text{cntV}[\psi] \text{ box}(\psi, x. \text{ U}[\psi, x]) \\
& \quad + \text{cntV}[\psi] \text{ box}(\psi, x. \text{ V}[\psi, x]) \\
\text{box}(\psi, x. \text{forall}(\lambda y. \text{W}[\psi, x, y])) & \Rightarrow \text{cntV}[\psi] \text{ box}(\psi, y, x. \text{ W}[\psi, x, y])
\end{align*}
\]

Fig. 2. Counting free variables using pattern matching and HOAS
by a closure $U[\sigma]$ consisting of a contextual variable $U$ and a postponed substitution $\sigma$. As soon as we know what the contextual variable stands for, we apply the substitution $\sigma$. In the example, the postponed substitution associated with $U$ is the identity substitution which essentially corresponds to $\alpha$-renaming. We write $id_\psi$ for the identity substitution with domain $\psi$. 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 $U[id_\psi]$ the contextual variable $U$ can be instantiated with any formula that either is closed (does not refer to any bound variable in the context $\psi$) or contains a bound variable from $\psi$. Since subformulas can refer to all variables in $\psi,x:nat$, we write $U[id_\psi,x]$.

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

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

In the third case, for box($\psi,x,forall(\lambda y.W[id_\psi,x,y])$), we analyze the quantified formula under the assumption that $y$ is a natural number. To do this, we pass an extended context ($\psi,y:nat$) to $cntV$. The variable $x$ appears last in $box(\psi,y,x,...)$, to match the argument type $o[...\ ,x:nat]$.

The function cntVN counts the occurrences of a variable $x$ in an object of type $nat[\psi,x:nat]$, considering four cases. The first case, box($\psi,x$), matches an occurrence of $x$. The second case, box($\psi,x,p[id_\psi]$), matches a variable that is not $x$ and occurs in $\psi$. 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 $id_\psi$ 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 cntVN we ensure that the set of patterns $\{x,p[id_\psi],Zero,Suc U[id_\psi,x]\}$ covers the type $nat[\psi,x:nat]$. In cntV, the set $\{eq U[id_\psi,x] V[id_\psi,x], imp U[id_\psi,x] V[id_\psi,x], forall(\lambda y.U[id_\psi,x,y])\}$ covers all elements of type $o[\psi,x:nat]$.

This set of patterns for covering the type $o[\psi,x:nat]$ is by no means the only one. Instead of explicitly counting the occurrences of $x$ in a natural number of type $nat[\psi,x:nat]$, we could have used higher-order pattern matching to enforce variable dependencies, refining the pattern $eq U[id_\psi,x] V[id_\psi,x]$ into the four cases

\[
\{eq U[id_\psi] V[id_\psi], eq U[id_\psi,x] V[id_\psi], eq U[id_\psi] V[id_\psi,x], eq U[id_\psi,x] V[id_\psi,x]\}
\]

exactly distinguishing (1) $x$ occurs in neither $U[id_\psi]$ nor $V[id_\psi]$, (2) $x$ occurs in $U[id_\psi,x]$ but not in $V[id_\psi]$, (3) $x$ occurs in $V[id_\psi,x]$ but not in $U[id_\psi]$, and (4) $x$ 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 emphasize that
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 [Pie08].

We support the logical framework LF plus dependent pairs \( \Sigma \). Our data layer closely follows contextual modal type theory [NPP08], extended with parameter variables and context variables [Pie08], and finally with \( \Sigma \) types. Perhaps most importantly, we formalize schemas, which classify contexts. We only characterize normal terms since only these are meaningful in the logical framework [WCPW02,NPP08]. This is achieved by a syntactic distinction between normal terms \( M \) and neutral terms \( R \). The syntax guarantees that terms contain no \( \beta \)-redexes, and the typing rules guarantee that all well-typed terms are fully \( \eta \)-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 \( \lambda \)-abstraction. These variables are declared in a context \( \Psi \). Contextual variables stand for open objects, and include meta-variables \( u \) and \( v \), which represent general open objects, and parameter variables \( p \) that can only be instantiated with an ordinary bound variable. Contextual variables are introduced in computation-level case expressions, and can be instantiated via pattern matching. They are associated with a postponed substitution \( \sigma \). The intent is to apply \( \sigma \) as soon as we know the object the contextual variable should stand for. The domain of \( \sigma \) thus includes the free variables of that object, and the type system statically guarantees this. Contextual variables are declared in a meta-level context \( \Delta \).

Our foundation supports context variables \( \psi \) which allow us to reason abstractly with contexts, and write recursive computations that manipulate higher-order data. Unlike some other uses of context variables [MS04], 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 \( \Psi \). Context variables’ schemas

\(^1\) Prior work also considered substitution variables, which we omit here for brevity.

\(^2\) Lifting this restriction would require tracking dependencies of context variables on each other: in \( \psi, x:A, \psi' \), the context substituted for \( \psi' \) could depend on \( x \) or even on variables in \( \psi \). Ensuring that \( \alpha \)-renaming holds in the presence of multiple context variables and dependent types appears difficult.
are given in a schema context $\Omega$. We define schemas in Section 3.2.

Substitutions $\sigma$ 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 $\sigma$. We also have a first-class notion of identity substitution $\text{id}_\psi$. 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 $\text{proj}_k^R$ for the $k$th projection of $R$. For example, the second element of a triple is $\text{proj}_2^R = \text{proj}_1(\text{proj}_2^R)$.

### 3.1 Data-level typing

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

#### Data-level normal terms

\[
\begin{array}{c}
\Delta; \Psi \vdash x : A \vdash M \iff B \\
\Delta; \Psi \vdash \lambda x . M \equiv \Pi x : A . B \\
\Delta; \Psi \vdash M_1 \iff A_1 \quad \Delta; \Psi \vdash M_2 \iff \left[ M_1/x \right]_A A_2 \\
\Sigma \Pi
\end{array}
\]

\[
\Delta; \Psi \vdash (M_1, M_2) \iff \Sigma x : A_1 . A_2 \\
\Delta; \Psi \vdash R \Rightarrow P' \\
P' = P \quad \text{turn}
\]

#### Data-level neutral terms

\[
\begin{array}{c}
x : A \in \Psi \\
\Delta; \Psi \vdash x \Rightarrow A \\
c : A \in \Sigma \\
\Delta; \Psi \vdash c \Rightarrow A \\
u : A[\Phi] \in \Delta \\
\Delta; \Psi \vdash u[\sigma] \Rightarrow [\sigma]_\Phi A \\
\Delta; \Psi \vdash p[\sigma] \Rightarrow [\sigma]_\Phi A \\
\Delta; \Psi \vdash \Pi x : A . B \\
\Delta; \Psi \vdash N \Rightarrow A \\
\Sigma \Pi \quad \text{IE}
\end{array}
\]

\[
\begin{array}{c}
\Delta; \Psi \vdash R \Rightarrow \Sigma x : A_1 . A_2 \\
\Delta; \Psi \vdash \text{proj}_1 R \Rightarrow A_1 \\
\Sigma \text{EI}_1
\end{array}
\]

\[
\begin{array}{c}
\Delta; \Psi \vdash R \Rightarrow \Sigma x : A_1 . A_2 \\
\Delta; \Psi \vdash \text{proj}_2 R \Rightarrow [\text{proj}_1 R/x]_A A_2 \\
\Sigma \text{EI}_2
\end{array}
\]

#### Data-level substitutions

\[
\begin{array}{c}
\Delta; \Psi \vdash \sigma \iff \Phi \\
\Delta; \Psi \vdash \sigma \Rightarrow \Phi \\
\Delta; \Psi \vdash \sigma \iff \Phi \\
\Delta; \Psi \vdash \sigma \Rightarrow \Phi \\
\Delta; \Psi \vdash \sigma \iff \Phi \\
\Delta; \Psi \vdash (\sigma; R) \iff (\Phi, x : A) \\
\Delta; \Psi \vdash (\sigma; M) \iff (\Phi, x : A)
\end{array}
\]

Fig. 4. Data-level typing and substitutions

We give the typing rules for data-level terms in Figure 4. We assume that data-level type constants $\alpha$ 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 $\sigma$ are defined only on ordinary variables $x$, not on modal variables $u$. We also require the usual conditions on bound variables. For example, in $\Pi$ the bound variable $x$ must be new and cannot already occur in $\Psi$. This can always be achieved via $\alpha$-renaming.
Element types $\tilde{A} ::= \Pi x:A. \tilde{A} | a \, N_1 \ldots N_n$

Schema elements $F ::= \text{all } x_1: \tilde{B}_1, \ldots, x_k: \tilde{B}_k, \Sigma y_1: \tilde{A}_1, \ldots, y_j: \tilde{A}_j. \tilde{A}$

Schemas $W ::= (F_1 + \cdots + F_n)^*$

Context $\Psi$ checks against schema $W$

\[
\begin{array}{c}
\text{for some } k \\
\hline
\Omega; \Delta \vdash \Psi \iff W \\
\Omega; \Delta; \Psi \vdash F \in F_k \\
\Omega; \Delta \vdash \Psi \iff (F_1 + \cdots + F_n)^*
\end{array}
\]

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

\[
\begin{array}{c}
\tilde{\Theta} = x_1: \tilde{C}_1, \ldots, x_n: \tilde{C}_n \\
\sigma = u_1[\text{id}(\Psi)]/x_1, \ldots, u_n[\text{id}(\Psi)]/x_n \\
\Omega; \Delta, u_1:: \tilde{C}_1[\Psi], \ldots, u_n:: \tilde{C}_n[\Psi]; \Psi \vdash A \equiv [\sigma]\Sigma \tilde{\Phi}. \tilde{B} / (\theta, \Delta)
\end{array}
\]

\[
\Omega; \Delta; \Psi \vdash A \in \text{all } \tilde{\Theta}. \Sigma \tilde{\Phi}. \tilde{B}
\]

Fig. 5. Schemas

The typing rules for neutral terms use hereditary substitutions $[\cdot \cdot \cdot]_A$ which preserve canonical forms [NPP08]. 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 the 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.

### 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 $(\text{nat})^*$ represented contexts of the form $x_1: \text{nat}, \ldots, x_n: \text{nat}$. But we allow much more expressive contexts. For instance, when reasoning about natural deductions, the rule $\Gamma^u$ adds an assumption of the form $u: (\text{nd } A)$ for some concrete proposition $A$. The inductive definition $\Gamma' ::= \cdot | \Gamma', x: \text{nat}, | \Gamma', u: (\text{nd } A)$ corresponds to the schema $(\text{nat} + (\text{all } A: \text{nd} A))^*$.

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

We give the grammar of schemas in Figure 5. Schemas are built of elements $F_1, \ldots, F_n$, each of the form $\text{all } \tilde{\Theta}. \Sigma y_1: \tilde{B}_1, \ldots, y_j: \tilde{A}_j. \tilde{A}$, where $\tilde{\Theta} = x_1: \tilde{C}_1, \ldots, x_k: \tilde{C}_k$. In other words, for any instantiation of $\tilde{\Theta}$ (that is, any substitution for $x_1, \ldots, x_k$), the element is of $\Sigma \Pi$-type, where we first introduce some $\Sigma$s, followed by $\Pi$s, with no subsequent $\Sigma$s. This restriction makes it easier to describe the inhabitants of the type. Twelf has a similar restriction on worlds. In Beluga, computation typing [Pic08] guarantees that contexts matching this grammar are the only contexts created during computation.
To check a context $\Psi$ against a schema $(F_1 + \cdots + F_n)$, we check that each element $x:A$ in $\Psi$ is an instance of a schema element $F_k = \text{all } \Theta. \Sigma y_1: B_1, \ldots, y_j: B_j. \overline{B}$, with all variables in $\Theta$ instantiated such that $x:A$ is an instance of $F_k$. The rule in Figure 5 uses higher-order pattern matching. The judgment $A = B / (\theta, \Delta)$ means that $\theta$ 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 [McB00, 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 [Coq92] and Schürmann and Pfenning [SP03] described coverage checking for closed terms, while Schürmann [Sch00, 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 $Z = \{\text{Zero, Suc } u\}$ covers all (closed) terms of type $\text{nat}[: ]$, we need to split $\text{nat}[: ]$ into the pattern set $Z' = \{\text{Zero, Suc } u_1\}$. Now it is obvious that $Z$ covers $\text{nat}[: ]$, because $Z'$—the result of splitting $\text{nat}[: ]$—is $\alpha$-equivalent to $Z$.

More generally, suppose we want to check that $Z$ covers $\text{nat}[\Psi]$. 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 $\text{nat}[\psi, x:\text{nat}, y:\text{o}]$, where $\psi$ represents a context of schema $(\text{o + nat})^*$. The split then includes the constructors, parameter variables denoting the generic case for variables from $\psi$ (one variable for each schema element), and the concrete variables $x$ and $y$:

\[
\text{constructors of nat} \quad \text{variables of } \psi \quad x:\text{nat}, y:\text{o} \\
\sim \text{Zero, Suc } u[id_\psi, x, y], \quad \overline{(p_1[id_\psi] : o), (p_2[id_\psi] : \text{nat})}, \quad \overline{x, y}
\]

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

For some sets $Z$ we would also need to split Suc’s argument $u[id_\psi, x, y]$ 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 $\Delta$. The judgment $\Omega; \Delta' \vdash \theta \iff \Delta$ says that $\theta$ is a contextual substitution with domain $\Delta$ and range $\Delta'$, under the schema context $\Omega$. We write $\rho$ as an abbreviation for (1) a context substitution on the schema context $\Omega$, substituting for context variables $\psi$, and (2) a contextual substitution $\theta$. The judgment $\Omega; \Delta' \vdash \rho : (\Omega; \Delta)$ says that the domain of $\rho$ 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 the appendix for details.

We allow higher-order patterns in the sense of Miller [Mil91], 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 [Pie03]. The proof of the following is a simple extension of the one in [Pie03].

**Theorem 4.1 (Soundness of higher-order pattern unification)**  
If $P$ and $Q$ are well-formed types under $\Omega; \Delta; \Psi$, and $\Omega; \Delta; \Psi \vdash Q \vdash P \vdash (\theta, \Delta')$, then $\Omega; \Delta' \vdash \theta : \Delta$ and $\Omega; \Delta'; [\theta]P = [\theta]Q$ and $\theta$ is the most general unifier, that is, for all $\vdash \rho : (\Omega; \Delta)$ there exists $\rho'$ such that $\rho = [\rho']\theta$.

**Lemma 4.2 (Object inversion)**  
If $\vdash \vdash \Psi \vdash R \iff P$ and $\vdash \Psi : W$ then either  
(1) $R = c \, N_1 \ldots N_k$ where $S(c) = \Pi x_1: A_1, \ldots, \Pi x_k: A_k.P'$ and $[\sigma]P' = P$, or  
(2) $R = x \, N_1 \ldots N_k$ where $(x : \Pi x_1: A_1, \ldots, \Pi x_k: A_k.P') \in \Psi$ and $[\sigma]P' = P$, or  
(3) $R = (\text{proj}_# y) \, N_1 \ldots N_k$ where $(y : \Sigma y_1: A_1, \ldots, y_m: A_m, A_{m+1}) \in \Psi$  
and $[\sigma]P' = P$ and $[\text{proj}_# y/y_1, \ldots, \text{proj}_# y/y_l]A_{l+1} = \Pi x_1: B_1, \ldots, \Pi x_k: B_k.P'$  
where $1 \leq l \leq m$.

**Proof.** By case analysis and inversion on the derivation of $\vdash \vdash \Psi \vdash R \iff P$. \hfill $\square$

### 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 $\Pi \Delta', \text{box}(\Psi, M) : A[\Psi']$, where $\Delta'$ 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 $\text{Suc} \, u[\text{id}_\psi, x]$, but $u : \text{nat} \, \psi, x : \text{nat} \, \text{box}(\psi, x, \text{Suc} \, u[\text{id}_\psi, x]) : \text{nat} \, \psi, x : \text{nat}$. In this example, and in many situations, $\Delta'$ and $A[\Psi']$ could be omitted in the source program and reconstructed. However, a dependently-typed $\Delta'$ such as $u : (\text{nd} (\text{eq} x x))[x : \text{nat}]$ actually restricts $u$ to match only natural-deduction proofs of $\text{eq} x x$. Similarly, a dependently-typed $A$ can constrain the entire pattern.

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

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


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

\[
\begin{array}{c}
\frac{M_1 \vdash A_1 \triangleright J \quad \vdots \quad M_n \vdash A_n \triangleright J}{\text{Obj}(\text{nat}[\cdot]) \triangleright J}
\end{array}
\]

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 \( \text{Obj}(A) \triangleright \text{covered-by} \ Z \) means to analyze \( A \) and “continue with” \( \text{covered-by} \ Z \). So, having analyzed \( A \), splitting as necessary, we eventually come to subderivations of \( M_k : A_k \triangleright \text{covered-by} \ Z \). 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: \( \text{covered-by}-Z \) picks out one pattern \( \zeta \) from the set \( Z \), and then \( \text{covered-by}-\zeta \) checks that \( M_k \) is an instance of \( \zeta \).\(^3\)

\[
\begin{array}{l}
\Omega \vdash (\Pi \Delta. \text{box}(\Psi, M_k) : A_k[\Psi]) \equiv \zeta \quad / \quad (\theta, \Delta)
\end{array}
\]

\[
\begin{array}{l}
\Omega \vdash \Pi \Delta. \text{box}(\Psi, M_k) : A_k[\Psi] \quad \text{covered-by-\zeta}
\end{array}
\]

\[
\begin{array}{l}
\Omega; \Delta; \Psi \vdash M_k : A_k \triangleright \text{covered-by} \{\ldots, \zeta, \ldots\}
\end{array}
\]

We assume that the pattern \( \zeta \) includes an explicit meta-variable context \( \Delta' \), explicit data-level names \( \Psi' \), and an explicit type \( A'[\Psi'] \). Thus, the premise of \( \text{covered-by-\zeta} \) is \( \Omega \vdash (\Pi \Delta. \text{box}(\Psi, M_k) : A_k[\Psi]) \equiv (\Pi \Delta'. \text{box}(\Psi', M') : A'[\Psi']) / (\theta, \Delta) \). This says that \( M_k \) is an instance of \( M' \) realized by \( \theta \), 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 \( \text{Obj}(A) \triangleright J \)

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 \( \text{Obj}(A) \triangleright J \). These are the four rules at the bottom of Figure 6.

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

\[
\begin{array}{l}
\Omega; \Delta; \Psi \vdash (\lambda x. M) : (\Pi x: A_1.A_2') \triangleright J
\end{array}
\]

\[
\begin{array}{l}
\Omega; \Delta; \Psi \vdash M : A_2' \triangleright \text{LAM} \triangleright J
\end{array}
\]

\[
\begin{array}{l}
\vdots
\end{array}
\]

\[
\begin{array}{l}
\Omega; \Delta; \Psi \vdash \text{Obj}(A_2) \triangleright \text{LAM} \triangleright J
\end{array}
\]

\[
\begin{array}{l}
\Omega; \Delta; \Psi \vdash \text{Obj}(\Pi x: A_1.A_2) \triangleright J
\end{array}
\]

\( \text{Obj-\Pi} \)

\(^3\) Note that we need matching, not just equality, in \( \text{covered-by-\zeta} \). Suppose \( Z = \{(\text{zero}, \text{nat}[\cdot]), (\text{zero}, u_2[\cdot])\} \). To show that \( (\text{zero}, \text{nat}[\cdot]) \) is covered (by the second pattern in \( Z \)), we need to split the first component, and to show that \( (\text{nat} u_1[\cdot], \text{zero}) \) is covered (by the first pattern in \( Z \)), we need to split the second component. This results in a set of patterns including \( (\text{zero}, \text{zero}) \), which is not equal to any pattern in \( Z \).
\[
\Omega \vdash \Pi \Delta. \text{box}(\bar{\Psi}, M) : A[\Psi] \text{ covered-by } \zeta
\]

\[
\Omega \vdash (\Pi \Delta. \text{box}(\bar{\Psi}, M) : A[\Psi]) \equiv (\Pi \Delta'. \text{box}(\bar{\Psi'}, M') : A'[\Psi']) \quad / \quad (\theta, \Delta')
\]

Covered-By-\zeta

\[
\Omega; \Delta; \Psi \vdash \text{App}(R)(A > P) \triangleright \mathcal{J}
\]

\[
\frac{\Omega; \Delta; \Psi \vdash Q \neq P}{\Omega; \Delta; \Psi \vdash \text{App}(R)(Q > P) \triangleright \mathcal{J}} \quad \text{App-\neq}
\]

\[
\frac{\Omega; \Delta; \Psi \vdash Q \equiv P / (\theta, \Delta')}{\Omega; \Delta; \Psi \vdash \text{App}(R)(Q > P) \triangleright \mathcal{J}} \quad \text{App-\equiv}
\]

\[
\frac{\Omega; \Delta; \Psi \vdash \text{App}(R)(M/\bar{M}B > P) \triangleright \mathcal{J}}{\Omega; \Delta; \Psi \vdash \text{neutral}(R)(x.B > P) \triangleright \mathcal{J}} \quad \text{App-\Pi}
\]

for \(0 \leq i \leq m:
\]

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

\[
\Omega; \Delta; \Psi \vdash M : A \triangleright \mathcal{J}
\]

\[
\Omega \vdash \Pi \Delta. \text{box}(\bar{\Psi}, M) : A[\Psi] \text{ covered-by } \zeta_k \quad \text{Covered-By-}\zeta
\]

\[
\Omega; \Delta; \Psi \vdash (\lambda x. M) \vdash \text{lam} \triangleright \mathcal{J}
\]

\[
\Omega; \Delta; \Psi \vdash \text{Obj}(A) \triangleright \mathcal{J}
\]

\[
\Omega; \Delta; \Psi \vdash \text{Obj}(\Pi x_1: A_1, A_2) \triangleright \mathcal{J} \quad \text{Obj-\Pi}
\]

\[
\Omega; \Delta; \Psi \vdash \text{Obj}(\Sigma x_1: \bar{A}_1, A_2) \triangleright \mathcal{J} \quad \text{Obj-\Sigma}
\]

\[
\Omega; \Delta; \Psi \vdash \text{MVars}(P) \triangleright \mathcal{J} \quad \text{Obj-no-split}
\]

\[
\Psi = \psi_1: \Sigma \bar{\Psi}_1, \bar{A}_1, \ldots, \psi_k: \Sigma \bar{\Psi}_k, \bar{A}_k
\]

\[
\Omega(\psi) = F_1 + \cdots + F_m
\]

\[
\Omega; \Delta; \Psi \vdash \text{PVars}(\psi : F_1) \triangleright \mathcal{J}
\]

\[
\Omega; \Delta; \Psi \vdash \text{App}(c_1)(\Sigma \bar{\Psi}_k, \bar{A}_k > P) \triangleright \mathcal{J}
\]

\[
\Omega; \Delta; \Psi \vdash \text{App}(c_n)(\Sigma \bar{\Psi}_k, \bar{A}_k > P) \triangleright \mathcal{J}
\]

\[
\Omega; \Delta; \Psi \vdash \text{Obj}(P) \triangleright \mathcal{J} \quad \text{Obj-split}
\]

Fig. 6. Coverage checking rules
Note that since splitting $A_2$ may produce several patterns, we may have more sub-derivations $(\lambda x. \ldots : (\Pi x:A_1, \ldots))$ than just the one shown.

If $A = \Sigma x:A_1, A_2$, rule Obj-$\Sigma$ 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 $\text{App}(c_k)(S(c_k) > P)$ 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 $\Omega; \Delta; \Psi \vdash \text{App}(R)(S(c_k) > P) \triangleright J$ 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 $S(c_k)$ must have the form $\Pi x_1:A_1, \ldots \Pi x_m:A_m.Q$, where $Q$ is a base type. In deriving this, we use $\text{App}-\Pi$, which uses 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 $\text{App}(c_k N_1 \ldots N_m)(Q > P)$, for various spines $N_1 \ldots N_m$. If $Q$ and $P$ do not unify (written $Q \not\equiv P$ in rule App-$\not\equiv$) we have a trivial coverage subderivation, but if $Q$ and $P$ do unify under some $\theta$, then we can use $\text{App}-\triangleright$, which has a premise $[\theta]R: [\theta]P \triangleright [\theta]J$.

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

The remaining premises of Obj-split have the form $\text{PVars}(\psi : F) > P \triangleright J$, characterizing the generic variable cases.

4.4 $\text{PVars}(\psi : F) > P \triangleright J$: Parameter variables

Exactly one rule concludes $\text{PVars}(\ldots)$, the rule $\text{PVars}$ in Figure 7. In $\text{PVars}$, 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 $F = \text{all }A:o.n.d.A$, then $p[\text{id}_\psi]$ has type $\text{id} u[\text{id}_\psi]$ where $u$ is a (fresh) meta-variable. In general, we get the type of a parameter from the element $\text{all }\Theta.\Sigma\Phi. A$ by generating a substitution $\sigma'$ that instantiates all variables in $\Theta$ with meta-variables, and applying $\sigma'$ to $\Sigma\Phi. A$. Then we use the ideas for concrete variables. Again, since $[\sigma']\Sigma\Phi. A$ is inhabited by tuples, we consider all possible projections.

4.5 $\text{MVars}(P) \triangleright J$: General case for all ground instances of $P$

The premise of rule Obj-no-split is $\text{MVars}(P)$, which is derivable only by rule MVars (Figure 7). This rule does not recursively analyze the given type $P$. Instead, it produces patterns $u[\text{id}(\Psi_k)]$, which any object of type $P[\Psi_k]$ matches. \(^4\)

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

\(^4\) The operation $\text{id}(\Psi)$ unrolls $\Psi$. For example, $\text{id}(\psi, x:\text{nat}) = \text{id}_\psi, x$. See the appendix for details.
\[ \Omega; \Delta; \Psi \vdash \text{PVars}(\psi : \text{all } \tilde{\Theta}, \Sigma \tilde{\Phi}, \tilde{A}_{j+1}) > P \triangleright J \]

\[ \tilde{\Theta} = y_1; \tilde{B}_1, \ldots, y_n; \tilde{B}_n \quad \text{and} \quad \tilde{\Phi} = x_1; \tilde{A}_1, \ldots, x_j; \tilde{A}_j \]

\[ \sigma = u_1[\text{id}_\psi/y_1, \ldots, u_n[\text{id}_\psi]/y_n \quad \Delta_\Theta = u_1::\tilde{B}_1[\psi], \ldots, u_n::\tilde{B}_n[\psi] \]

for \(0 \leq i \leq j:\)

\[ \sigma' = (\text{proj}^# p[\text{id}_\psi]/x_1, \ldots, (\text{proj}^# p[\text{id}_\psi])/x_i \]

\[ \Omega; \Delta, \Delta_\Theta, \rho::[\sigma](\Sigma\tilde{\Phi}, \tilde{A}_{j+1}); \Psi \vdash \text{App}(\text{proj}^# p[\text{id}_\psi])((\sigma'[\sigma])\tilde{A}_{j+1} > P) \triangleright J \]

\[ \Omega; \Delta; \Psi \vdash \text{PVars}(\psi : \text{all } \tilde{\Theta}, \Sigma \tilde{\Phi}, \tilde{A}_{j+1}) > P \triangleright J \]

\[ \text{ValidWk}(\Omega; \Delta \vdash P[\Psi]) = \{ \Psi_1, \ldots, \Psi_n \} \]

\[ \Omega; \Delta, \rho::P[\Psi_n]; \Psi \vdash (u[\text{id}(\Psi_n)]) : P \triangleright J \]

\[ \Omega; \Delta; \Psi \vdash \text{MVars}(P) \triangleright J \]

\[ \text{MVars}(P) \]

Fig. 7. Coverage checking rules (continued)

In fact, we generate all valid weakenings of \( \Psi \). A weakening \( \Psi' \subseteq \Psi \) has zero or more assumptions from \( \Psi \) (preserving order). These contexts are weaker because they provide less information. Not all weakenings make sense; for example, removing \( \text{pronat} \) from \( (x:\text{nat}; y;(\text{eq } x)) \) yields \( (y;(\text{eq } x)) \), which is dependent on an undeclared \( x \). The valid weakenings \( \text{ValidWk}(\Omega; \Delta \vdash A[\Psi]) \) of a context \( \Psi \) 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 \( \vdash \Psi : \text{Obj}(A) \triangleright \text{COVERED-BY-Z} \), 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, which will guarantee that if \( D \) derives \( \text{Obj}(A) \triangleright J \) then, for every ground \( M' \) of type \( A \), there is within \( D \) a derivation of \( M_i : A \triangleright J \), where \( M_i \) 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 \( J = \text{COVERED-BY-Z} \), the lemma gives a subderivation \( D' \) of \( \ldots \vdash M : A \triangleright \text{COVERED-BY-Z} \), and inversions bring us to the premise of \( \text{Covered-By-Z} \).

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

Thus, the antecedent that \( M' \) has type \( A \) can be ground: \( \vdash \Theta[\rho]\Psi \vdash M' \iff [\rho]A \), where \( \Omega \) and \( \Delta \) are grounded by \( \vdash \rho : (\Omega; \Delta) \). In addition, the domain of \( D' \) need not exactly match the domain of \( D \). In fact, the type in \( D' \) will be \( [\theta]\Psi \), where \( \theta \) is a substitution from \( \Delta \) to \( \Delta' \). This is consistent with the intuition that types become more precise as we move into subderivations.
As we have \( \theta \) from \( \Delta \) to \( \Delta' \), and \( \rho \) from \( (\Omega; \Delta) \) to ground \( (\cdot; \cdot) \), the lemma also asserts the existence of a \( \rho' \) from \( (\Omega; \Delta') \) to ground, so that \( \rho = [\rho'][\theta]. \)

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

**Lemma 4.3 (Coverage Soundness)**

1. If \( D :: \Omega; \Delta, \Psi \vdash \text{Obj}(A) \triangleright J \) and \( \cdot \vdash [\rho] \Psi \vdash M' \Leftarrow [\rho] A \) and \( \cdot \vdash : (\Omega; \Delta) \) then there exist \( \theta, \rho \) such that \( \Omega; \Delta', \Psi \vdash \theta \Leftarrow \Delta \) and \( \rho' \) s.t. \( \rho = [\rho'][\theta] \theta \) and \( M' = [\rho'] M \).

2. If \( D :: \Omega; \Delta, \Psi \vdash \text{App}(R)(\tilde{A} > P) \triangleright J \) and \( \Omega; \Delta; \Psi \vdash R \Rightarrow \tilde{A} \) and \( \cdot \vdash : (\Omega; \Delta) \) and for all spines \( N_1', \ldots, N_n' \) of some length \( n \) such that \( \cdot \vdash [\rho] \Psi \vdash ([\rho] R) N_1' \ldots N_n' \Leftarrow [\rho] P \), then \( \rho' \) s.t. \( \rho = [\rho'][\theta] \theta \) and there exists \( \theta' \) s.t. \( \rho = [\rho'][\theta'] \).

**Proof.** By complete induction on the height of \( D \).

**Theorem 4.4 (Coverage Soundness)**

If \( \cdot \vdash : \Psi \vdash M' \Leftarrow A \) and \( \cdot \vdash : \Psi \vdash \text{Obj}(A) \triangleright \text{COVERED-BY} \ Z \) then there exists \( \zeta \in Z \) such that \( \zeta = (\Pi \Delta', \text{box}(\Psi, M_k) : A_k[\Psi_k]) \) and \( \cdot \vdash : (\Pi \Delta'. \text{box}(\Psi, M) : A[\Psi]) \triangleright \zeta / (\theta_k, \Delta') \) 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 [Coq92] in the setting of Agda and later refinements of this approach [McB00,Nor07], dealt with closed data objects. In the setting of logical frameworks, theoretical work on coverage also concentrated on closed objects [SP03]. 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 \( \psi \). 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 [PS08] as well.
We plan to implement a coverage algorithm based on the ideas in this paper within the Beluga prototype.

References


A Appendix: Substitutions

A.1 Ordinary substitution

In the definition of ordinary data-level substitutions, we need to be careful because the only meaningful data-level terms are those in canonical form. To ensure that substitution preserves canonical form, we use a technique pioneered by Watkins et al. [WCPW02] and described in detail in [NPP08]. The idea is to define hereditary substitution as a primitive recursive functional that always returns a canonical object.

In the formal development, it is simpler if we can stick to non-dependent types. We therefore first define type approximations $\alpha$ and an erasure operation $(\cdot)^-$ that removes dependencies. Before applying any hereditary substitution $[M/x]^\alpha_A(B)$ we first erase dependencies to obtain $\alpha = A^-$ and then carry out the hereditary substitution formally as $[M/x]^\alpha_A(B)$. A similar convention applies to the other forms of hereditary substitutions.

Type approximations $\alpha, \beta ::= a | \alpha \rightarrow \beta | \alpha \times \beta$

Types relate to type approximations via an erasure operation $(\cdot)^-$ which we overload to work on types.

\[
(a N_1 \ldots N_n)^- = a \\
(\Pi x : A \ldots B)^- = A^- \rightarrow B^- \\
(\Sigma x : A \ldots B)^- = A^- \times B^-
\]

We can define $[M/x]^\alpha\,^n_a(N)$, $[M/x]^\alpha\,^r_a(R)$, and $[M/x]^\alpha\,^s_{\bar{\psi}}(\sigma)$ by nested induction, first on the structure of the type approximation $\alpha$ and second on the structure of the objects $N$, $R$ and $\sigma$. In other words, we either go to a smaller type approximation (in which case the objects can become larger), or the type approximation remains the same and the objects become smaller. We define the hereditary substitution operations in Figure A.1. We write $\alpha \leq \beta$ and $\alpha < \beta$ if $\alpha$ occurs in $\beta$ (as a proper subexpression in the latter case). If the original term is not well-typed, a hereditary substitution, though terminating, cannot always return a meaningful term. We formalize this as failure to return a result. However, on well-typed terms, hereditary substitution will always return well-typed terms. This substitution operation can be extended to types for which we write $[M/x]^\alpha\,^n_a(A)$.

Theorem A.1 (Termination)

$[M/x]^\alpha\,^n_a(\_)$ where $\_ = \{n, r, s, a\}$ terminates, either by returning a result or failing after a finite number of steps.

Theorem A.2 (Hereditary Substitution Principles)

If $\Delta; \Psi \vdash M \leftarrow A$ and $\Delta; \Psi, x : A, \Psi' \vdash J$
then $\Delta; \Psi, [M/x]^\alpha\,^*_{\bar{\psi}} \Psi' \vdash [M/x]^\alpha\,^*_{\bar{\psi}}(J)$ for all $* \in \{n, r, s, a\}$.

Building on the ideas in [NPP08], we can also define simultaneous substitution $[\sigma]^\alpha_{\bar{\psi}}(M)$ (respectively $[\sigma]^r_{\bar{\psi}}(R)$ and $[\sigma]^s_{\bar{\psi}}(\sigma)$). We write $\bar{\psi}$ for the context approxi-
The different variables (ordinary variables $x$, context variables $\psi$, and contextual variables $u[\sigma]$ and $p[\sigma]$) give rise to different substitution operations. The re-

\[ [M/x]_{\alpha}^n(N) = N' \]
\[ [M/x]_{\alpha}^n(R) = R' \text{ or } M' : \alpha' \]
\[ [M/x]_{\alpha}^n(\sigma) = \sigma' \]

Normalized terms $N$
Neutral terms $R$
Substitutions $\sigma$

Data-level substitutions
\[ [M/x]_{\alpha}^n(\lambda y. N) = \lambda y. N' \] where $N' = [M/x]_{\alpha}^n(N)$
choosing $y \not\in \text{FV}(M)$, and $y \neq x$
\[ [M/x]_{\alpha}^n(M_1, M_2) = (N_1, N_2) \] if $[M/x]_{\alpha}^n(M_1) = N_1$ and $[M/x]_{\alpha}^n(M_2) = N_2$
\[ [M/x]_{\alpha}^n(R) = M' \] if $[M/x]_{\alpha}^n(R) = M' : \alpha'$
\[ [M/x]_{\alpha}^n(R) = R' \] if $[M/x]_{\alpha}^n(R) = R'$
\[ [M/x]_{\alpha}^n(\sigma) = \sigma' \]

Data-level normal terms
\[ [M/x]_{\alpha}^n(y) = y \] if $y \neq x$
\[ [M/x]_{\alpha}^n(u[\sigma]) = u[\sigma'] \] where $\sigma' = [M/x]_{\alpha}^n(\sigma)$
\[ [M/x]_{\alpha}^n(p[\sigma]) = p[\sigma'] \] where $\sigma' = [M/x]_{\alpha}^n(\sigma)$
\[ [M/x]_{\alpha}^n(R N) = R' N' \] where $R' = [M/x]_{\alpha}^n(R)$ and $N' = [M/x]_{\alpha}^n(N)$
\[ [M/x]_{\alpha}^n(R N) = M'' : \beta \] if $[M/x]_{\alpha}^n(R) = \lambda y. M' : \alpha_1 \to \beta$
where $\alpha_1 \to \beta \leq \alpha$ and $N' = [M/x]_{\alpha}^n(N)$
and $M'' = [N'/y]_{\alpha_1}^n(M')$
\[ [M/x]_{\alpha}^n(\text{proj}_i R) = N_i : \alpha_i \] if $[M/x]_{\alpha}^n(R) = (N_1, N_2) : \alpha_1 \times \alpha_2$
\[ [M/x]_{\alpha}^n(\text{proj}_i R) = \text{proj}_i R' \] if $[M/x]_{\alpha}^n(R) = R'$
\[ [M/x]_{\alpha}^n(R) = \cdot \]

Substitutions
\[ [M/x]_{\alpha}^n(\cdot) = \cdot \]
\[ [M/x]_{\alpha}^n(\sigma ; N) = (\sigma' ; N') \] where $\sigma' = [M/x]_{\alpha}^n(\sigma)$ and $N' = [M/x]_{\alpha}^n(N)$
\[ [M/x]_{\alpha}^n(\sigma , R) = (\sigma' , R') \] if $[M/x]_{\alpha}^n(R) = R'$ and $\sigma' = [M/x]_{\alpha}^n(\sigma)$
\[ [M/x]_{\alpha}^n(\sigma , R) = (\sigma' ; M') \] if $[M/x]_{\alpha}^n(R) = M' : \alpha'$ and $\sigma' = [M/x]_{\alpha}^n(\sigma)$
\[ [M/x]_{\alpha}^n(\text{id}_{\psi}) = \text{id}_{\psi} \]
\[ [M/x]_{\alpha}^n(\psi) = \psi \]

\[ (\psi)^{-} = (\psi) \]
\[ (\cdot)^{-} = \cdot \]

A.2 Substitution operations

The different variables (ordinary variables $x$, context variables $\psi$, and contextual variables $u[\sigma]$ and $p[\sigma]$) give rise to different substitution operations. The re-

\[ (\Psi, x:A)^{-} = (\Psi)^{-}, x:(A)^{-} \]

Fig. A.1. Hereditary substitution (data-level)
maining substitution operations do not require any significant changes from earlier work [Pie08,NPP08] to handle dependent types, and we revisit them in this section.

**Substitution for context variables**

If we encounter a context variable $\psi$, we simply replace it with the context $\Psi$.

**Data-level context**

$$\begin{align*}
[\Psi/\psi](\cdot) & = \cdot \\
[\Psi/\psi](\Phi, x:A) & = (\Phi', x:A') \quad \text{if } x \notin \mathcal{V}(\Phi'), \quad [\Psi/\psi]A = A', \quad [\Psi/\psi]\Phi = \Phi' \\
[\Psi/\psi](\psi) & = \Psi \\
[\Psi/\psi](\phi) & = \phi \quad \text{if } \phi \neq \psi
\end{align*}$$

When we apply the substitution $[\Psi/\psi]$ to the context $\Phi, x:A$, we apply the substitution to the type $A$, yielding some new type $A'$, and to the context $\Phi$, yielding some new context $\Phi'$. Applying the substitution to the type $A$ is necessary in the dependently-typed setting, since $A$ may contain terms and in particular identity substitutions $id_\psi$. When we replace $\psi$ with $\Psi$ in the substitution $id_\psi$, we unfold the identity substitution. Expansion of the identity substitution is defined by the operation $id(\Psi)$ for valid contexts $\Psi$:

$$\begin{align*}
id(\cdot) & = \cdot \\
id(\Psi, x:A) & = id(\Psi), x \\
id(\psi) & = id_\psi
\end{align*}$$

**Lemma A.3 (Unfolding identity substitution)**

*If* $id(\Psi) = \sigma$ *then* $\Delta; \Psi, \Psi' \vdash \sigma \iff \Psi$.

**Proof.** By induction on the structure of $\Psi$. □

When we combine $\Phi'$ and the declaration $x:A'$ to yield a new context, we must ensure that $x$ is not already declared in $\Phi'$. This can always be achieved by appropriately renaming bound variable occurrences. We write $\mathcal{V}(\Phi')$ for the set of variables declared in $\Phi'$. The rest of the definition is mostly straightforward.

**Theorem A.4 (Substitution for context variables)**

*If* $\Omega, \psi :: W, \Omega'; \Delta; \Phi \vdash J$ *and* $\Omega \vdash \Psi \iff W$ *then* $\Omega, \Omega', [\Psi/\psi] \Delta; [\Psi/\psi](\Phi) \vdash [\Psi/\psi]J$.

**Proof.** By induction on the first derivation using Lemma A.3. □

**Contextual substitution for contextual variables**

Substitution for contextual variables is a little more difficult, but is essentially similar to the definitions in [Pie08]. We can think of $u[\sigma]$ as a closure where, as soon as we know which term $u$ should stand for, we can apply $\sigma$ to it. The typing will ensure that the type of $M$ and the type of $u$ agree, i.e. we can replace $u$ of type $A[\Psi]$ with a normal term $M$ if $M$ has type $A$ in the context $\Psi$. Because of $\alpha$-conversion, the variables substituted at different occurrences of $u$ may differ, and we write the contextual substitution as $[\Psi.M/u](N)$, $[\Psi.M/u](R)$, and $[\Psi.M/u](\sigma)$, where $\Psi$ binds all free variables in $M$. Applying $[\Psi.M/u]$ to the closure $u[\sigma]$ first obtains the
This typing annotation is necessary since we apply the substitution
and substitutions as follows.

We define the contextual substitution operations for normal object, neutral objects
\( A \).

\[\text{Simultaneous contextual substitution}\]

\[
\frac{\Delta \vdash \theta \Rightarrow \Delta' \quad \Delta; [\theta][M/u] A}{\Delta \vdash \theta \Rightarrow \Delta' \quad \Delta; [\theta][M/u] A} \]

\[
\Delta \vdash (\theta, [\hat{\Psi}.x/p]) \Rightarrow \Delta', u::A[\Psi]
\]

A.2.1 Contextual substitution for meta-variables

We define the contextual substitution operations for normal object, neutral objects
and substitutions as follows.

\[
[\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^n(N) = N' \quad \text{Normal terms } N
\]

\[
[\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^n(R) = R' \text{ or } M' : \alpha' \quad \text{Neutral terms } R
\]

\[
[\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^n(\sigma) = \sigma' \quad \text{Substitutions } \sigma
\]

As mentioned earlier, \( u[\sigma] \) represents a closure where, as soon as we know which
term \( u \) should stand for, we can apply \( \sigma \) to it. Because of \( \alpha \)-conversion, the variables
substituted at different occurrences of \( u \) may differ, and we write \( \hat{\Psi}.M \) to allow for
necessary \( \alpha \)-renaming. The contextual substitution is indexed with the type of \( u \).

This typing annotation is necessary since we apply the substitution \( \sigma \) hereditarily
once we know which term \( u \) represents, and hereditary substitution requires the
type to ensure termination.

Data-level normal terms

\[
[\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^n(\lambda y. N) = \lambda y. N' \quad \text{where } [\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^n(N) = N'
\]

\[
[\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^n(N_1, N_2) = (N'_1, N'_2)
\]

\[
\quad \text{where } [\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^n(N_1) = N'_1 \text{ and } [\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^n(N_2) = N'_2
\]

\[
[\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^n(R) = R' \quad \text{where } [\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^r(R) = R'
\]

\[
[\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^n(R) = M' \quad \text{where } [\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^r(R) = M' : \beta
\]

\[
[\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^n(N) = \text{fails otherwise}
\]

In the following, note that applying \( [\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^r \) to the closure \( u[\sigma] \) first obtains
the simultaneous substitution \( \sigma' = [\hat{\Psi}.M/u]_{\alpha[\hat{\psi}]}^s \sigma \), but instead of returning \( M[\sigma'] \),
it eagerly applies \( \sigma' \) to \( M \). However before that we recover its domain by \( [\sigma'/\bar{\psi}] \).

To enforce that we always return a normal object as a result of contextual substitution, we resort to ordinary hereditary substitution. For a thorough explanation, see [NPP08].

Data-level neutral terms
\[
\begin{align*}
\tilde{\Psi} & . M/u \stackrel{r}{\alpha}[\bar{\psi}] (x) = x \\
\tilde{\Psi} & . M/u \stackrel{r}{\alpha}[\bar{\psi}] (u[\sigma]) = N : \alpha \quad \text{where} \quad \tilde{\Psi} . M/u \stackrel{s}{\alpha}[\bar{\psi}] \sigma = \sigma' \quad \text{and} \quad [\sigma'/\bar{\psi}]^* \sigma M = N \\
\tilde{\Psi} & . M/u \stackrel{r}{\alpha}[\bar{\psi}] (u'[\sigma]) = u'[\sigma'] \quad \text{where} \quad \tilde{\Psi} . M/u \stackrel{s}{\alpha}[\bar{\psi}] \sigma = \sigma' \quad \text{choosing} \quad u' \neq u \\
\tilde{\Psi} & . M/u \stackrel{r}{\alpha}[\bar{\psi}] (p[\sigma]) = p[\sigma'] \quad \text{where} \quad \tilde{\Psi} . M/u \stackrel{s}{\alpha}[\psi] \sigma = \sigma' \\
\tilde{\Psi} & . M/u \stackrel{r}{\alpha}[\bar{\psi}] (R N) = (R' N') \quad \text{where} \quad \tilde{\Psi} . M/u \stackrel{r}{\alpha}[\bar{\psi}] R = R' \quad \text{and} \quad \tilde{\Psi} . M/u \stackrel{s}{\alpha}[\bar{\psi}] (N) = N' \\
\tilde{\Psi} & . M/u \stackrel{r}{\alpha}[\bar{\psi}] (R N') = M' : \alpha_2 \quad \text{if} \quad \tilde{\Psi} . M/u \stackrel{r}{\alpha}[\bar{\psi}] R = \lambda x. M_0 : \alpha_1 \rightarrow \alpha_2 \quad \text{for} \quad \alpha_1 \rightarrow \alpha_2 \leq \alpha[\bar{\psi}] \\
\quad \quad \text{and} \quad \tilde{\Psi} . M/u \stackrel{s}{\alpha}[\psi] (N) = N' \quad \text{and} \quad [N'/x]^* \sigma \ (M_0) = M' \\
\tilde{\Psi} & . M/u \stackrel{r}{\alpha}[\bar{\psi}] (\proj R) = \proj R' \quad \text{if} \quad \tilde{\Psi} . M/u \stackrel{s}{\alpha}[\bar{\psi}] (R) = R' \\
\tilde{\Psi} & . M/u \stackrel{r}{\alpha}[\bar{\psi}] (\proj R) = M_i : \alpha_i \quad \text{if} \quad \tilde{\Psi} . M/u \stackrel{s}{\alpha}[\bar{\psi}] (R) = (M_1, M_2) : \alpha_1 \times \alpha_2 \\
\tilde{\Psi} & . M/u \stackrel{r}{\alpha}[\bar{\psi}] (R) \quad \text{fails} \quad \text{otherwise}
\end{align*}
\]

A.2.2 Contextual substitution for parameter variables

Contextual substitution for parameter variables follows similar principles, but substitutes an ordinary variable for a parameter variable. We write parameter substitutions as \([x/p]^* \sigma \) where \( \ast \in \{n, r, s, a\} \). When we encounter a parameter variable \( p[\sigma] \), we replace \( p \) with the ordinary variable \( x \) and apply the substitution \([x/p]^* \sigma \) to \( \sigma \) obtaining a substitution \( \sigma' \). Instead of returning a closure \( x[\sigma'] \) as the final result we apply \( \sigma' \) to the ordinary variable \( x \). This may again yield a normal term, so we must ensure that contextual substitution for parameter variables preserves normal forms.

\[
\begin{align*}
[x/p]^* \sigma (p[\sigma]) &= M : \alpha \quad \text{if} \quad [x/p]^* \sigma = \sigma' \quad \text{and} \quad [\sigma'/\bar{\psi}]^* x = M : \alpha \\
[x/p]^* \sigma (p[\sigma]) &= R \quad \text{if} \quad [x/p]^* \sigma = \sigma' \quad \text{and} \quad [\sigma'/\bar{\psi}]^* x = R \\
[x/p]^* \sigma (p'[\sigma]) &= p'[\sigma'] \quad \text{where} \quad [x/p]^* \sigma = \sigma'
\end{align*}
\]

Theorem A.5 (Termination)

\( \tilde{\Psi} . M/u \stackrel{s}{\alpha}[\bar{\psi}] ( \_ ) \) and \( \tilde{\Psi} . x/p \stackrel{s}{\alpha}[\bar{\psi}] ( \_ ) \) where \( \ast \in \{n, r, s, a\} \) terminate, either by returning a result or failing after a finite number of steps.

Theorem A.6 (Contextual Substitution Principles)

(i) If \( \Delta_1; \Phi \vdash M \Leftarrow A \) and \( \Delta_1, u : A[\Phi], \Delta_2; \Psi \vdash J \)
then \( \Delta_1, \tilde{\Psi}.M/u \stackrel{s}{\alpha}[\bar{\psi}] \Delta_2; \tilde{\Psi}.M/u \stackrel{s}{\alpha}[\bar{\psi}] \Psi \vdash \tilde{\Psi}.M/u \stackrel{s}{\alpha}[\bar{\psi}] J \) where \( \ast \in \{n, r, s, a\} \).

(ii) If \( \Delta_1; \Phi \vdash x \Rightarrow A \) and \( \Delta_1, p : A[\Phi], \Delta_2; \Psi \vdash J \)
then \( \Delta_1, \tilde{\Psi}.x/p \stackrel{s}{\alpha}[\bar{\psi}] \Delta_2; \tilde{\Psi}.x/p \stackrel{s}{\alpha}[\bar{\psi}] \Psi \vdash \tilde{\Psi}.x/p \stackrel{s}{\alpha}[\bar{\psi}] J \) where \( \ast \in \{n, r, s, a\} \).