Merwyn RFC: Alternatives

Peter Jin

September 11, 2019

Abstract

This note describes a proposed implementation of *alternatives*, via the `alt` syntax, to provide static, lexically scoped, monomorphic multimethods in Merwyn.

1 Example

```
alt zero in
alt add' in
(* ... *)
let alt zero: Int = 0n in
let alt zero: Flp = 0.0f in
let alt add': [Int, Int] -> Int = \x, y -> x + y
 in
let alt add': [Flp, Flp] -> Flp = \x, y -> x + y
 in
(* ... *)
```

2 Introduction

In a typical ML-like implementation of λ-calculus with `let`-binding, variable identifiers (such as `x` or `foo`) can be uniquely rewritten based on their lexical binding order, so that the rewritten identifiers correspond one-to-one with bound definitions. One desirable language extension is some form of overloading or multimethods, where identifiers may be allowed to refer to one and only one of several differently typed implementations that are simultaneously in scope. When a second form of binding is added to enable a form of overloading or multimethods, then identifiers may potentially map to a set of definitions. The use of such an identifier is a choice among the set of definitions, which generally requires knowledge of types. Thus, some identifiers may require type inference in order to resolve to a particular implementation with a principal type.

In our proposed implementation, identifiers are not directly typed via the *type context* Γ. Rather, only definitions, which correspond to a unique binding of an expression to an identifier, are directly typed. The indirection between identifiers and definitions is represented by a *binding context*, which we denote with B, mirroring the more familiar type context. In our proposed implementation, the typing context strictly maps only definitions to types; the binding context maps identifiers to either definitions or a set of *alternative* definitions.

3 Static Semantics

3.1 Syntax

Notation:

$\texttt{x} \in \texttt{Id}$	(identifiers)
$x \in \texttt{Def}$	(definitions)
$a \in 2^{\texttt{Def}}$	(alternative definitions)
$e \in \texttt{Exp}$	(expressions)
$\tau \in \texttt{Ty}$	(types)
$\Gamma : \texttt{Def} \to \texttt{Ty}$	(type context)
$\mathrm{B} : \texttt{Id} \to \texttt{Def} + 2^{\texttt{Def}}$	(binding context)

Expressions:

$e ::= c$	(literal constant)
$\mid x$	(variable)
$\mid \lambda x.\ e$	(function abstraction)
$\mid e[e']$	(function application)
$\mid \texttt{let x = } e \texttt{ in } e'$	(let-binding)
$\mid \texttt{alt x in } e$	(alt-declaration)
$\mid \texttt{let alt x: } \tau \texttt{ = } e \texttt{ in } e'$	(alt-binding)

Types:

$\tau ::= \alpha$	(type variable)
$\mid \beta$	(primitive type)
$\mid \tau \to \tau$	(function type)

3.2 Static Semantics

3.2.1 Typing rules

$$\frac{}{\Gamma; \mathrm{B} \vdash c : \beta}\ (\textsc{Lit})$$

$$\frac{\Gamma, x : \tau; \mathrm{B}, \mathtt{x} \triangleq x \vdash e : \tau'}{\Gamma; \mathrm{B} \vdash \lambda x.\ e : \tau \to \tau'}\ (\textsc{Abs})$$

$$\frac{\Gamma; \mathrm{B} \vdash e : \tau' \to \tau, e' : \tau'}{\Gamma; \mathrm{B} \vdash e[e'] : \tau}\ (\textsc{App})$$

$$\frac{}{\Gamma, x : \tau; \mathrm{B}, \mathtt{x} \triangleq x \vdash \mathtt{x} : \tau}\ (\textsc{Ident})$$

$$\frac{\Gamma; \mathrm{B}, \mathtt{x} \triangleq a \vdash \mathtt{x} : \tau \text{ where } x \in a \wedge \tau = \Gamma[x]}{\Gamma - (a \backslash x); \mathrm{B}, \mathtt{x} \triangleq a \vdash \mathtt{x} : \tau}\ (\textsc{Ident-Alt})$$

$$\frac{\Gamma; \mathrm{B} \vdash e : \tau \qquad \Gamma, x : \tau; \mathrm{B}, \mathtt{x} \triangleq x \vdash e' : \tau' \text{ where } x \text{ fresh in } \Gamma}{\Gamma; \mathrm{B} \vdash \mathtt{let\ x\ =}\ e\ \mathtt{in}\ e' : \tau'}\ (\textsc{Let})$$

$$\frac{\Gamma; \mathrm{B}, \mathtt{x} \triangleq \emptyset \vdash e : \tau}{\Gamma; \mathrm{B} \vdash \mathtt{alt\ x\ in}\ e : \tau}\ (\textsc{Alt})$$

$$\frac{\begin{array}{l}\Gamma, x : \tau; \mathrm{B}, \mathtt{x} \triangleq a \vdash e' : \tau' \\ \text{where } x \in a \text{ fresh in } \Gamma \\ \Gamma; \mathrm{B}, \mathtt{x} \triangleq a \backslash x \vdash e : \tau \qquad \wedge\ \forall x_a \in a \backslash x.\ \tau \neq \Gamma[x_a]\end{array}}{\Gamma; \mathrm{B}, \mathtt{x} \triangleq a \backslash x \vdash \mathtt{let\ alt\ x:}\ \tau\ \mathtt{=}\ e\ \mathtt{in}\ e' : \tau'}\ (\textsc{Let-Alt})$$

4 Limitations

One of the obvious limitations of our proposed implementation of alternatives in Merwyn is its restriction to a monomorphic fragment of the language. We have not defined how alternatives as proposed may be extended with, e.g., parametric polymorphism or other approaches to generics. However, there are substantial benefits to monomorphism-by-default [1]. In numerical or mathematical programs which are likely to be written using Merwyn, monomorphism-by-default might make even more sense. So, limiting ourselves to the monomorphic fragment of the language may be an acceptable tradeoff, at least for the moment, and particularly with respect to our design of alternatives.

A corollary to the monomorphic limitation of alternatives is that lexical, monomorphic alternatives can hinder *compositionality*.[1] The current design of alternatives does not have a satisfactory answer to the question of compositionality, other than to say that an eventual design for generics will need to address this.

[1] Thanks to Jamie Brandon pointing this out.

5 Related Work

Some earlier approaches to overloading via type inference involve computing least common generalizers or anti-unification [2, 3]. This complication arises because the interaction between identifier overloading and parametric polymorphism, specifically type schemes, yields (quantified) constrained types. Consequently, resolving the overloaded identifier necessitates resolvable solutions to type constraints, which is nontrivial. In our implementation, anti-unification of monomorphic alternative types is essentially trivial.

Our design of alternatives involves first declaring the overloadable identifier before defining alterntive definitions, which bears perhaps a more than passing resemblance to type classes [4]. However we have not further explored the connection between the two.

References

[1] Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. Let Should not be Generalised. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2010.

[2] Geoffrey S Smith. Principal Type Schemes for Functional Programs with Overloading and Subtyping. *Science of Computer Programming*, 23(2-3): 197–226, 1994.

[3] Carlos Camarão and Lucília Figueiredo. Type Inference for Overloading without Restrictions, Declarations or Annotations. In *International Symposium on Functional and Logic Programming*, 1999.

[4] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1989.