An overview of alphaCaml

François Pottier

September 2005

INRIA

Introduction

A specification language

Implementation techniques

Translating specifications

Conclusion

Motivation

Our programming languages do not support *abstract syntax with binders* in a satisfactory way.

Hand-coding the operations that deal with lexical scope (capture-avoiding substitution, etc.) is tedious and error-prone.

How about a more *declarative*, *robust*, *automated* approach?

— cf. Shinwell's Fresh O'Caml, Cheney's FreshLib.

Three facets

Let's distinguish three facets of the problem:

- a *specification language*,
- an *implementation technique*,
- an *automated translation* of the former to the latter.

In this talk, I emphasize the first aspect.

Introduction

A specification language

Implementation techniques

Translating specifications

Conclusion

Prior art

There have been a few proposals to enrich algebraic specification languages with *names* and *abstractions*.

An abstraction usually takes the form $\langle a \rangle e$, or $\langle a_1, \ldots, a_n \rangle e$, or, as in Fresh Objective Caml, $\langle e_1 \rangle e_2$.

Abstraction is always *binary:* the names (or *atoms*) a that appear on the left-hand side are bound, and their scope is the expression e that appears on the right-hand side.

Example: pure λ-calculus

Pure λ-calculus:

$$M := a \mid M\,M \mid \lambda a.M$$

is modelled in Fresh Objective Caml as follows:

bindable_type *var*

type *term* =
  | *EVar* of *var*
  | *EApp* of *term* ∗ *term*
  | *ELam* of ⟨*var*⟩*term*

A more delicate example

Let's add *simultaneous* definitions:

$$M ::= \ldots \mid \textbf{let } a_1 = M_1 \textbf{ and } \ldots \textbf{ and } a_n = M_n \textbf{ in } M$$

The atoms a_i are bound, so they must lie *within* the abstraction's left-hand side. The terms M_i are outside the abstraction's lexical scope, so they must lie *outside* of the abstraction:

$$\textbf{type } term =$$
$$\mid \; ...$$
$$\mid \; ELet \textbf{ of } term \; list * \langle var \; list \rangle term$$

Another delicate example

Simultaneous *recursive* definitions pose a similar problem:

$$M ::= \ldots \mid \textbf{letrec}\ a_1 = M_1\ \textbf{and}\ \ldots\ \textbf{and}\ a_n = M_n\ \textbf{in}\ M$$

The terms M_i are now inside the abstraction's lexical scope, so they must lie within the abstraction's *right-hand* side:

$$\begin{array}{l} \textbf{type}\ \mathit{term} = \\ \quad \mid\ \ldots \\ \quad \mid\ \mathit{ELetRec}\ \textbf{of}\ \langle \mathit{var}\ \mathit{list} \rangle (\mathit{term}\ \mathit{list} * \mathit{term}) \end{array}$$

The problem

The root of the problem is the assumption that *lexical* and *physical* structure should coincide.

A solution

Within an abstraction, alphaCaml distinguishes three basic components: *binding occurrences* of names, expressions that lie *within* the abstraction's lexical scope, and expressions that lie *outside* the scope.

These components are assembled using sums and products, giving rise to a syntactic category of so-called *patterns*. Abstraction becomes *unary* and holds a pattern.

$$t ::= \mathsf{unit} \mid t \times t \mid t + t \mid \mathsf{atom} \mid \langle u \rangle \qquad \textit{Expression types}$$
$$u ::= \mathsf{unit} \mid u \times u \mid u + u \mid \mathsf{atom} \mid \mathsf{inner}\ t \mid \mathsf{outer}\ t \qquad \textit{Pattern types}$$

Back to pure λ-calculus

Pure λ-calculus is modelled in alphaCaml as follows:

sort *var*

type *term* =
  | *EVar* of atom *var*
  | *EApp* of *term* ∗ *term*
  | *ELam* of ⟨*lamp*⟩

type *lamp* binds *var* =
  atom *var* ∗ inner *term*

A second look at simultaneous definitions

Simultaneous definitions are modelled without difficulty:

type *term* =
  | ...
  | *ELet* of ⟨*letp*⟩

type *letp* binds *var* =
  binding *list* ∗ inner *term*

type *binding* binds *var* =
  atom *var* ∗ outer *term*

More advanced examples

Abstract syntax for patterns in an Objective Caml-like programming language could be declared like this:

```
type pattern binds var =
  | PWildcard
  | PVar of atom var
  | PRecord of pattern StringMap.t
  | PInjection of [ constructor ] * pattern list
  | PAnd of pattern * pattern
  | POr of pattern * pattern
```

Introduction

A specification language

Implementation techniques

Translating specifications

Conclusion

Three known techniques

1. *de Bruijn* indices. Require *shifting*, which is fragile. No freshening. Generic equality and hashing functions respect α-equivalence.
2. *Atoms*. Require *freshening* upon opening abstractions. No shifting. Require custom equality and hashing functions.
3. *Pollack mix:* free names as atoms and bound names as indices. Analogous to 2, except generic equality and hashing respect α-equivalence.

alphaCaml follows 2.

Some more details

Atoms are represented as pairs of an integer and a string. The latter is used only as a hint for display.

Sets of atoms and renamings are encoded as Patricia trees.

Renamings are *suspended* and *composed* at abstractions, which allows linear-time term traversals.

Even though the fresh atom generator has state, *closed* terms can safely be marshalled to disk.

Introduction

A specification language

Implementation techniques

Translating specifications

Conclusion

Types

The specification of pure λ-calculus is translated down to Objective Caml as follows. Atoms and abstractions are *abstract*.

```
type var = Var.Atom.t

type term =
  | EVar of var
  | EApp of term * term
  | ELam of opaque_lamp

and lamp =
    var * term

and opaque_lamp
```

Code

Opening an abstraction automatically *freshens* its bound atoms.

$$\textbf{val}\ open\_lamp : opaque\_lamp \rightarrow lamp$$
$$\textbf{val}\ create\_lamp : lamp \rightarrow opaque\_lamp$$

This enforces Barendregt's informal convention.

More boilerplate is *generated* for computing sets of free or bound atoms, applying renamings, helping clients succinctly define transformations (such as capture-avoiding substitution), etc.

Introduction

A specification language

Implementation techniques

Translating specifications

Conclusion

Status

alphaCaml is *available*. There are very few known users so far.

The distribution comes with *two demos*:

- a naïve typechecker and evaluator for $F_{\leq}$
- a naïve evaluator for a calculus of mixins (Hirschowitz *et al.*)

These limited experiments are encouraging.

Limitations

One *must* go through *open* functions to examine abstractions. *Deep pattern matching* is impossible.

Clients can write *meaningless* code, such as a function that pretends to collect the bound atoms in an expression.

Towards alpha-(your-favorite-prover-here)?

How about translating a specification language like alphaCaml's into *theorems* (recursion and induction principles) and *proofs?*

— cf. Pitts, Urban and Tasson, Norrish...