Programs Using Syntax with First-Class Binders

Francisco Ferreira and Brigitte Pientka

McGill University
fferre8@cs.mcgill.ca, bpientka@cs.mcgill.ca

Abstract. We present a general methodology for adding support for higher-order abstract syntax definitions and first-class contexts to an existing ML-like language. As a consequence, low-level infrastructure that deals with representing variables and contexts can be factored out. This avoids errors in manipulating low-level operations, eases the task of prototyping program transformations and can have a major impact on the effort and cost of implementing such systems.
We allow programmers to define syntax in a variant of the logical framework LF and to write programs that analyze these syntax trees via pattern matching as part of their favorite ML-like language. The syntax definitions and patterns on syntax trees are then eliminated via a translation using a deep embedding of LF that is defined in ML. We take advantage of GADTs which are frequently supported in ML-like languages to ensure our translation preserves types. The resulting programs can be type checked reusing the ML type checker, and compiled reusing its first-order pattern matching compilation. We have implemented this idea in a prototype written for and in OCaml and demonstrated its effectiveness by implementing a wide range of examples such as type checkers, evaluators, and compilation phases such as CPS translation and closure conversion.

Keywords: Higher-Order Abstract Syntax, Programming with Binders, Functional Programming, ML

1 Introduction

Writing programs that manipulate other programs is a common activity for a computer scientist, either when implementing interpreters, writing compilers, or analyzing phases for static analysis. This is so common that we have programming languages that specialize in writing these kinds of programs. In particular, ML-like languages are well-suited for this task thanks to recursive data types and pattern matching. However, when we define syntax trees for realistic input languages, there are more things on our wish list: we would like support for representing and manipulating variables and tracking their scope; we want to compare terms up-to α-equivalence (i.e. the renaming of bound variables); we would like to avoid implementing capture avoiding substitutions, which is tedious and error-prone. ML languages typically offer no high-level abstractions or support for manipulating variables and the associated operations on abstract syntax trees.

Over the past decade, there have been several proposals to add support for defining and manipulating syntax trees into existing programming environments.

For example: FreshML [22], the related system Romeo [23], and Cαml [20] use Nominal Logic [18] as a basis and the Hobbits library for Haskell [25] uses a name based formalism. In this paper, we show how to extend an existing (functional) programming language to define abstract syntax trees with variable binders based on higher-order abstract syntax (HOAS) (sometimes also called λ-trees [11]). Specifically, we allow programmers to define object languages in the simply-typed λ-calculus where programmers use the intentional function space of the simply typed λ-calculus to define binders (as opposed to the extensional function space of ML). Hence, HOAS representations inherit α-renaming from the simply-typed λ-calculus and we can model object-level substitution for HOAS trees using β-reduction in the underlying simply-typed λ-calculus. We further allow programmers to express whether a given sub-tree in the HOAS tree is closed by using the necessity modality of S4 [6]. This additional expressiveness is convenient to describe that sub-trees in our abstract syntax tree are closed.

Our work follows the pioneering work of HOAS representations in the logical framework LF [9]. On the one hand we restrict it to the simply-typed setting to integrate it smoothly into existing simply-typed functional programming languages such as OCaml, and on the other hand we extend its expressiveness by allowing programmers to distinguish between closed and open parts of their syntax trees. As we analyze HOAS trees, we go under binders and our sub-trees may not remain closed. To model the scope of binders in sub-trees we pair a HOAS tree together with its surrounding context of variables following ideas from Beluga [15,12]. In addition, we allow programmers to pattern match on such contextual objects, i.e. an HOAS tree together with its surrounding context.

Our contribution is two-fold: First, we present a general methodology for adding support for HOAS tree definitions and first-class contexts to an existing (simply-typed) programming language. In particular, programmers can define simply-typed HOAS definitions in the syntactic framework (SF) based on modal S4 following [12,6]. In addition, programmers can manipulate and pattern match on well-scoped HOAS trees by embedding HOAS objects together with their surrounding context into the programming language using contextual types [15]. The result is a programming language that can express computations over open HOAS objects. We describe our technique abstractly and generically using a language that we call Core-ML. In particular, we show how Core-ML with first-class support for HOAS definitions and contexts can be translated in into a language Core-ML$^{\text{gadt}}$ that supports Generalized Abstract Data Types (GADTs) using a deep (first-order) embedding of SF and first-class contexts (see Fig. 1 for an overview). We further show that our translation preserves types.

Second, we show how this methodology can be realized in OCaml by describing our prototype Babybel[1]. In our implementation of Babybel we take advantage of the sophisticated type system, in particular GADTs, that OCaml provides to ensure our translation is type-preserving. By translating HOAS objects together with their context to a first-order representation in OCaml with GADTs we can also reuse OCaml's first-order pattern matching compilation allowing for

[1] available at `www.github.com/fferreira/babybel/`

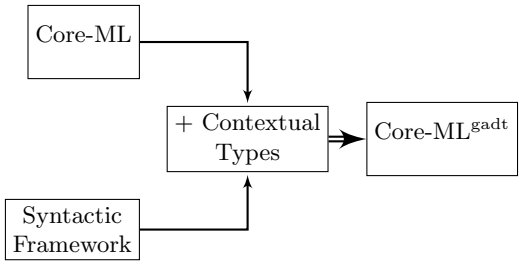

Fig. 1. Adding Contextual Types to ML

a straightforward compilation. Programmers can also exploit OCaml's impure features such as exceptions or references when implementing programs that manipulate HOAS syntax trees. We have used Babybel to implement a type-checker, an evaluator, closure conversion (shown in Section 2.3 together with a variable counting example and a syntax desugaring examples), and a continuation passing style translation. These examples demonstrate that our approach allows programmers to write programs that operate over abstract syntax trees in a manner that is safe and effective.

Finally, we would like to stress that our translation which eliminates the language extensions and permits programmers to define, analyze and manipulate HOAS trees is not specific to OCaml or even to simple types in our implementation. The same approach could be implemented in Haskell, and with some care (to be really useful it would need an equational theory for substitutions) this technique can be extended to a dependently typed language.

2 Main Ideas

In this section, we show some examples that illustrate the use of Babybel, our proof of concept implementation where we embed the syntactic framework SF inside OCaml. To smoothly integrate SF into OCaml, Babybel defines a PPX filter (a mechanism for small syntax extensions for OCaml). In particular, we use attributes and quoted strings to implement our syntax extension.

2.1 Example: Removing Syntactic Sugar

In this example, we describe the compact and elegant implementation of a compiler phase that de-sugars programs functional programs with let-expressions by translating them into function applications. We first specify the syntax of a simple functional language that we will transform. To do this we embed the syntax specification using this tag:

```
[@@@signature {def| ... |def}]
```

Inside the **@@@signature** block we will embed our SF specifications.

Our source language is defined using the type `tm`. It consists of constants (written as `cst`), pairs (written as `pair`), functions (built using `lam`), applications (built using `app`), and let-expressions.

```
[@@@signature {def|
tm : type.
cst     : tm.
pair    : tm → tm → tm.
lam     : (tm → tm) → tm.
fst     : tm → tm.
snd     : tm → tm.
letpair : tm → (tm → tm → tm) → tm.
letv    :  tm → (tm → tm) → tm.
app     : tm → tm → tm.
|def}]
```

Our definition of the source language exploits HOAS using the function space of our syntactic framework SF to represent binders in our object language. For example, the constructor `lam` takes as an argument a term of type `tm → tm`. Similarly, the definition of let-expressions models variable binding by falling back to the function space of our meta-language, in our case the syntactic framework SF. As a consequence, there is no constructor for variables in our syntactic definition and moreover we can reuse the substitution operation from the syntactic framework SF to model substitution in our object language. This avoids building up our own infrastructure for variables bindings.

We now show how to simplify programs written in our source language by replacing uses of `letpair` in terms with projections, and uses of `letv` by β reduction. Note how we use higher-order abstract syntax to represent let-expressions and abstractions.

$$\texttt{letv}\ M\ (\lambda\texttt{x}.N) \equiv \mathrm{N}[M/\texttt{x}]$$
$$\texttt{letpair}\ M\ (\lambda\texttt{x}.\lambda\texttt{y}.\ N) \equiv \mathrm{N}[(\texttt{fst}\ M\ )/\texttt{x},(\texttt{snd}\ M)/\texttt{y}]$$

To implement this simplification phase we implement an OCaml program `rewrite`: it analyzes the structure of our terms, calls itself on the sub-terms, and eliminates the use of the let-expressions into simpler constructs. As we traverse terms, our sub-terms may not remain closed. For simplicity, we use the same language as source and target for our transformation. We therefore specify the type of the function `rewrite` using contextual types pairing the type `tm` together with a context γ in which the term is meaningful inside the tag `[@type .... ]`.

```
rewrite[@type γ.[γ ⊢ tm]→[γ ⊢ tm]]
```

The type can be read: for all contexts γ, given a `tm` object in the context γ, we return a `tm` object in the same context. In general, contextual types associate a context and a type in the syntactic framework SF. For example if we want to specify a term in the empty context we would write `[ ⊢ tm]` or for a term that depends on some context with at least one variable and potentially more we would write `[γ,x:tm ⊢ tm]`.

We now implement the function `rewrite` by pattern matching on the structure of a contextual term. In Babybel, contextual terms are written inside boxes ($(\!|\ldots|\!)$) and contextual patterns inside ($(\!|\ldots|\!)_p$).

```
let rec rewrite[@type γ.[γ ⊢ tm]→[γ ⊢ tm]]
= function
| (| cst |)_p → (|cst|)
| (| pair 'm 'n |)_p → let mm, nn = rewrite m, rewrite n
                       in (|pair 'mm 'nn|)
| (| fst 'm |)_p → let mm = rewrite m in (|fst 'mm|)
| (| snd 'm |)_p → let mm = rewrite m in (|snd 'mm|)
| (| app 'm 'n |)_p → let mm,nn = rewrite m, rewrite n in
                      (|app 'mm 'nn|)
| (| lam (λx. 'm) |)_p → let mm = rewrite m in (|lam (λx. 'mm)|)
| (| #x |)_p → (|#x|)
| (| letpair 'm (λf.λs. 'n) |)_p → let mm = rewrite m in
                                   rewrite (|'n [snd 'mm;fst 'mm]|)
| (| letv 'm (λx. 'n) |)_p → rewrite (|'n['m]|)
```

Note that we are pattern matching on potentially open terms. Although we do not write the context γ explicitly, in general patterns may mention their context (i.e.: $(\!|\_ \vdash \texttt{cst}|\!)_p$[2]. As a guiding principle, we may omit writing contexts, if they do not mention variables explicitly and are irrelevant at run-time. Inside patterns or terms, we specify incomplete terms using quoted variables (e.g.: `'m`). Quoted variables are an 'unboxing' of a computational expression inside the syntactic framework SF. The quote signals that we are mentioning the computational variable inside SF.

The interesting cases are the let-expressions. For them, we perform the rewriting according to the two rules given earlier. The syntax of the substitutions puts in square brackets the terms that will be substituted for the variables. We consider contexts and substitutions ordered, this allows for efficient implementations and more lightweight syntax (e.g.: substitutions omit the name of the variables because contexts are ordered). Importantly, the substitution is an operation that is eagerly applied during run-time and not part of the representation. Consequently, the representation of the terms remains normal and substitutions cannot be written in patterns. We come back to this design decision later.

To translate contextual SF objects and contexts, Babybel takes advantage of OCaml's advanced type system. In particular, we use Generalized Abstract Data Types [4,26] to index types with the contexts in which they are valid. Type indices, in particular contexts, are then erased at run-time.

2.2 Finding the Path to a Variable

In this example, we compute the path to a specific variable in an abstract syntax tree describing a lambda-term. This will show how to specify particular context

[2] The underscore means that there might be a context but we do not bind any variable for it because contexts are not available at run-time.

shapes, how to pattern match on variables, how to manage our contexts, and how the Babybel extensions interact seamlessly with OCaml's impure features. For this example, we concentrate on the fragment of terms that consists only of abstractions and application which we repeat here.

```
[@@@signature {def|
tm : type.
app : tm → tm → tm.
lam : (tm → tm) → tm.
|def}]
```

To find the first occurrence of a particular variable in the HOAS tree, we use backtracking that we implement using the user-defined OCaml exception `Not_found`. To model the path to a particular variable occurrence in the HOAS tree, we define an OCaml data type `step` that describes the individual steps we take and finally model a `path` as a list of individual steps.

```
exception Not_found
type step
= Here  (*the path ends here*)
| AppL  (*take left on app*)
| AppR  (*take right on app*)
| InLam (*go inside the body of the term*)
type path = step list
```

The main function `path_aux` takes as input a term that lives in a context with at least one variable and returns a path to the occurrence of the top-most variable or an empty list, if the variable is not used. Its type is:

```
[@type γ. [γ, x:tm ⊢ tm] → path].
```

We again quantify over all contexts γ and require that the input term is meaningful in a context with at least one variable. This specification simply excludes closed terms since there would be no top-most variable. Note also how we mix in the type annotation to this function both contextual types and OCaml data types.

```
let rec path_aux [@type γ.[γ, x:tm ⊢ tm] → path]
= function
| (|_, x ⊢ x|)_p→ [Here]
| (|_, x ⊢ #y|)_p→ raise Not_found
| (|_, x ⊢ lam (λy. 'm)|)_p→ InLam::(path_aux(|_,x,y ⊢ 'm[_;y;x]|))
| (|_, x ⊢ app 'm 'n|)_p→ try AppL::(path_aux m)
                          with _ → AppR::(path_aux n)
```

All patterns in this example make the context explicit, as we pattern match on the context to identify whether the variable we encounter refers to the top-most variable declaration in the context. The underscore simply indicates that there might be more variables in the context. The first case, matches against the bound variable `x`. The second case has a special pattern with the sharp symbol, the pattern `#y` matches against any variable in the context `_, x`. Because of the first

pattern if it had been `x` it would have matched the first case. Therefore, it simply raises the exception to backtrack to the last choice we had.

The case for lambda expressions is interesting because the recursive call happens in an extended context. Furthermore, in order to keep the variable we are searching for on top, we need to swap the two top-most variables. For that purpose, we apply the `[_ ; y; x]` substitution. In this substitution the underscore stands for the identity on the rest of the context, or more precisely, the appropriate shift in our internal representation that relies on de Bruijn encoding. Once elaborated, this substitution becomes `[↑2 ; y; x]` where the shift by two arises, because we are swapping variables as opposed to instantiating them.

The final case is for applications. We first look on the left side and if that raises an exception we catch it and search again on the right. We again use quoted variables (e.g.: `'m`) to bind and refer to ML variables in patterns and terms of the syntactic framework and more generally be able to describe incomplete terms.

```
let get_path [@type γ.[γ, x:tm ⊢ tm] → path]
= fun t → try path_aux t with _ → []
```

The `get_path` function has the same type as the `path_aux` function. It simply handles the exception and returns an empty path in case that variable `x` is not found in the term.

2.3 Closure Conversion

In the final example, we describe the implementation of a naive algorithm for closure conversion for untyped λ-terms following [3]. We take advantage of the syntactic framework SF to represent source terms (using the type family `tm`) and closure-converted terms (using the type family `ctm`). In particular, we use SF's closed modality box to ensure that all functions in the target language are closed. This is impossible when we simply use LF as the specification framework for syntax as in [3]. We omit here the definition of lambda-terms, our source language, that was given in the previous section and concentrate on the target language `ctm`.

```
[@@@signature {def|
ctm: type.  % closed term
btm: type.  % binder term
sub: type.  % environment

capp   : ctm → ctm → ctm.
clam   : {btm} → ctm.
clo    : ctm → sub → ctm.
embed  : ctm → btm.
bind   : (ctm → btm) → btm.

empty  : sub.
dot    : sub → ctm → sub.
|def}]
```

Applications in the target language are defined using the constructor `capp` and simply take two target terms to form an application. Functions (constructor `clam`), however, take a `btm` object wrapped in `{}` braces. This means that the object inside the braces is closed. The curly braces denote the internal closed modality of the syntactic framework. As the original functions may depend on variables in the environment, we need closures where we pair a function with a substitution that points to the appropriate environment. We define our own substitutions explicitly, because they are part of the target language and the built-in substitution is an operation on terms that is eagerly computed away. Inside the body of the function, we need to bind all the variables from the environment that the body uses such that later we can instantiate them applying the substitution. This is achieved by defining multiple bindings using constructors `bind` and `embed` inside the term.

When writing a function that translates between representations, their open terms depend on contexts that store assumptions of different representations and it is often the case that one needs to relate these contexts. In our example here we define a context relation that keeps the input and output contexts in sync using a GADT data type `rel` in OCaml where we model contexts as types. The relation statically checks correspondence between contexts, but it is also available at run-time (i.e. after type-erasure).

```
type (_ , _) rel  =
    Empty : ([.], [.]) rel
  | Both  : ([γ], [δ]) rel →  ([γ, x:tm], [δ, y:ctm]) rel

exception Error of string

let rec lookup [@type γ δ.[γ ⊢ tm]→(γ, δ) rel→[δ ⊢ ctm]] =
fun t → function
| Both r' → begin match t with
  | ⟪ _,x ⊢ x ⟫p → ⟪_,x ⊢ x⟫
  | ⟪ _,x ⊢ ##v ⟫p → let v1 = lookup (⟪#v⟫) r'
                      in ⟪_, x ⊢ 'v1 [_]⟫
  | _ → raise Error (``Term that is not a variable'')
| Empty → raise Error (``Term is not a variable'')
end
```

The function `lookup` searches for related variables in the context relation. If we have a source context γ`,x:tm` and a target context δ`,y:ctm`, then we consider two variable cases: In the first case, we use matching to check that we are indeed looking for the top-most variable `x` and we simply return the corresponding target variable. If we encounter a variable from the context, written as `##v`, then we recurse in the smaller context stripping off the variable declaration `x`. Note that `##v` denotes a variable from the context `_`, that is not `x`, while `#v` describes a variable from the context `_, x`, i.e. it could be also `x`. The recursive call returns the corresponding variable `v1` in the target context that does not include the variable declaration `x`. We hence need to weaken `v1` to ensure it is meaningful in

the original context. We therefore associate `'v1` with the identity substitution for the appropriate context, namely: `[_]`. In this case, it will be elaborated into a one variable shift in the internal de Bruijn representation that is used in our implementation. The last case returns an exception whenever we are trying to look up in the context something that is not a variable.

As we cannot express at the moment in the type annotation that the input to the lookup function is indeed only a variable from the context γ and not an arbitrary term, we added another fall-through case for when the context is empty. In this case the input term cannot be a variable, as it would be out of scope.

Finally, we implement the function `conv` which takes an untyped source term in a context γ and a relation of source and target variables, described by `(γ, δ) rel` and returns the corresponding target term in the target context δ.

```
let rec close [@type γ δ. (γ, δ) rel→[δ ⊢ btm]→[btm]]
= fun r m → match r with
| Empty → m
| Both r → close r (|bind (λx. 'm)|)

let rec envr [@type γ δ. (γ, δ)rel→[δ ⊢ sub]]
= fun r → match r with
| Empty → (|empty|)
| Both r →
  let s = envr r in (|_, x ⊢ dot ('s[_]) x|)

let rec conv [@type γ δ.(γ, δ)rel→[γ ⊢tm]→[δ ⊢ctm]]
= fun r m → match m with
| (| lam (λx. 'n)  |)_p →
  let mc = conv (Both r) n in
  let mb = close r (|bind(λx. embed 'mc)|)
  in let s = envr r in (|clo (clam {'mb}) 's|)
| (|#x|)_p → lookup (|#x|) r
| (|app 'm 'n|)_p → let mm, nn = conv r m, conv r n in
                    (|capp 'mm 'nn|)
```

The core of the translation is defined in functions `conv`, `envr`, and `close`. The main function is `conv`. It is implemented by recursion on the source term. There are three cases: i) source variables simply get translated by looking them up in the context relation, ii) applications just get recursively translated each term in the application, and iii) lambda expressions are translated recursively by converting the body of the expression in the extended context (notice the recursive call with `Both r`) and then turning the lambda expression into a closure.

In the first step we generate the closed body by the function `close` that adds the multiple binders (constructors `bind` and `embed`) and generates the closed term. Note that the return type `[btm]` of `close` guarantees that the final result is indeed a closed term, because we omit the context. For clarity, we could have written `[ ⊢ btm]`.

Finally, the function `envr` computes the substitution (represented by the type `sub`) for the closure.

The implementation of closure conversion shows how to enforce closed terms in the specification, and how to make contexts and their relationships explicit at run-time using OCaml's GADTs. We believe it also illustrates well how HOAS trees can be smoothly manipulated and integrated into OCaml programs that may use effects.

3 Core-ML: A Functional Language with Pattern Matching and Data Types

We now introduce Core-ML, a functional language based on ML with pattern matching and data types. In Section 5 we will extend this language to also support contextual types and terms in our syntactic framework SF.

We keep the language design of Core-ML minimal in the interest of clarity. However, our prototype implementation which we describe in Section 9 supports interaction with all of OCaml's features such as exceptions, references and GADTs.

$$
\begin{array}{lll}
\text{Types} & & \tau ::= D \mid \tau_1 \to \tau_2 \\
\text{Expressions} & & e ::= i \mid \texttt{fun}\, f(x) = e \mid \texttt{let}\, x = i\, \texttt{in}\, e \mid \texttt{match}\, i\, \texttt{with}\ \overrightarrow{b} \\
\text{Neutral Exp.} & & i ::= i\, e \mid C\ \overrightarrow{e} \mid x \mid e : \tau \\
\text{Patterns} & & pat ::= C\ \overrightarrow{pat} \mid x \\
\text{Branches} & & b ::= \mid pat \mapsto e \\
\text{Contexts} & & \Gamma ::= \cdot \mid \Gamma, x : \tau \\
\text{Signature} & & \Xi ::= \cdot \mid \Xi, D \mid \Xi, C : \overrightarrow{\tau} \to D
\end{array}
$$

In Core-ML, we declare data-types by adding type formers (D) and type constructors (C) to the signature (Ξ). Constructors must be fully-applied. In addition all functions are named and recursive. The language supports pattern matching with nested patterns where patterns consist of just variables and fully applied constructors. We assume that all patterns are linear (i.e. each variable occurs at most once) and that they are covering.

The bi-directional typing rules for Core-ML have access to a signature Ξ and are standard (see Fig. 2). For lack of space, we omit the operational semantics which is standard. We also will not address the details of pattern matching compilation but merely state that it is possible to implement it in an efficient manner using decision trees [1].

4 A Syntactic Framework

In this section we describe the Syntactic Framework (SF) based on S4 [6]. Our framework characterizes only normal forms. All computation is delegated to the ML layer, that will perform pattern matching and substitutions on terms.

$\boxed{\Gamma \vdash e \Leftarrow \tau}$: Expression e checks against type τ in context Γ

$$\frac{\Gamma, f : \tau \to \tau', x : \tau \vdash e \Leftarrow \tau'}{\Gamma \vdash \texttt{fun}\, f(x) = e \Leftarrow \tau \to \tau'}\ \texttt{t-rec} \qquad \frac{\Gamma \vdash i \Rightarrow \tau' \quad \Gamma, x : \tau' \vdash e \Leftarrow \tau}{\Gamma \vdash \texttt{let}\, x = i\, \texttt{in}\, e \Leftarrow \tau}\ \texttt{t-let}$$

$$\frac{\Gamma \vdash i \Rightarrow \tau' \quad \forall b_k \in \overrightarrow{b}\ .\ \Gamma \vdash b_k \Leftarrow \tau' \to \tau}{\Gamma \vdash \texttt{match}\, i\, \texttt{with}\ \overrightarrow{b} \Leftarrow \tau}\ \texttt{t-match} \qquad \frac{\Gamma \vdash i \Rightarrow \tau' \quad \tau = \tau'}{\Gamma \vdash i \Leftarrow \tau}\ \texttt{t-emb}$$

$\boxed{\Gamma \vdash i \Rightarrow \tau}$: Neutral expr. i synthesizes type τ in context Γ

$$\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau}\ \texttt{t-ann} \qquad \frac{\Xi(C) = \overrightarrow{\tau} \to D \quad \forall \tau_i \in \overrightarrow{\tau}\ .\ \forall e_i \in \overrightarrow{e}\ .\ \Gamma \vdash e_i \Leftarrow \tau_i}{\Gamma \vdash C\, \overrightarrow{e} \Rightarrow D}\ \texttt{t-constr}$$

$$\frac{\Gamma \vdash i \Rightarrow \tau' \to \tau \quad \Gamma \vdash e \Leftarrow \tau'}{\Gamma \vdash i\, e \Rightarrow \tau}\ \texttt{t-app} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x \Rightarrow \tau}\ \texttt{t-var}$$

$\boxed{\Gamma \vdash\mid pat \mapsto e \Leftarrow \tau_1 \to \tau_2}$: Branch $\mid pat \mapsto e$ checks against types τ_1 and τ_2 in Γ

$$\frac{\vdash pat : \tau' \downarrow \Gamma' \quad \Gamma, \Gamma' \vdash e \Leftarrow \tau}{\Gamma \vdash\mid pat \mapsto e \Leftarrow \tau' \to \tau}\ \texttt{t-branch}$$

$\boxed{\vdash pat : \tau \downarrow \Gamma}$: Pattern pat is of type τ and binds variables in context Γ

$$\frac{}{\vdash x : \tau \downarrow x : \tau}\ \texttt{t-pat-var}$$

$$\frac{\Xi(C) = \overrightarrow{\tau} \to D \qquad \forall \tau_i \in \overrightarrow{\tau}\ .\ \forall pat_i \in \overrightarrow{pat}\ .\quad \vdash pat_i : \tau_i \downarrow \Gamma_i}{\vdash C\, \overrightarrow{pat} : D \downarrow \Gamma_1, ..., \Gamma_i}\ \texttt{t-pat-con}$$

Fig. 2. Core-ML Typing Rules

4.1 The definition of SF

The Syntactic Framework (SF) is a simply typed λ-calculus based on S4 where the type system forces all variables to be of base type, and all constants declared in a signature Σ to be fully applied. This simplifies substitution, as variables of base type cannot be applied to other terms, and in consequence, there is no need for hereditary substitution in the specification language. Finally, the syntactic framework supports the box type to describe closed terms [13]. It can also be viewed as a restricted version of the contextual modality in [12] which could be an interesting extension to our work.

Having closed objects enforced at the specification level is not strictly necessary. However, being able to state that some objects are closed in the specification has two distinct advantages: first, the user can specify some objects as closed so their contexts are always empty. This removes the need for some unnecessary substitutions. Second, it allows us to encode more fine-grained invariants and is hence an important specification tool (i.e. when implementing closure conversion

in Section 2.3).

$$\begin{array}{lll} \text{Types} & A, B & ::= \mathbf{a} \mid A \to B \mid \Box\, A \\ \text{Terms} & M, N & ::= \mathbf{c}\, \overrightarrow{M} \mid \lambda x.M \mid \{M\} \mid x \\ \text{Contexts} & \Psi, \Phi & ::= \cdot \mid \Psi, x : \mathbf{a} \\ \text{Signature} & \Sigma & ::= \cdot \mid \Sigma, \mathbf{a} : K \mid \Sigma, \mathbf{c} : A \end{array}$$

Fig. 3 shows the typing rules for the syntactic framework. Note that constructors always are fully applied (as per rule `t-con`), and that all variables are of base type as enforced by rules `t-var` and `t-lam`.

$\boxed{\Psi \vdash M : A}$: M has type A in context Ψ

$$\frac{\Psi, x : \mathbf{a} \vdash M : A}{\Psi \vdash \lambda x.M : \mathbf{a} \to A}\ \texttt{t-lam} \qquad \frac{\cdot \vdash M : A}{\Psi \vdash \{M\} : \Box\, A}\ \texttt{t-box} \qquad \frac{\Psi(x) = \mathbf{a}}{\Psi \vdash x : \mathbf{a}}\ \texttt{t-var}$$

$$\frac{\Sigma(\mathbf{c}) = A \quad \Psi \vdash \overrightarrow{M} : A/\mathbf{a}}{\Psi \vdash \mathbf{c}\, \overrightarrow{M} : \mathbf{a}}\ \texttt{t-con}$$

$\boxed{\Psi \vdash \overrightarrow{M} : A/\mathbf{a}}$: spine $\overrightarrow{M}$ checks against type A and has target type $\mathbf{a}$

$$\frac{}{\Psi \vdash \cdot : \mathbf{a}/\mathbf{a}}\ \texttt{t-sp-em} \qquad \frac{\Psi \vdash N : A \quad \Psi \vdash \overrightarrow{M} : B/\mathbf{a}}{\Psi \vdash N, \overrightarrow{M} : A \to B/\mathbf{a}}\ \texttt{t-sp}$$

Fig. 3. Syntactic Framework Typing

4.2 Contextual Types

We use contextual types to embed possibly open SF objects in Core-ML and ensure that they are well-scoped. Contextual types pair the type A of an SF object together with its surrounding context Ψ in which it makes sense. This follows the design of Beluga [15,3].

$$\begin{array}{lll} \text{Contextual Types} & U & ::= [\Psi \vdash A] \\ \text{Type Erased Contexts} & \hat{\Psi} & ::= \cdot \mid \hat{\Psi}, x \\ \text{Contextual Objects} & C & ::= [\hat{\Psi} \vdash M] \end{array}$$

Contextual objects, written as $[\hat{\Psi} \vdash M]$ pair the term M with the variable name context $\hat{\Psi}$ to allow for α-renaming of variables occurring in M. Note how the $\hat{\Psi}$ context just corresponds to the context with the typing assumptions erased.

When we embed contextual objects in a programming language we want to refer to variables and expressions from the ambient language, in order to support incomplete terms. Following [12,15], we extend our syntactic framework SF with

two ideas: first, we have incomplete terms with meta-variables to describe holes in terms. As in Beluga, there are two different kinds: *quoted variables* `'u` represent a hole in the term that may be filled by an arbitrary term. In contrast, *parameter variables* `v` represent a hole in a term that may be filled only with some bound variable from the context. Concretely, a parameter variable may be `#x` and describe any concrete variable from a context Ψ. We may also want to restrict what bound variables a parameter variable describes. For example, if we have two sharp signs (i.e. `##x`) the top-most variable declaration is excluded. Intuitively, the number of sharp signs, after the first, in front of `x` correspond to a weakening (or in de Bruijn lingo the number of shifts). Second, substitution operations allow us to move terms from one context to another.

We hence extend the syntactic framework SF with quoted variables, parameter variables and closures, written as $M[\sigma]^{\Phi}_{\Psi}$. We annotate the substitution with its domain and range to simplify the typing rule, however our prototype omits these typing annotations and lets type inference infer them.

$$
\begin{array}{ll}
\text{Parameter Variables} & \mathtt{v} ::= \mathtt{\#x} \mid \mathtt{\#v} \\
\text{Terms} & M ::= \cdots \mid \mathtt{'u} \mid \mathtt{v} \mid M[\sigma]^{\Phi}_{\Psi} \\
\text{Substitutions} & \sigma ::= \cdot \mid \sigma, M/x \\
\text{Ambient Ctx.} & \Gamma ::= \cdots \mid \Gamma, \mathtt{u} : [\Psi \vdash \mathbf{a}]
\end{array}
$$

In addition, we extend the context Γ of the ambient language Core-ML to keep track of assumptions that have a contextual type.

Finally, we extend the typing rules of the syntactic framework SF to include quoted variables, parameter variables, closures, and substitutions. We keep all the previous typing rules for SF from Section 4 where we thread through the ambient Γ, but the rules remain unchanged otherwise.

$\boxed{\Gamma; \Psi \vdash_v \mathtt{v} : \mathbf{a}}$: Parameter Variable `v` has type $\mathbf{a}$ in contexts Ψ and Γ

$$
\frac{\Gamma(x) = [\Psi \vdash \mathbf{a}]}{\Gamma; \Psi \vdash_v \mathtt{\#x} : \mathbf{a}}\ \mathtt{t\text{-}pvar\text{-}v} \qquad \frac{\Gamma; \Psi \vdash_v \mathtt{v} : \mathbf{a}}{\Gamma; \Psi, y : \_ \vdash_v \mathtt{\#v} : \mathbf{a}}\ \mathtt{t\text{-}pvar\text{-}\#}
$$

$\boxed{\Gamma; \Psi \vdash M : A}$: Term M has type A in contexts Ψ and Γ

$$
\frac{\Gamma(\mathtt{u}) = [\Psi \vdash \mathbf{a}]}{\Gamma; \Psi \vdash \mathtt{'u} : \mathbf{a}}\ \mathtt{t\text{-}qvar} \qquad \frac{\Gamma; \Psi \vdash_v \mathtt{v} : \mathbf{a}}{\Gamma; \Psi \vdash \mathtt{v} : \mathbf{a}}\ \mathtt{t\text{-}pvar}
$$

$$
\frac{\Gamma; \Psi \vdash \sigma : \Phi \quad \Gamma; \Phi \vdash M : A}{\Gamma; \Psi \vdash M[\sigma]^{\Phi}_{\Psi} : A}\ \mathtt{t\text{-}sub}
$$

$\boxed{\Gamma; \Psi \vdash \sigma : \Psi'}$: Substitution σ has domain Ψ' and range Ψ in the amb. ctx. Γ

$$
\frac{}{\Gamma; \Psi \vdash \cdot : \cdot}\ \mathtt{t\text{-}empty\text{-}sub} \qquad \frac{\Gamma; \Psi \vdash \sigma : \Psi' \quad \Gamma; \Psi \vdash M : \mathbf{a}}{\Gamma; \Psi \vdash \sigma, M/x : (\Psi', x : \mathbf{a})}\ \mathtt{t\text{-}dot\text{-}sub}
$$

The rules for quoted variables (`t-qvar`) and parameter variables (`t-pvar`) might seem very restrictive as we can only use a meta-variable of type $\Psi \vdash \mathbf{a}$ in the same context Ψ. As a consequence meta-variables often occur as a closure paired

with a substitution (i.e.: $\texttt{'u}\,[\sigma]^{\Phi}_{\Psi}$). This leads to the following admissible rule:

$$\frac{\Gamma(\mathtt{u}) = [\Phi \vdash \mathbf{a}] \quad \Delta; \Psi \vdash \sigma : \Phi}{\Gamma; \Psi \vdash \texttt{'u}\,[\sigma]^{\Phi}_{\Psi} : \mathbf{a}}\ \texttt{t-qvar-adm}$$

The substitution operation is straightforward to define and we omit it here. The next step is to define the embedding of this framework in a programming language that will provide the computational power to analyze and manipulate contextual objects.

5 Core-ML with Contextual Types

To embed contextual SF objects into Core-ML, we extend the syntax of Core-ML as follows:

Types	$\tau ::= \cdots \mid [\Psi \vdash \mathbf{a}]$
Expressions	$e ::= \cdots \mid [\hat{\Psi} \vdash M] \mid \texttt{cmatch}\, e\, \texttt{with}\ \overrightarrow{c}$
Patterns	$pat ::= \cdots \mid [\hat{\Psi} \vdash R]$
Contextual Branches	$c ::= \cdots \mid\mid [\Psi \vdash R] \mapsto e$

In particular, we allow programmers to directly pattern match on the syntactic structures they define in SF using the case-expression $\texttt{cmatch}\, e\, \texttt{with}\ \overrightarrow{c}$.

5.1 SF Objects as SF Patterns

The grammar of SF patterns follows the grammar of SF objects.

SF Parameter Pattern	$\mathtt{w} ::= \texttt{\#p} \mid \texttt{\#w}$
SF Patterns	$R ::= \lambda x.R \mid \{R\} \mid x \mid \mathbf{c}\ \overrightarrow{R} \mid \texttt{'u} \mid \mathtt{w}$

However, there is an important restriction: closures are not allowed in SF patterns. Intuitively this means that all quoted variables are associated with the identity substitution and hence depend on the entire context in which they occur. Parameter variables may be associated with weakening substitutions. This allows us to easily infer the type of quoted variables and parameter variables as we type check a pattern. This is described by the judgment

$\boxed{\Psi \vdash R : A \downarrow \Gamma}$: Pattern R has type A in Ψ and binds Γ

We omit these rules as they follow closely the typing rules for SF terms that are given in the previous section. We only show here the interesting rules for parameter patterns. They illustrate the built-in weakening.

$\boxed{\Psi \vdash_v \mathtt{w} : \mathbf{a} \downarrow \Gamma}$: Parameter Pattern $\mathtt{w}$ has type $\mathbf{a}$ in Ψ and binds Γ

$$\frac{}{\Psi \vdash_v \texttt{\#p} : \mathbf{a} \downarrow \mathtt{p} : [\Psi \vdash \mathbf{a}]}\ \texttt{tp-pvar} \qquad \frac{\Psi \vdash_v \mathtt{w} : \mathbf{a} \downarrow \Gamma}{\Psi, y : \mathbf{b} \vdash_v \texttt{\#w} : \mathbf{a} \downarrow \Gamma}\ \texttt{tp-pvar-\#}$$

Further, the matching algorithm for SF patterns degenerates to simple first-order matching [17] and can be defined straightforwardly. Because of space constraints, we only describe the successful matching operation. However, it is worth considering the matching rules for parameter patterns. As matching will only consider well-typed terms, we know that in the rules `m-pv` and `m-pv-#` the variable x is well-typed in the context $\hat{\Psi}$.

$\boxed{\Gamma; \hat{\Psi} \vdash_v w \doteq x/\rho}$: Param. Pattern w matches var. x from $\hat{\Psi}$ producing ρ.

$$\frac{}{\mathtt{p} : [\Psi \vdash A]; \hat{\Psi} \vdash_v \mathtt{\#p} \doteq x/\cdot, [\hat{\Psi} \vdash x]/\mathtt{p}}\ \mathtt{m\text{-}pv} \qquad \frac{x \neq y \quad \Gamma; \hat{\Psi} \vdash_v w \doteq x/\rho}{\Gamma; \widehat{\Psi, y} \vdash_v \mathtt{\#w} \doteq x/\rho}\ \mathtt{m\text{-}pv\text{-}\#}$$

$\boxed{\Gamma; \hat{\Psi} \vdash R \doteq M/\rho}$: M matches pattern R with bound vars. in $\hat{\Psi}$ producing ρ.

$$\frac{\Gamma; \hat{\Psi}, x \vdash R \doteq M/\rho}{\Gamma; \hat{\Psi} \vdash \lambda x.R \doteq \lambda x.M/\rho}\ \mathtt{m\text{-}\lambda} \qquad \frac{}{\cdot; \hat{\Psi} \vdash x \doteq x/\cdot}\ \mathtt{m\text{-}bv}$$

$$\frac{\Gamma; \cdot \vdash R \doteq M/\rho}{\Gamma; \hat{\Psi} \vdash \{R\} \doteq \{M\}/\rho}\ \mathtt{m\text{-}box} \qquad \frac{\text{for all } R_i \in \overrightarrow{R} \text{ such as } \Gamma; \hat{\Psi} \vdash R_i \doteq M_i/\rho_i}{\Gamma; \hat{\Psi} \vdash \mathbf{c}\,\overrightarrow{R} \doteq \mathbf{c}\,\overrightarrow{M}/\rho_0, \ldots, \rho_n}\ \mathtt{m\text{-}cc}$$

$$\frac{}{\mathtt{u} : [\Psi \vdash A]; \hat{\Psi} \vdash \mathtt{'u} \doteq M/\cdot, [\hat{\Psi} \vdash M]/\mathtt{u}}\ \mathtt{m\text{-}cv} \qquad \frac{\Gamma; \hat{\Psi} \vdash_v \mathtt{w} \doteq x/\rho}{\Gamma; \hat{\Psi} \vdash \mathtt{w} \doteq x/\rho}\ \mathtt{m\text{-}pv}$$

Finally, it has another important consequence: closures only appear in the branches of case-expressions. As Core-ML has a call-by-value semantics, we know the instantiations of quoted variables and parameter variables when they appear in the body of a case-expression and all closures can be eliminated by applying the substitution eagerly.

5.2 Typing Rules for Core-ML with Contextual Types

We now add the following typing rules for contextual objects and pattern matching to the typing rules of Core-ML:

$$\frac{\Gamma; \Psi \vdash M : \mathbf{a}}{\Gamma \vdash [\hat{\Psi} \vdash M] \Leftarrow [\Psi \vdash \mathbf{a}]}\ \mathtt{t\text{-}ctx\text{-}obj}$$

$$\frac{\Gamma \vdash i \Rightarrow [\Psi \vdash \mathbf{a}] \quad \forall b \in \overrightarrow{b}\ .\ \Gamma \vdash b \Leftarrow [\Psi \vdash \mathbf{a}] \to \tau}{\Gamma \vdash \mathtt{cmatch}\, i\, \mathtt{with}\ \overrightarrow{b} \Leftarrow \tau}\ \mathtt{t\text{-}cm}$$

$$\frac{\Psi \vdash R : \mathbf{a} \downarrow \Gamma' \quad \Gamma, \Gamma' \vdash e \Leftarrow \tau}{\Gamma \vdash [\Psi \vdash R] \mapsto e \Leftarrow [\Psi \vdash \mathbf{a}] \to \tau}\ \mathtt{t\text{-}cbranch}$$

The typing rule for contextual objects (rule `t-ctx-obj`) simply invokes the typing judgment for contextual objects. Notice, that we need the context Γ when checking contextual objects, as they may contain quoted variables from Γ.

Extending the operational semantics to handle contextual SF objects is also straightforward.

6 Core-ML with GADTs

So far we reviewed how to support contextual types and contextual objects in a standard functional programming language. This allows us to define syntactic structures with binders and manipulate them with the guarantee that variables will not escape their scopes. This brings some of the benefits of the Beluga system to mainstream languages focusing on writing programs instead of proofs. A naive implementation of this language extension requires augmenting the type checker and operational semantics of the host language. This is a rather significant task – especially if it includes implementing a compiler for the extended language. In this section, we describe how to embed Core-ML with contextual types in a functional language with GADTs, called Core-ML$^{\text{gadt}}$, based on $\lambda_{2,G\mu}$ by Xi et al. [26]. The choice of this target language is motivated by the fact that it is close to what realistic typed languages already offer (e.g.: OCaml and Haskell) and it directly lends itself to an implementation.

$$
\begin{array}{lll}
\text{Signatures} & \Sigma & ::= \cdot \mid \Sigma, D : (*, \ldots, *) \to * \mid C : \forall \overrightarrow{\alpha} \,.\, \tau \to D[\overrightarrow{\tau}] \\
\text{Types} & \tau & ::= D[\overrightarrow{\tau}] \mid \forall \alpha \,.\, \tau \mid \tau_1 \to \tau_2 \mid \alpha \mid \tau_1 \times \tau_2 \\
\text{Expressions} & e & ::= x \mid C[\overrightarrow{\tau}]\, e \mid \texttt{fix}\, f : \tau = e \mid e_1\, e_2 \mid (e_1,\, e_2) \mid \lambda x \,.\, e \\
 & & \quad \mid \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \mid \texttt{match}\, e\, \texttt{with}\ \overrightarrow{b} \mid \Lambda \alpha \,.\, e \mid e[\tau] \mid (e_1,\, e_2) \\
\text{Branch} & b & ::= pat \mapsto e \\
\text{Pattern} & pat & ::= x \mid C[\overrightarrow{\alpha}]\, pat \mid (pat_1,\, pat_2) \\
\text{Exp. Ctx.} & \Gamma & ::= \cdot \mid \Gamma, x : \tau \\
\text{Type Ctx.} & \Delta & ::= \cdot \mid \Delta, \alpha \mid \Delta, \tau_1 \equiv \tau_2
\end{array}
$$

Core-ML$^{\text{gadt}}$ contains polymorphism and GADTs, which makes it a good ersatz OCaml that is still small and easy to reason about. GADTs are particularly convenient, since they allow us to track invariants about our objects in a similar fashion to dependent types. Compared to Core-ML, Core-ML$^{\text{gadt}}$'s signatures now store type constants and constructors that are parametrized by other types. We show the typing judgments for the language in Fig. 4.

The operational semantics is fundamentally the same as the semantics for Core-ML, after all, type information is irrelevant at run-time (i.e. Core-ML$^{\text{gadt}}$ has strong type separation). The interested reader can find the operational semantics in [26].

7 Deep Embedding of SF into Core-ML$^{\text{gadt}}$

We now show how to translate objects and types defined in the syntactic framework SF into Core-ML$^{\text{gadt}}$ using a deep embedding. We take advantage of the advanced features of Core-ML$^{\text{gadt}}$'s type system to fully type-check the result. Our representation of SF objects and types is inspired by [2] but uses GADTs instead of full dependent types. We add the idea of typed context shifts, that represent weakening, to be able to completely erase types at run-time.

$\boxed{\Delta; \Gamma \vdash e : \tau}$: e is of type τ in contexts Δ and Γ.

$$\frac{\Sigma(C) = \forall \overrightarrow{\alpha} \,.\, \tau_1 \to D[\overrightarrow{\tau}] \quad \Delta; \Gamma \vdash e : \tau_1[\overrightarrow{\tau}] \quad \Delta \vdash \overrightarrow{\tau}\ \mathtt{wf}}{\Delta; \Gamma \vdash C\,\overrightarrow{\tau}\,e : D[\overrightarrow{\tau}]}\ \mathtt{g\text{-}con}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1\, e_2 : \tau_2}\ \mathtt{g\text{-}app} \qquad \frac{\Delta; \Gamma \vdash e_1 : \tau_1 \quad \Delta; \Gamma \vdash e_1 : \tau_2}{\Delta; \Gamma \vdash (e_1,\, e_2) : \tau_1 \times \tau_2}\ \mathtt{g\text{-}pair}$$

$$\frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}\ \mathtt{g\text{-}var} \qquad \frac{\Delta; \Gamma, f : \tau \vdash e : \tau}{\Delta; \Gamma \vdash \mathtt{fix}\, f : \tau = e : \tau}\ \mathtt{g\text{-}fix} \qquad \frac{\Delta; \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x \,.\, e : \tau_1 \to \tau_2}\ \mathtt{g\text{-}lam}$$

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha \,.\, \tau \quad \Delta; \Gamma \vdash \tau\ \mathtt{wf}}{\Delta; \Gamma \vdash e[\tau] : \tau_1}\ \mathtt{g\text{-}tapp} \qquad \frac{\Delta; \Gamma \vdash e_1 : \tau_1 \quad \Delta; \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Delta; \Gamma \vdash \mathtt{let}\, x = e_1\ \mathtt{in}\ e_2 : \tau}\ \mathtt{g\text{-}let}$$

$$\frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha \,.\, e : \tau}\ \mathtt{g\text{-}Lam} \qquad \frac{\Delta; \Gamma \vdash e : \tau_1 \quad \text{for all } i. \Delta; \Gamma \vdash b_i : \tau_1 \to \tau}{\Delta; \Gamma \vdash \mathtt{match}\, e\, \mathtt{with}\ \overrightarrow{b} : \tau}\ \mathtt{g\text{-}match}$$

$$\frac{\Delta; \Gamma \vdash pat : \tau \downarrow \Delta'; \Gamma' \quad \Delta, \Delta'; \Gamma, \Gamma' \vdash e : \tau_2}{\Delta; \Gamma \vdash pat \mapsto e : \tau_1 \to \tau_2}\ \mathtt{g\text{-}branch}$$

$\boxed{\Delta_o \vdash pat : \tau \downarrow \Delta; \Gamma}$: pat is of type τ and binds variables in Δ and Γ

$$\frac{\Delta_0 \vdash \tau\ \mathtt{wf}}{\Delta_0 \vdash x : \tau \downarrow \cdot; x : \tau}\ \mathtt{gp\text{-}var}$$

$$\frac{\Delta_0 \vdash pat_1 : \tau_1 \downarrow \Delta_1; \Gamma_1 \quad \Delta_0 \vdash pat_2 : \tau_2 \downarrow \Delta_2; \Gamma_2}{\Delta_0 \vdash (pat_1,\, pat_2) : \tau_1 \times \tau_2 \downarrow \Delta_1, \Delta_2; \Gamma_1, \Gamma_2}\ \mathtt{gp\text{-}pair}$$

$$\frac{\Sigma(C) = \forall \overrightarrow{\alpha} \,.\, \tau \to D[\overrightarrow{\tau_1}] \quad \Delta_0, \overrightarrow{\alpha}, \overrightarrow{\tau_1} \equiv \overrightarrow{\tau_2} \vdash pat : \tau \downarrow \Delta; \Gamma}{\Delta_o \vdash C[\overrightarrow{\alpha}]\, pat : D[\overrightarrow{\tau_2}] \downarrow \overrightarrow{\alpha}, \overrightarrow{\tau_1} \equiv \overrightarrow{\tau_2}, \Delta; \Gamma}\ \mathtt{gp\text{-}con}$$

Fig. 4. The typing of Core-ML$^{\mathrm{gadt}}$

To ensure SF terms are well-scoped and well-typed, we define SF types in Core-ML$^{\mathrm{gadt}}$ and index their representations by their type and context. The following types are only used as indices for GADTs. Because of that, they do not have any term constructors.

$$\Sigma = \mathtt{base} : * \to *, \mathtt{arr} : (*, *) \to *, \mathtt{boxed} : * \to *, \mathtt{prod} : (*, *) \to *, \mathtt{unit} : *$$

We define three type families, one for each of SF's type constructors. It is important to note the number of type parameters they require. Base types take one parameter: a type from the signature. Function types simply have a source and target type. Finally, boxes contain just one type.

Terms are also indexed by the contexts in which they are valid. To this effect, we define two types to statically represent contexts. Analogously to the representation of types, these two types are only used statically and there will be no instances at run-time. The type `nil` represents an empty context and thus has no parameters. The constructor `cons` has two parameters, the first one is

the rest of the context and the second one is the type of the top-most variable.

$$\Sigma = \ldots, \texttt{nil} : *, \texttt{cons} : (*, *) \to *$$

We show the encoding of well-typed SF objects and types in Fig. 5. Every declaration is parametrized with the type of constructors that the user defined inside of the **@@@signature** blocks.

$$
\begin{array}{l}
\Sigma = \ldots, \texttt{var} : (*, *) \to *, \\
\texttt{Top} : \forall \gamma, \alpha \;.\; \texttt{var}[\texttt{cons}[\gamma, \alpha], \alpha], \\
\texttt{Pop} : \forall \gamma, \alpha, \beta \;.\; \texttt{var}[\gamma, \alpha] \to \texttt{var}[\texttt{cons}[\gamma, \beta], \alpha], \\
\texttt{tm} : (*, *) \to *, \texttt{sp} : (*, *, *) \to * \\
\texttt{Lam} : \forall \gamma, \alpha, \tau \;.\; \texttt{tm}[\texttt{cons}[\gamma, \texttt{base}[\alpha]], \tau] \to \texttt{tm}[\gamma, \texttt{arr}[\texttt{base}[\alpha], \tau]], \\
\texttt{Var} : \forall \gamma, \alpha \;.\; \texttt{var}[\gamma, \alpha] \to \texttt{tm}[\gamma, \texttt{base}[\alpha]], \\
\texttt{Box} : \forall \gamma, \tau \;.\; \texttt{tm}[\cdot, \tau] \to \texttt{tm}[\gamma, \tau], \\
\texttt{C} : \forall \gamma, \tau, \alpha \;.\; \texttt{con}[\tau, \alpha] \times \texttt{tm}[\gamma, \tau] \to \texttt{tm}[\gamma, \texttt{base}[\alpha]], \\
\texttt{Empty} : \forall \gamma, \tau \;.\; \texttt{sp}[\gamma, \tau, \tau], \\
\texttt{Cons} : \forall \gamma, \tau_1, \tau_2, \tau_3 \;.\; \texttt{tm}[\gamma, \tau_1] \times \texttt{sp}[\gamma, \tau_2, \tau_3] \to \texttt{sp}[\gamma, \texttt{arr}[\tau_1, \tau 2], \tau_3], \\
\texttt{shift} : (*, *) \to *, \\
\texttt{Id} : \forall \gamma \;.\; \texttt{shift}[\gamma, \gamma], \\
\texttt{Suc} : \forall \gamma, \delta, \alpha \;.\; \texttt{shift}[\gamma, \delta] \to \texttt{shift}[\texttt{cons}[\gamma, \texttt{base}[\alpha]], \delta], \\
\texttt{sub} : (*, *) \to *, \\
\texttt{Shift} : \forall \gamma, \delta \;.\; \texttt{shift}[\gamma, \delta] \to \texttt{sub}[\gamma, \delta], \\
\texttt{Dot} : \forall \gamma, \delta, \tau \;.\; \texttt{sub}[\gamma, \delta] \times \texttt{tm}[\gamma, \tau] \to \texttt{sub}[\gamma, \texttt{cons}[\delta, \tau]]
\end{array}
$$

Fig. 5. Syntactic Framework Definition

The specification takes the form of the type $\texttt{con} : (*, *) \to *$, where `con` is the name of a constructor indexed by the type of its parameters and the base type they produce.

Variables and terms are indexed by two types, the first parameter is always their context and the second is their type. The type `var` represents variables with two constructors: `Top` represents the variable that was introduced last in the context and if `Top` corresponds to the de Bruijn index 0 then the constructor `Pop` represents the successor of the variable that it takes as parameter. It is interesting to consider the parameters of these constructors. `Top` is simply indexed by its context and type (variables γ and α respectively). On the other hand, `Pop` requires three type parameter: the first γ represents a context, α the resulting type of the variable, and β the type of the extension of the context. These parameter make it so that if we apply the constructor `Pop` to a variable of type α in context γ, we obtain a variable of type α in the context γ extended with type β.

As mentioned, terms described by the type family `tm` are indexed by their context and their type. It is interesting to check in some detail how the indices

of the term constructors follow the typing rules from Fig. 3. The constructor for lambda terms (`Lam`), extends the context γ with base type α and then it produces a term in γ of function type from the base type α to the type of the body τ. The constructor for boxes simply forces its body to be closed by using the context type `nil`. The constructor `Var` simply embeds variables as terms. Finally the `C` constructor has two parameters, one is the name of the constructor from the user's definitions that constrains the type of the second parameter, the other is the term of the appropriated type.

The definition of substitution is a modified presentation of the substitution for well-scoped de Bruijn indices, as for example presented in [2]. We define two types, `sub` and `shift` indexed by two contexts, the domain and the range of the substitutions. Substitutions are either a shift (constructor `Shift`) or the combination of a term for the top-most variable and the rest of the substitution (constructor `Dot`).

Our implementation differs from Benton et al.[2] in the representation of renamings. Benton et.al define substitutions and renamings, the latter as a way of representing shifts. However to compute a shift, they need the context that they use to index the data-types. Hence, contexts are not erasable during run-time. As we do want contexts to be erasable at run-time, we cannot use renamings. Instead, we replace renamings with typed shifts (defined in type `shift`), that encode how many variables we are shifting over. This is encoded in the indices of shifts.

Finally, we omit the function implementing the substitution as it is standard We will simply mention that we implement a function `apply_sub` of type: $\forall \gamma, \delta, \tau$. $\texttt{tm}[\gamma, \tau] \to \texttt{sub}[\gamma, \delta] \to \texttt{tm}[\delta, \tau]$ that applies a substitution moving a term from context γ to context δ.

8 From Core-ML with Contextual Types to Core-ML$^{\mathbf{gadt}}$

In this section, we translate Core-ML with contextual types into Core-ML$^{\text{gadt}}$. Because our embedding of the syntactic framework SF in Core-ML$^{\text{gadt}}$ is intrinsically typed, there is no need to extend the type-checker to accommodate contextual objects. Further, recall that we restricted quoted variables and parameter variables such that the matching operation remains first order. In addition, as our deep embedding uses a representation with canonical names (namely de Bruijn indices), we are able to translate pattern matching into Core-ML$^{\text{gadt}}$'s pattern matching; thus there is no need to extend the operational semantics of the language.

The translation we describe in this section provides the footprint of an implementation to directly generate OCaml code, as Core-ML$^{\text{gadt}}$ is essentially a subset of OCaml. It therefore shows how to extend a functional programming language such as OCaml with the syntactic framework with minimal impact on OCaml's compiler.

We begin by translating SF types and contexts into Core-ML$^{\text{gadt}}$ types. These types are used to index terms in the implementation of SF:

$$\begin{array}{lrl}
\text{SF Types:} & \ulcorner \mathbf{a} \to A \urcorner &= \texttt{arr}[\mathbf{a}, \ulcorner A \urcorner] \\
& \ulcorner \Box\, A \urcorner &= \texttt{boxed}[\ulcorner A \urcorner] \\
& \ulcorner \mathbf{a} \urcorner &= \mathbf{a} \\
\\
\text{SF Contexts:} & \ulcorner \cdot \urcorner &= \texttt{nil}[] \\
& \ulcorner \Psi, x : \mathbf{a} \urcorner &= \texttt{cons}[\ulcorner \Psi \urcorner, \mathbf{a}]
\end{array}$$

The translation of SF terms is directed by their contextual type $\Psi \vdash A$, because it needs the context to perform the translation of names to de Bruijn indices and the types to appropriately index the terms.

$$\begin{array}{lrl}
\text{SF Terms:} & \ulcorner \lambda x.M \urcorner_{\Psi \vdash \mathbf{a} \to A} &= \texttt{Lam}[\texttt{cons}[\ulcorner \Psi \urcorner, \mathbf{a}], \ulcorner A \urcorner] \ulcorner M \urcorner_{\Psi, \mathbf{a} \vdash A} \\
& \ulcorner \{M\} \urcorner_{\Psi \vdash \Box\, A} &= \texttt{Box}[\ulcorner \Psi \urcorner, \ulcorner A \urcorner] \ulcorner M \urcorner_{\cdot \vdash A} \\
& \ulcorner x \urcorner_{\Psi \vdash \mathbf{a}} &= \texttt{Var}[\ulcorner \Psi \urcorner, \mathbf{a}] \ulcorner x \urcorner^{v}_{\Psi} \\
& \ulcorner \mathbf{c}\, \overrightarrow{M} \urcorner_{\Psi \vdash \mathbf{a}} &= \texttt{C}[\ulcorner \Psi \urcorner, \ulcorner A \urcorner, \mathbf{a}](\mathbf{c}, \ulcorner \overrightarrow{M} \urcorner_{\Psi \vdash A \downarrow \mathbf{a}}) \\
& & \quad \text{with } \Sigma(\mathbf{c}) = A \\
& \ulcorner M[\sigma]^{\Phi}_{\Psi} \urcorner_{\Psi \vdash A} &= \texttt{apply_sub} \ulcorner M \urcorner_{\Phi \vdash A} \ulcorner \sigma \urcorner_{\Psi \vdash \Phi} \\
& \ulcorner \texttt{'u} \urcorner_{\Psi \vdash A} &= u \\
& \ulcorner \texttt{\#v} \urcorner_{\Psi \vdash \mathbf{a}} &= \texttt{Var}[\ulcorner \Psi \urcorner, \mathbf{a}] \ulcorner \texttt{v} \urcorner_{\Psi \vdash \mathbf{a}} \\
& \ulcorner N, \overrightarrow{M} \urcorner_{\Psi \vdash A \to B \downarrow \mathbf{a}} &= \texttt{Cons}[\ulcorner \Psi \urcorner, \texttt{arr}[\ulcorner A \urcorner, \ulcorner B \urcorner], \mathbf{a}] \\
& & \quad (\ulcorner N \urcorner_{\Psi \vdash A}, \ulcorner \overrightarrow{M} \urcorner_{\Psi \vdash B \downarrow \mathbf{a}}) \\
& \ulcorner \cdot \urcorner_{\Psi \vdash \mathbf{a} \downarrow \mathbf{a}} &= \texttt{Empty}[\ulcorner \psi \urcorner, \mathbf{a}, \mathbf{a}] \\
\text{Param. Vars:} & \ulcorner \texttt{x} \urcorner_{\Psi \vdash \mathbf{a}} &= x \\
& \ulcorner \texttt{\#v} \urcorner_{\Psi, y : \mathbf{a}' \vdash \mathbf{a}} &= \texttt{Pop}[\ulcorner \Psi \urcorner, \mathbf{a}, \mathbf{a}'] \ulcorner \texttt{v} \urcorner_{\Psi \vdash \mathbf{a}}
\end{array}$$

There are are three kinds of variables in the syntactic framework SF: bound variables, quoted variables and parameter variables. Each kind requires a different translation strategy. Bound variables are translated into de Bruijn indices where the numbers are encoded using the constructors `Top` and `Pop`. Quoted variables are simply translated into the Core-ML$^{\text{gadt}}$ variables they quote. And finally the parameter variables are translated into a `Var` constructor to indicate that the resulting expression is an SF variable, and the shifts (indicated by extra '`#`') are translated to applications of the constructor `Pop`.

Notice how substitutions are not part of the representation. They are translated to the eager application of `apply_sub`, an OCaml function that performs the substitution. Before we call `appy_sub` we translate the substitution. This amounts to generating the right shift for empty substitutions and otherwise recursively translating the terms and the substitution.

Translating variables requires computing the de Bruijn index with the appropriate type annotations.

We also need to translate SF patterns into Core-ML$^{\text{gadt}}$ expressions with the right structure. The special cases are:

– Variables are translated to de Bruijn indexes.

– Quoted variables simply translate to Core-ML$^{\text{gadt}}$ variables.

– Parameter variables translate to a pattern that matches only variables by specifying the `Var` constructor.

The translation of patterns follows the same line as the translation of terms, however, we do not use the indices of type variables in Core-ML$^{\text{gadt}}$ patterns. This is indicated by writing an underscore.

$$
\begin{array}{lrll}
\text{SF Patterns:} & \ulcorner \lambda x.R \urcorner^{\Gamma}_{\Psi \vdash \mathbf{a} \to A} & = \texttt{Lam}[\_,\_,\_] \ulcorner R \urcorner^{\Gamma}_{\Psi, \mathbf{a} \vdash A} & \\
& \ulcorner \{R\} \urcorner^{\Gamma}_{\Psi \vdash \Box\, A} & = \texttt{Box}[\_,\_] \ulcorner R \urcorner^{\Gamma}_{\cdot \vdash A} & \\
& \ulcorner x \urcorner^{\cdot}_{\Psi \vdash \mathbf{a}} & = \texttt{Var}[\_,\_] \ulcorner x \urcorner^{p}_{\Psi} & \\
& \ulcorner \mathbf{c}\ \overrightarrow{R} \urcorner^{\Gamma}_{\Psi \vdash \mathbf{a}} & = \texttt{C}[\_,\_,\_] \ulcorner \overrightarrow{R} \urcorner^{\Gamma}_{\Psi \vdash A \downarrow \mathbf{a}} & \text{with } \Sigma(\mathbf{c}) = A \\
& \ulcorner \texttt{,u} \urcorner^{u : \ulcorner \Psi \vdash A \urcorner}_{\Psi \vdash A} & = u & \\
& \ulcorner \texttt{\#x} \urcorner^{x : \ulcorner \Psi \vdash A \urcorner}_{\Psi \vdash \mathbf{a}} & = \texttt{Var}[\_,\_] x & \\
& \ulcorner \texttt{\#\#x} \urcorner^{x : \ulcorner \Psi, y : \_ \vdash A \urcorner}_{\Psi, y : \_ \vdash \mathbf{a}} & = \texttt{Var}[\_,\_](\texttt{Pop}[\_,\_,\_] x) & \\
& \ulcorner R, \overrightarrow{R'} \urcorner^{\Gamma, \Gamma'}_{\Psi \vdash A \to B \downarrow \mathbf{a}} & = \texttt{Cons}[\_,\_,\_,\_](\ulcorner R \urcorner^{\Gamma}_{\Psi \vdash A}, \ulcorner \overrightarrow{R'} \urcorner^{\Gamma'}_{\Psi \vdash B \downarrow \mathbf{a}}) & \\
& \ulcorner \cdot \urcorner^{\cdot}_{\Psi \vdash \mathbf{a} \downarrow \mathbf{a}} & = \texttt{Empty}[\_,\_] & \\
\text{SF Variables:} & \ulcorner x \urcorner^{p}_{\Psi, x : \mathbf{a}} & = \texttt{Top}[\_,\_] & \\
& \ulcorner y \urcorner^{p}_{\Psi, x : \mathbf{b}} & = \texttt{Pop}[\_,\_,\_] \ulcorner y \urcorner^{p}_{\Psi} &
\end{array}
$$

Our main translation of Core-ML to Core-ML$^{\text{gadt}}$ uses the following main operations:

$$
\begin{array}{rl}
\ulcorner \tau \urcorner, \ulcorner \Xi \urcorner, \ulcorner \Gamma \urcorner : & \text{Translate types, signatures and contexts.} \\
\ulcorner e \urcorner_{\Gamma \vdash \tau} : & \text{Type directed translation of expressions.} \\
\ulcorner pat \urcorner^{\Gamma'}_{\Gamma \vdash \tau} : & \text{Translates patterns and outputs } \Gamma' \text{ the} \\
& \text{context of the bound variables.}
\end{array}
$$

The translation of Core-ML expressions into Core-ML$^{\text{gadt}}$ directly follows the structure of programs in Core-ML and is type directed to fill in the required

types for the Core-ML$^{\text{gadt}}$ representation. The translation is as follows:

$$
\begin{aligned}
\ulcorner x \urcorner_{\Gamma \vdash \tau} &= x \\
\ulcorner C\ \overrightarrow{e} \urcorner_{\Gamma \vdash D} &= C[\,]\ulcorner \overrightarrow{e} \urcorner_{\Gamma \vdash \overrightarrow{\tau}} \text{ with } \Xi(C) = \overrightarrow{\tau} \to D \\
\ulcorner \mathtt{fun}\, f(x) = e \urcorner_{\Gamma \vdash \tau_1 \to \tau_2} &= \mathtt{fix}\, f : \ulcorner \tau_1 \to \tau_2 \urcorner = \lambda x\, .\, \ulcorner e \urcorner_{\Gamma, x:\tau_1 \vdash \tau_2} \\
\ulcorner i\, e \urcorner_{\Gamma \vdash \tau} &= \ulcorner i \urcorner_{\Gamma \vdash \tau_1 \to \tau} \ulcorner e \urcorner_{\Gamma \vdash \tau_1} \\
&\quad \text{with } \Gamma \vdash i \Rightarrow \tau_1 \to \tau \\
\ulcorner \mathtt{let}\, x = i\ \mathtt{in}\, e \urcorner_{\Gamma \vdash \tau} &= \mathtt{let}\, x = \ulcorner i \urcorner_{\Gamma \vdash \tau_1}\ \mathtt{in}\ \ulcorner e \urcorner_{\Gamma, x:\tau_1 \vdash \tau} \\
&\quad \text{with } \Gamma \vdash i \Rightarrow \tau_1 \\
\ulcorner \mathtt{match}\, i\, \mathtt{with}\ \overrightarrow{b} \urcorner_{\Gamma \vdash \tau} &= \mathtt{match} \ulcorner i \urcorner_{\Gamma \vdash \tau_1}\, \mathtt{with}\, \ulcorner \overrightarrow{b} \urcorner_{\Gamma \vdash \tau_1 \to \tau} \\
&\quad \text{with } \Gamma \vdash i \Rightarrow \tau_1 \\
\ulcorner e_1, \ldots, e_n \urcorner_{\Gamma \vdash \overrightarrow{\tau}} &= \ulcorner e_1 \urcorner_{\Gamma \vdash \tau_1}, \ldots, \ulcorner e_n \urcorner_{\Gamma \vdash \tau_n} \\
\ulcorner [\hat{\Psi} \vdash M] \urcorner_{\Gamma \vdash [\Psi \vdash A]} &= \ulcorner M \urcorner_{\Psi \vdash A} \\
\ulcorner \mathtt{cmatch}\, i\, \mathtt{with}\ \overrightarrow{c} \urcorner_{\Gamma \vdash \tau} &= \mathtt{match} \ulcorner i \urcorner_{\Gamma \vdash [\Psi \vdash A]}\, \mathtt{with}\, \ulcorner \overrightarrow{c} \urcorner_{\Gamma \vdash [\Psi \vdash A] \to \tau} \\
&\quad \text{with } \Gamma \vdash i \Rightarrow [\Psi \vdash A]
\end{aligned}
$$

Translating branches and patterns:

Branch: $\ulcorner pat \mapsto e \urcorner_{\Gamma \vdash \tau_1 \to \tau_2} = \ulcorner pat \urcorner^{\Gamma'}_{\Gamma \vdash \tau_1} \mapsto \ulcorner e \urcorner_{\Gamma, \Gamma' \vdash \tau_2}$

Patterns:

$$
\begin{aligned}
\ulcorner x \urcorner^{x:\tau}_{\Gamma \vdash \tau} &= x \\
\ulcorner C\ \overrightarrow{pat} \urcorner^{\Gamma'}_{\Gamma \vdash D} &= C[]\ulcorner \overrightarrow{pat} \urcorner^{\Gamma'}_{\Gamma \vdash \overrightarrow{\tau}} \qquad \text{with } \Xi(C) = \overrightarrow{\tau} \to D
\end{aligned}
$$

Finally, we define the translation of branches for $\mathtt{cmatch}\, i\, \mathtt{with}\ \overrightarrow{c}$. Note how we use the context generated from the pattern to translate the body of the branch.

$$\ulcorner [\Psi \vdash R] \mapsto e \urcorner_{\Gamma \vdash [\Psi \vdash A] \to \tau} = \ulcorner R \urcorner^{\Gamma'}_{\Psi \vdash A} \mapsto \ulcorner e \urcorner_{\Gamma, \Gamma' \vdash \tau}$$

Finally we show that the translation from Core-ML with contextual types into Core-ML$^{\text{gadt}}$ preserves types.

Thm. 1 (Main)

1. *If* $\Gamma \vdash e \Leftarrow \tau$ *then* $\cdot; \ulcorner \Gamma \urcorner \vdash \ulcorner e \urcorner_{\Gamma \vdash \tau} : \ulcorner \tau \urcorner$.
2. *If* $\Gamma \vdash i \Rightarrow \tau$ *then* $\cdot; \ulcorner \Gamma \urcorner \vdash \ulcorner i \urcorner_{\Gamma \vdash \tau} : \ulcorner \tau \urcorner$.

Our result relies on several lemmas that deal with the other judgments and context lookups:

Lemma 1 (Ambient Context) *If* $\Gamma(u) = [\Psi \vdash a]$ *then* $\ulcorner \Gamma \urcorner(u) = \mathtt{tm}[\ulcorner \Psi \urcorner, a]$.

Lemma 2 (Terms)

1. *If* $\Gamma; \Psi \vdash M : A$ *then* $\cdot; \ulcorner \Gamma \urcorner \vdash \ulcorner M \urcorner_{\Psi \vdash A} : \ulcorner \Psi \vdash A \urcorner$.
2. *If* $\Gamma; \Psi \vdash \sigma : \Phi$ *then* $\cdot; \ulcorner \Gamma \urcorner \vdash \ulcorner \sigma \urcorner_{\Psi \vdash \Phi} : \ulcorner \Psi \vdash \Phi \urcorner$

Lemma 3 (Pat.) *If* $\vdash pat : \tau \downarrow \Gamma$ *then* $\cdot \vdash \ulcorner pat \urcorner^{\Gamma}_{\Psi \vdash A} : \ulcorner \tau \urcorner \downarrow \Gamma$.

Lemma 4 (Ctx. Pat.) *If* $\Psi \vdash R : A \downarrow \Gamma$ *then* $\cdot \vdash \ulcorner R \urcorner^{\Gamma}_{\Psi \vdash A} : \ulcorner \Psi \vdash A \urcorner \downarrow \Gamma$.

Given our set-up, the proofs are straightforward by induction on the typing derivation.

9 A Proof of Concept Implementation

In this section, we describe the implementation[3] of Babybel which uses the ideas from previous sections. One major difference is that Babybel translates OCaml programs that use syntax extensions for contextual SF types and terms and translates them into pure OCaml with GADTs. In fact, even our input OCaml programs may use GADTs to for example describe context relations on SF contexts (see also our examples from Sec.2).

The presence of GADTs in our source language also means that we can specify precise types for functions where we can quantify over contexts. Let's revisit some of the types of the programs that we wrote earlier in Sec.2:

- `rewrite: γ. [γ ⊢ tm]→[γ ⊢ tm]`: In this type we implicitly quantify over all contexts `g` and then we take a potentially open term and return another term in the same context. These constraints imposed in the types are due to being able to index types with types thanks to GADTs.
- `get_path: γ. [γ,x:tm ⊢ tm]→path`: In this case we quantify over all contexts, but the input of the function is some term in a non-empty context.
- `conv: γ δ.(γ, δ) rel→[γ ⊢ tm]→[δ ⊢ ctm]`:
  This final example shows that we can also use the contexts to index regular OCaml GADTs. In this function we are translating between terms in different representations.To be able to translate between these different context representations, it is necessary to establish a relation between these contexts. So we need to define a special OCaml type (i.e.:`rel`) that relates variable to variable in each contexts.

By embedding the SF in OCaml using contextual types, we can combine and use the impure features of OCaml. Our example, in Section 2.2 takes advantage of them in our implementation of backtracking with exceptions. Additionally, performing I/O or using references works seamlessly in the prototype.

The presence of GADTs in our target language also makes the actual implementation of Babybel simpler than the theoretical description, as we take advantage of OCaml's built-in type reconstruction. In addition to GADTs, our implementation depends on several OCaml extensions. We use Attributes from Section 7.18 of the reference manual [10] and strings to embed the specification of our signature. We use quoted strings from Section 7.20 to implement the boxes for terms ($\{\!|\ldots|\!\}$) and patterns ($\{\!|\ldots|\!\}_p$). All these appear as annotations in the internal Abstract Syntax Tree in the compiler implementation. To perform the translation (based on Section 8) we define a PPX rewriter as discussed in Section 23.1 of the OCaml manual. In our rewriter, we implement a parser for the framework SF and translate all the annotations using our embedding.

10 Related Work

Babybel and the syntactic framework SF are derived from ideas that originated in proof checkers based on the logical framework LF such as the Twelf system [14].

[3] Available at `www.github.com/fferreira/babybel/`

In the same category are the proof and programming languages Delphin [19] and Beluga [16] that offer a computational language on top of the LF. In many ways, the work that we present in this paper and forms the foundation of Babybel are a distillation of Beluga's ideas applied to a mainstream programming language. As a consequence, we have shown that we can get some of the benefits from Beluga at a much lower cost, since we do not have to build a stand-alone system or extend the compiler of an existing language to support contexts, contextual types and objects.

Our approach of embedding an LF specification language into a host language is in spirit related to the systems Abella [8] and Hybrid [7] that use a two-level approach. In these systems we embed the specification language (typically hereditary harrop formulas) in first-order logic (or a variant of it). While our approach is similar in spirit, we focus on embedding SF specifications into a programming language instead of embedding it into a proof theory. Moreover, our embedding is type preserving by construction.

There are also many approaches and tools that specifically add support for writing programs that manipulate syntax with binders – even if they do not necessarily use HOAS. FreshML [22] and Cαml [20] extend OCaml's data types with the ideas of names and binders from nominal logic [18]. In these system, name generation is an effect, and if the user is not careful variables may extrude their scopes. Purity can be enforced by adding a proof system with a decision procedure that statically guarantees that no variable escapes its scope [21]. This adds some complexity to the system. We feel that Babybel's contextual types offer a simpler formalism to deal with bound variables. On the other hand, Babybel's approach does not try to model variables that do not have lexical scope, like top-level definitions. Another related language is Romeo [23] that uses ideas from Pure FreshML to represent binders. Where our system statically catches variables escaping their scope, the Romeo system either throws a run time exception or uses an SMT solver to prove the absence of scoping issues. The Hobbits system for Haskell [25] is implemented in a similar way to ours, using quasi-quoting but they use a different formalism based on the concepts of names and freshness. Last but not least, approaches based on parametric HOAS (PHOAS) [5] also model binding in the object language by re-using the function space of the meta-language. In particular, Washburn and Weirich [24] propose a library that uses catamorphisms to compute over a parametric HOAS representation. This is a powerful approach but requires a different way of implementing recursive functions. A fundamental difference between this line of work and ours, is that in PHOAS functions are extensional, i.e. they are black box functions, while our approach introduces a distinction between an intensional and extensional function space. The intensional function space from SF allows us to model binding and supports pattern matching. The extentional function space allows us to write recursive functions.

11 Conclusion and Future Work

In this work, we describe the syntactic framework SF (a simply typed variant of the logical framework LF with the necessity modality from S4) and explain the embedding of SF into a functional programming language using contextual types. This gives programmers the ability to write programs that manipulate abstract syntax trees with binders while knowing at type checking time that no variables extrude their scope. We also show how to translate the extended language back into a first order representation. For this, we use de Bruijn indices and GADTs to implement the SF in Core-ML$^{\text{gadt}}$. Important characteristics of the embedding are that it preserves the phase separation, making types (and thus contexts) erasable at run-time. This allows pattern matching to remain first-order and thus it is possible to compile with the traditional algorithms.

Finally, we describe Babybel an implementation of these ideas that embeds SF in OCaml using contextual types. The embedding is flexible enough that we can take advantage of the more powerful type system in OCaml to make the extension more powerful. We use GADTs in our examples to express more powerful invariants (e.g. that the translation preserves the context).

In the future, we plan to implement our approach also in other languages. In particular, it would be natural to implement our approach in Haskell. We do not expect that the the type system extensions to GHC pose any challenging issues. Finally, it would be interesting to extend our approach to type systems with dependent types (e.g. Coq or Agda) where we can reason about the programs we write. This extension would require extending SF with theorems about substitutions (e.g. proving that applying an identity substitution does not change a term).

References

1. Augustsson, L.: Compiling pattern matching. In: Jouannaud, J.P. (ed.) Functional Programming Languages and Computer Architecture (FPCA'85). Lecture Notes in Computer Science (LNCS), vol. 201, pp. 368–381. Springer (1985)
2. Benton, N., Hur, C.K., Kennedy, A.J.., McBride, C.: Strongly typed term representations in Coq. Journal of Automated Reasoning 49(2), 141–159 (2012)
3. Cave, A., Pientka, B.: Programming with binders and indexed data-types. In: 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12). pp. 413–424. ACM Press (2012)
4. Cheney, J., Hinze, R.: First-class phantom types. Tech. Rep. CUCIS TR2003-1901, Cornell University (2003)
5. Chlipala, A.J.: Parametric higher-order abstract syntax for mechanized semantics. In: Hook, J., Thiemann, P. (eds.) 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08). pp. 143–156. ACM (2008)
6. Davies, R., Pfenning, F.: A modal analysis of staged computation. Journal of the ACM 48(3), 555–604 (2001)
7. Felty, A., Momigliano, A.: Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. Journal of Automated Reasoning 48(1), 43–105 (2012)

8. Gacek, A., Miller, D., Nadathur, G.: A two-level logic approach to reasoning about computations. Journal of Automated Reasoning 49(2), 241–273 (2012)
9. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM 40(1), 143–184 (January 1993)
10. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml System Release 4.03 – Documentation and user's manual. Institut National de Recherche en Informatique et en Automatique (2016)
11. Miller, D., Palmidessi, C.: Foundational aspects of syntax. ACM Comput. Surv. 31(3es) (Sep 1999)
12. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic 9(3), 1–49 (2008)
13. Pfenning, F., Davies, R.: A judgmental reconstruction of modal logic. Mathematical Structures in Computer Science 11(4), 511–540 (2001)
14. Pfenning, F., Schürmann, C.: System description: Twelf — a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) 16th International Conference on Automated Deduction (CADE-16). pp. 202–206. Lecture Notes in Artificial Intelligence (LNAI 1632), Springer (1999)
15. Pientka, B.: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08). pp. 371–382. ACM Press (2008)
16. Pientka, B., Cave, A.: Inductive Beluga:Programming Proofs (System Description). In: Felty, A.P., Middeldorp, A. (eds.) 25th International Conference on Automated Deduction (CADE-25). pp. 272–281. Lecture Notes in Computer Science (LNCS 9195), Springer (2015)
17. Pientka, B., Pfenning, F.: Optimizing higher-order pattern unification. In: Baader, F. (ed.) 19th International Conference on Automated Deduction (CADE-19). pp. 473–487. Lecture Notes in Artificial Intelligence (LNAI) 2741, Springer-Verlag (2003)
18. Pitts, A.: Nominal logic, a first order theory of names and binding. Information and Computation 186(2), 165–193 (Nov 2003)
19. Poswolsky, A., Schürmann, C.: System description: Delphin—a functional programming language for deductive systems. In: International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08). Electronic Notes in Theoretical Computer Science (ENTCS), vol. 228, pp. 135–141. Elsevier (2009)
20. Pottier, F.: An overview of Cαml. Electronic Notes in Theoretical Computer Science 148(2), 27 – 52 (2006), proceedings of the ACM-SIGPLAN Workshop on ML (ML'05)
21. Pottier, F.: Static name control for FreshML. In: 22nd IEEE Symposium on Logic in Computer Science (LICS'07). pp. 356–365. IEEE Computer Society (Jul 2007)
22. Shinwell, M.R., Pitts, A.M., Gabbay, M.J.: FreshML: programming with binders made simple. In: 8th International Conference on Functional Programming (ICFP'03). pp. 263–274. ACM Press (2003)
23. Stansifer, P., Wand, M.: Romeo: A system for more flexible binding-safe programming. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 53–65. ICFP '14 (2014)
24. Washburn, G., Weirich, S.: Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. Journal of Functional Programming 18(01), 87–140 (2008)

25. Westbrook, E., Frisby, N., Brauner, P.: Hobbits for Haskell: a library for higher-order encodings in functional programming languages. In: 4th ACM Symposium on Haskell (Haskell'11). pp. 35–46. ACM (2011)
26. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03). pp. 224–235. ACM Press (2003)