Merwyn RFC: Alternatives

Peter Jin

October 12, 2019

Abstract

This note describes a proposed implementation of *alternatives*, via the `alt` syntax, to provide static, lexically scoped, monomorphic overloading 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 functional language based on 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, analogously to α-renaming in the formal λ-calculus. 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 environment* 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 environment. 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

In this section we provide a semantics for our design of alternatives by extending a simply-typed λ-calculus.

3.1 Language

Notation:

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

Expressions:

$$\begin{array}{rll}
e ::= & c & \text{(literal constant)} \\
& \mid \mathtt{x} & \text{(identifier)} \\
& \mid x & \text{(definition)} \\
& \mid \lambda \mathtt{x}.\ e & \text{(function abstraction)} \\
& \mid e[e'] & \text{(function application)} \\
& \mid \mathtt{let\ x\ =}\ e\ \mathtt{in}\ e' & \text{(let-binding)} \\
& \mid \mathtt{alt\ x\ in}\ e & \text{(alt-declaration)} \\
& \mid \mathtt{let\ alt\ x:}\ \tau\ \mathtt{=}\ e\ \mathtt{in}\ e' & \text{(alt-binding)}
\end{array}$$

Types:

$$\begin{array}{rll}
\tau ::= & \alpha & \text{(type variable)} \\
& \mid \beta & \text{(primitive type)} \\
& \mid \tau \to \tau & \text{(function type)}
\end{array}$$

3.2 Typing rules

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

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

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

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

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

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

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

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

The rules Lit, Abs, App, Id, and Let are analogous to the usual typing rules in e.g. a simply typed λ-calculus. Our versions of these rules are somewhat elaborate as they involve some indirection between identifiers and definitions via the binding context, but our rules are otherwise as expected.

The rule Id-Alt may appear superfluous as both the premise and conclusion yield the typing $\mathtt{x} : \tau$, but its purpose is quite necessary: Id-Alt enables the resolution of an identifier bound to a set of alternatives into a single definition within the set. In other words, Id-Alt allows the rewriting of identifiers into definitions; this latter interpretation based on rewriting is what the compiler implements.

The rule Alt seeds an alternative-bound identifier with the empty set of definitions $\emptyset$.

The rule Let-Alt extends an identifier already bound to a set of alternatives with a new definition of type τ, so long as no existing definition in the alternatives set also has type τ. Both rules Let and Let-Alt assume a facility for allocating fresh definitions. Additionally, the three binding rules Let, Alt, and Let-Alt are all scoped.

4 Related Work

Some earlier approaches to overloading via type inference involve computing least common generalizers or anti-unification [1, 2]. 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 design, anti-unification of monomorphic alternative types is essentially trivial.

Our design of alternatives involves first declaring the overloadable identifier before extending it with alternative definitions, which bears perhaps a more than passing resemblance to type classes [3]. However, although instances of type classes are not restricted, the definition of the type class must be global, i.e. defined at the top-level [4]. Other earlier approaches to overloading also limit the overloadable bindings to the program top-level [5]. Our design admits lexically scoped overloading that is not limited to top-level bindings.

5 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 [6]. 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.* 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.

Acknowledgments

Thanks to Jamie Brandon for an interesting discussion.

References

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

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

[3] 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.

[4] Andres Löh and Ralf Hinze. Open Data Types and Open Functions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2006.

[5] Martin Odersky, Philip Wadler, and Martin Wehr. A Second Look at Overloading. In *Proceedings of the FPCA '95 Conference on Functional Programming Languages and Computer Architecture*, 1995.

[6] 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.