Type systems for programming languages

Didier Rémy

Academic year 2020-2021
Version of October 4, 2022
# Contents

1 Introduction ........................................... 9
  1.1 Overview of the course .............................. 9
  1.2 Requirements .................................... 11
  1.3 About Functional Programming ...................... 11
  1.4 About Types .................................... 11
  1.5 Acknowledgment .................................. 13

2 The untyped \( \lambda \)-calculus ....................... 15
  2.1 Syntax ......................................... 15
  2.2 Semantics ..................................... 17
    2.2.1 Strong v.s. weak reduction strategies .......... 17
    2.2.2 Call-by-value semantics ...................... 18
  2.3 Answers to exercises ................................ 20

3 Simply-typed lambda-calculus ......................... 23
  3.1 Syntax ......................................... 23
  3.2 Dynamic semantics ................................ 23
  3.3 Type system .................................... 24
  3.4 Type soundness .................................. 27
    3.4.1 Proof of subject reduction ................... 28
    3.4.2 Proof of progress ............................ 30
  3.5 Simple extensions ................................ 32
    3.5.1 Unit ....................................... 32
    3.5.2 Boolean .................................... 32
    3.5.3 Pairs ....................................... 33
    3.5.4 Sums ....................................... 34
    3.5.5 Modularity of extensions ...................... 34
    3.5.6 Recursive functions .......................... 35
    3.5.7 A derived construct: let-bindings ............ 35
  3.6 Exceptions ...................................... 37
    3.6.1 Semantics .................................. 37
**CONTENTS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6.2 Typing rules</td>
<td>38</td>
</tr>
<tr>
<td>3.6.3 Variations</td>
<td>39</td>
</tr>
<tr>
<td>3.7 References</td>
<td>41</td>
</tr>
<tr>
<td>3.7.1 Language definition</td>
<td>41</td>
</tr>
<tr>
<td>3.7.2 Type soundness</td>
<td>43</td>
</tr>
<tr>
<td>3.7.3 Tracing effects with a monad</td>
<td>44</td>
</tr>
<tr>
<td>3.7.4 Memory deallocation</td>
<td>45</td>
</tr>
<tr>
<td>3.8 Ommitted proofs and answers to exercises</td>
<td>46</td>
</tr>
<tr>
<td><strong>4 Polymorphism and System $\mathcal{F}$</strong></td>
<td>51</td>
</tr>
<tr>
<td>4.1 Polymorphism</td>
<td>51</td>
</tr>
<tr>
<td>4.2 Polymorphic $\lambda$-calculus</td>
<td>53</td>
</tr>
<tr>
<td>4.2.1 Types and typing rules</td>
<td>53</td>
</tr>
<tr>
<td>4.2.2 Semantics</td>
<td>54</td>
</tr>
<tr>
<td>4.2.3 Extended System $\mathcal{F}$ with datatypes</td>
<td>56</td>
</tr>
<tr>
<td>4.3 Type soundness</td>
<td>60</td>
</tr>
<tr>
<td>4.4 Type erasing semantics</td>
<td>64</td>
</tr>
<tr>
<td>4.4.1 Implicitly-typed System $\mathcal{F}$</td>
<td>64</td>
</tr>
<tr>
<td>4.4.2 Type instance</td>
<td>67</td>
</tr>
<tr>
<td>4.4.3 Type containment in System $\mathcal{F}_e$</td>
<td>68</td>
</tr>
<tr>
<td>4.4.4 A definition of principal typings</td>
<td>70</td>
</tr>
<tr>
<td>4.4.5 Type soundness for implicitly-typed System $\mathcal{F}$</td>
<td>71</td>
</tr>
<tr>
<td>4.5 References</td>
<td>75</td>
</tr>
<tr>
<td>4.5.1 A counter example</td>
<td>75</td>
</tr>
<tr>
<td>4.5.2 Internalizing configurations</td>
<td>77</td>
</tr>
<tr>
<td>4.6 Damas and Milner’s type system</td>
<td>79</td>
</tr>
<tr>
<td>4.6.1 Definition</td>
<td>80</td>
</tr>
<tr>
<td>4.6.2 Syntax-directed presentation</td>
<td>82</td>
</tr>
<tr>
<td>4.6.3 Type soundness for ML</td>
<td>84</td>
</tr>
<tr>
<td>4.7 Ommitted proofs and answers to exercises</td>
<td>86</td>
</tr>
<tr>
<td><strong>5 Existential types</strong></td>
<td>93</td>
</tr>
<tr>
<td>5.1 Towards typed closure conversion</td>
<td>94</td>
</tr>
<tr>
<td>5.2 Existential types</td>
<td>96</td>
</tr>
<tr>
<td>5.2.1 Existential types in Church style (explicitly typed)</td>
<td>96</td>
</tr>
<tr>
<td>5.2.2 Implicitly-typed existential types</td>
<td>99</td>
</tr>
<tr>
<td>5.2.3 Existential types in ML</td>
<td>101</td>
</tr>
<tr>
<td>5.2.4 Existential types in OCaml</td>
<td>102</td>
</tr>
<tr>
<td>5.3 Typed closure conversion</td>
<td>103</td>
</tr>
<tr>
<td>5.3.1 Environment-passing closure conversion</td>
<td>103</td>
</tr>
<tr>
<td>5.3.2 Closure-passing closure conversion</td>
<td>105</td>
</tr>
</tbody>
</table>
## CONTENTS

5.3.3  Mutually recursive functions .......................... 107

6  Fomega: higher-kinds and higher-order types 111
   6.1  Introduction ............................................. 111
   6.2  From System $F$ to System $F^\omega$ .................. 112
       6.2.1  Properties ........................................ 115
   6.3  Expressiveness .......................... 115
       6.3.1  Map on pairs .................................... 115
       6.3.2  Abstracting over type operators .............. 116
       6.3.3  Existential types .............................. 117
       6.3.4  Church encoding of non-regular ADT ............ 118
       6.3.5  Encoding GADT—with explicit coercions ........ 121
   6.4  Beyond $F^\omega$ ....................................... 123
       6.4.1  Stratification .................................... 123
       6.4.2  Kinds ............................................. 123
       6.4.3  Recursion ......................................... 123
       6.4.4  Encoding of functors ............................ 124
       6.4.5  System $F^\omega$ in OCaml ..................... 126

7  Logical Relations 127
   7.1  Introduction ............................................. 127
       7.1.1  Parametricity ..................................... 127
   7.2  Normalization of simply-typed $\lambda$-calculus ... 129
   7.3  Observational equivalence ........................... 132
   7.4  Logical rel in simply-typed $\lambda$-calculus ....... 133
       7.4.1  Logical equivalence for closed terms .......... 133
       7.4.2  Logical equivalence for open terms .......... 135
   7.5  Logical rel. in $F$ ................................... 138
       7.5.1  Logical equivalence for closed terms .......... 139
   7.6  Applications ........................................... 144
   7.7  Extensions ............................................. 146
       7.7.1  Natural numbers .................................. 146
       7.7.2  Products ......................................... 147
       7.7.3  Sums .............................................. 147
       7.7.4  Lists .............................................. 147
       7.7.5  Existential types ................................ 147
       7.7.6  Step-indexed logical relations ................. 148
   7.8  Proofs and Solution to Exercises .................. 149
8 Type reconstruction

8.1 Introduction ................................................. 151
8.2 Type inference for simply-typed λ-calculus .......................... 152
  8.2.1 Constraints ........................................... 153
  8.2.2 A detailed example ..................................... 154
  8.2.3 Soundness and completeness of type inference ................. 156
  8.2.4 Constraint solving ..................................... 156
8.3 Type inference for ML ......................................... 158
  8.3.1 Milner’s Algorithm J ................................... 158
  8.3.2 Constraints ........................................... 159
  8.3.3 Constraint solving by example ............................ 163
  8.3.4 Type reconstruction .................................... 166
8.4 Type annotations ............................................. 169
  8.4.1 Explicit binding of type variables .......................... 170
  8.4.2 Polymorphic recursion .................................. 173
  8.4.3 mixed-prefix ........................................... 174
8.5 Equi- and iso-recursive types .................................. 175
  8.5.1 Equi-recursive types ................................... 175
  8.5.2 Iso-recursive types .................................... 177
  8.5.3 Algebraic data types .................................... 178
8.6 HM(X) ....................................................... 179
8.7 Type reconstruction in System F ................................ 181
  8.7.1 Type inference based on Second-order unification ............ 181
  8.7.2 Bidirectional type inference ................................ 182
  8.7.3 Partial type inference in MLF ............................. 184
8.8 Proofs and Solution to Exercises ................................ 184

9 Overloading .................................................... 187

9.1 An overview .................................................. 187
  9.1.1 Why use overloading? .................................... 187
  9.1.2 Different forms of overloading ............................. 188
  9.1.3 Static overloading ....................................... 189
  9.1.4 Dynamic resolution with a type passing semantics ........... 189
  9.1.5 Dynamic overloading with a type erasing semantics .......... 190
9.2 Mini Haskell .................................................. 191
  9.2.1 Examples in MH .......................................... 191
  9.2.2 The definition of Mini Haskell ............................. 192
  9.2.3 Semantics of Mini Haskell ................................ 194
  9.2.4 Elaboration of expressions ............................... 196
  9.2.5 Summary of the elaboration ............................... 197
Chapter 6

Fomega: higher-kinds and higher-order types

6.1 Introduction

From ML to System F. While simple types lacks polymorphism and thus forces many functions to be duplicated at different types, ML style prenex (or rank-1) polymorphism is already a considerable improvement that avoids most of code duplication.

In fact, this is mostly due to toplevel polymorphism. Although ML also allows local let-bound polymorphism in ML, this is less crucial, which allows Haskell to require explicit annotations for local polymorphism. Of course, core ML still lacks first-class polymorphism, which means higher-rank polymorphism, as well as primitive existential types. The absence of existential types has been partly balanced by the ML module system, which allows for type abstraction—a key feature for programming in the large. Nowadays, ML also feature first-class modules, which enables the encoding of first class-existential types. More recently, first-class iso-existential types have also been added together with GADTs.

Of course, moving to System F enables primitive first-class existential and universal-types in the core language, avoiding annoying encodings or limitations. This increases expressiveness by enabling encoding of data structures and many more programming patterns. Still, System F polymorphism is limited. . . .

Limits of System F. Although System F has higher-rank polymorphism, this is still sometimes quite limited.

Let us first illustrate this by considering the simple and rather illustrative example of the pair_map function, defined as \( \lambda f.\, x y.\, (f\, x, f\, y) \) which expects a functions and a pair of arguments and returns the pair of the applications of the function to each argument. (We could have taken the arguments in a pair, hence the name pair_map, but it is slightly simpler to receive them separately.)
This function can be given the two following incompatible types in System F:
\[
\forall \alpha_1. \forall \alpha_2. \ ((\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2
\]
\[
\forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2
\]
The first one requires \(x\) and \(y\) to admit a common type, while the second one requires \(f\) to be polymorphic. Or conversely, the first one allows the function domain and codomain to be arbitrary, while the second one allows the arguments to be arbitrary.

Unfortunately, we cannot give a type to \(\text{pair_map}\) that subsumes both types above in System F: it is missing the ability to describe the types of functions that are polymorphic in one parameter but whose domain and codomain are otherwise arbitrary \(i.e.\) of the form \(\forall \alpha. \tau[\alpha] \to \sigma[\alpha]\) for arbitrary one-hole types \(\tau\) and \(\sigma\).

To solve this, we need to abstract over \(\sigma\) and \(\tau\), which are here one-hole contexts, \(i.e.\) type functions, of kind \(* \to *\):
\[
\forall \varphi . \forall \psi . \forall \alpha_1 . \forall \alpha_2 . (\forall \alpha. \varphi \alpha \to \psi \alpha) \to \varphi \alpha_1 \to \varphi \alpha_2 \to \psi \alpha_1 \times \psi \alpha_2
\]
This is exactly what System \(F^\omega\) provides.

### 6.2 From System F to System F\(^\omega\)

**Kinds.** To emphasize the small difference between System F and System \(F^\omega\), we first introduce an alternative presentation of System F with kinds. This does not change the expressiveness at all. Indeed, kinds are used to categorize type variables of different kinds, but we introduce a unique kind \(*\). We still use a metavariable \(\kappa\) to range over kinds, even though it has a single element so far.

Well-formedness of types \(\Gamma \vdash \tau\) may then be rewritten as a kinding judgment \(\Gamma \vdash \tau : *\) defined inductively as follows:

\[
\begin{align*}
\vdash \Gamma & \quad \alpha : \kappa \in \Gamma \\
\vdash \Gamma & \quad \alpha : \kappa \\
\vdash \Gamma & \quad \alpha \notin \text{dom}(\Gamma) \\
\vdash \Gamma, \alpha : \kappa
\end{align*}
\]

\[
\begin{align*}
\vdash \Gamma, \alpha : \kappa & \\
\vdash \Gamma, \tau_1 \to \tau_2 : * \\
\vdash \Gamma, \forall \alpha : \kappa , \tau_1 : * \\
\vdash \Gamma, \forall \alpha : \kappa , \tau_2 : * \\
\vdash \Gamma, \forall \alpha : \kappa , \tau_1 \to \tau_2 : * \\
\vdash \Gamma, \forall \alpha : \kappa , \tau : *
\end{align*}
\]

We then add kind annotations on type variables in type abstractions and type polymorphism:
\[
\tau ::= \ldots \mid \forall \alpha : \kappa . \tau \\
M ::= \ldots \mid \Lambda \alpha : \kappa . M
\]
Typing rules for type abstraction and type applications are modified accordingly.

**Tabs**
\[
\begin{align*}
\Gamma & \quad \alpha : \kappa \vdash M : \tau \\
\Gamma & \quad \Lambda \alpha : \kappa , M : \forall \alpha : \kappa . \tau
\end{align*}
\]

**Tapp**
\[
\begin{align*}
\Gamma & \quad M : \forall \alpha : \kappa , \tau \\
\Gamma & \quad \tau' : \kappa \\
\Gamma & \quad M \ \tau' : \left[\alpha \to \tau'\right] \tau
\end{align*}
\]
So far, this is an equivalent formalization of System F.
Type functions. We now add type functions, moving from System $F$ to System $F^\omega$. For that purpose, we extend kinds to allow kinds of type functions:

$$\kappa ::= \ast \mid \kappa \Rightarrow \kappa$$

Notice that this does not only introduce a new kind $\ast \Rightarrow \ast$ but kinds of arbitrary shapes, to also allow type functions to take other type functions as arguments or return them as results. We may now introduce type functions and type application in type expressions:

$$\tau ::= \ldots \mid \lambda \alpha : \kappa.\tau \mid \tau \tau$$

These come with the following kinding rules:

$$\begin{align*}
\text{WfTypeApp} & : \quad \Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \quad \Gamma \vdash \tau_2 : \kappa_2 \\
& \quad \quad \quad \quad \Gamma \vdash \tau_1 \tau_2 : \kappa_1 \\
\text{WfTypeAbs} & : \quad \Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2 \\
& \quad \quad \quad \quad \Gamma \vdash \lambda \alpha : \kappa_1.\tau : \kappa_1 \Rightarrow \kappa_2
\end{align*}$$

Type reduction. Types must also be equipped with type-level $\beta$-reduction:

$$(\lambda \alpha.\tau) \sigma \longrightarrow [\alpha \mapsto \tau]\sigma$$

which is applicable in any type context. That is, if $T$ is an arbitrary one-hold type context

$$\tau \longrightarrow \tau'$$

Notice that the language of types became isomorphic to the simply-typed $\lambda$-calculus where types became kinds and terms became types. Hence, type reduction is the same as (full reduction) in simply-typed $\lambda$-calculus. Thus, type reduction preserves kinds\(^1\). Hence, kinds are erasable: they may only be checked when reading type expressions and ignored afterwards. As types, they do not contribute to the reduction, but are just carried over during the reduction.

Type reduction, which is strongly normalizing induces an equivalence on types written $\equiv_\beta$: two types are equivalent if they have have the same normal-form. An efficient implementation may however reduce terms by need.

Typing of expressions is up to type equivalence:

$$\begin{align*}
\text{TConv} & : \quad \Gamma \vdash M : \tau \\
& \quad \quad \quad \quad \quad \tau \equiv_\beta \tau' \\
& \quad \quad \quad \quad \quad \Gamma \vdash M : \tau'
\end{align*}$$

Notice that well-typedness $\Gamma \vdash M : \tau$ ensures well-kindness $\Gamma \vdash \tau : \ast$ (in the same way that it ensures well-formedness in System $F$). Notice that decidability of type checking in System $F^\omega$ relies on decidability of type equivalence, which here follows from strong normalization for types.

\(^1\)We have only proved subject reduction for CBV in the previous lessons, but still hold for full reduction.
Syntax
\[
\begin{align*}
\kappa & ::= \ast \mid \kappa \Rightarrow \kappa \\
\tau & ::= \alpha \mid \tau \rightarrow \tau \mid \forall \alpha :: \kappa.\tau \mid \lambda \alpha :: \kappa.\tau \mid \tau \tau \\
M & ::= x \mid \lambda x :: \tau. M \mid M M \mid \Lambda \alpha :: \kappa. M \mid M \tau
\end{align*}
\]

Kinding rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Precondition</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>\vdash \emptyset</td>
<td></td>
<td>\vdash \emptyset, \alpha :: \kappa</td>
</tr>
<tr>
<td>\vdash \emptyset</td>
<td>\alpha \not\in \text{dom} (\Gamma)</td>
<td>\vdash \Gamma, \alpha :: \kappa</td>
</tr>
<tr>
<td>\vdash \emptyset</td>
<td>\tau \not\in \text{dom} (\Gamma)</td>
<td>\vdash \Gamma, \tau :: \ast</td>
</tr>
</tbody>
</table>

Typing rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Precondition</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>\vdash x :: \tau \in \Gamma</td>
<td>\Gamma \vdash x :: \tau</td>
<td></td>
</tr>
<tr>
<td>\vdash \Gamma, x :: \tau_1 \vdash M :: \tau_2 \rightarrow \tau_2</td>
<td>\Gamma \vdash \lambda x :: \tau_1. M :: \tau_1 \rightarrow \tau_2</td>
<td></td>
</tr>
<tr>
<td>\vdash \Gamma \vdash M_1 :: \tau_1 \rightarrow \tau_2 \vdash M_2 :: \tau_1</td>
<td>\Gamma \vdash M_1 \vdash M_2 :: \tau_2</td>
<td></td>
</tr>
</tbody>
</table>

Dynamic semantics (unchanged, up to kind annotations in terms)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Precondition</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>\vdash \lambda x :: \tau. M \mid \Lambda \alpha :: \kappa. V</td>
<td>\Gamma \vdash \Lambda \alpha :: \kappa. M :: \forall \alpha :: \kappa. \tau</td>
<td></td>
</tr>
<tr>
<td>\Gamma \vdash M :: \forall \alpha :: \kappa. \tau \vdash M' :: \forall \alpha :: \kappa. \tau'</td>
<td>\Gamma \vdash M :: \forall \alpha :: \kappa. \tau'</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.1: System $F^\omega$, altogether
6.3. EXPRESSIVENESS

Still, we need not reduce types inside terms. Type reduction is needed for type conversion during typechecking but such reduction need not be performed on terms, which carries kind annotations attached to bound type variables, unchanged.

6.2.1 Properties

Main properties are preserved. Proofs are similar to those for System $F^\omega$.

- **Type soundness.** The proof is by *subject reduction* and *progress*.

- **Termination of reduction.** This holds in the absence of other constructs that can be use to introduce recursion, such as recursive types, recursive definitions or side effects (references, exceptions, control, *etc.*).

- **Typechecking is decidable.** This requires reduction at the level of types to check type equality. Checking type equality can be performed by putting types in normal forms using full reduction (on types)—or just in head normal forms. Normal forms for types exists as the language of type is a simply-typed $\lambda$-calculus (where kinds plays the role of types).

6.3 Expressiveness

System $F^\omega$ increases expressiveness and solves the limitations of System $F$ discussed above: Abstraction over type operators allows for more polymorphism, hence more principal types as illustrated with *pair_map*, abstraction over data structures such as monads, more encodings such as non regular datatypes or type equality, and more.

Kind annotations may often be obfuscating. For convenience, we often leave them implicit, using $\alpha$ and $\beta$ for type variables of kind $\star$ and $\varphi$ and $\psi$ for type variables of kind $\star \Rightarrow \star$, or of some arbitrary kind $\kappa$ given by context.

6.3.1 Map on pairs

We may now type the example of *pair_map*, whose implicitly-typed definition is $\lambda f x y. (f \, x, \, f \, y)$ by abstracting over (one parameter) type functions, *i.e.* type functions of kind $\star \rightarrow \star$. That is, the explicitly-typed version of *pair_map* is:

\[
\Lambda \varphi . \Lambda \psi . \Lambda \alpha_1 . \Lambda \alpha_2 . \lambda (f :: \forall \alpha . \varphi \, \alpha \rightarrow \psi \, \alpha) . \lambda x :: \varphi \, \alpha_1 . \lambda y :: \varphi \, \alpha_2 . (f \, \alpha_1 \, x, \, f \, \alpha_2 \, y)
\]

and has type:

\[
\forall \varphi . \forall \psi . \forall \alpha_1 . \forall \alpha_2 . (\forall \alpha . \varphi \, \alpha \rightarrow \psi \, \alpha) \rightarrow \varphi \, \alpha_1 \rightarrow \varphi \, \alpha_2 \rightarrow \psi \, \alpha_1 \times \psi \, \alpha_2
\]
We may recover the two incomparable types it had in System F by instantiation:

\[ \Lambda \alpha_1. \Lambda \alpha_2. \text{pair}_{\text{map}} (\lambda \alpha. \alpha_1) (\lambda \alpha. \alpha_2) \alpha_1 \alpha_2 \]

: \forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2

and

\[ \text{pair}_{\text{map}} (\lambda \alpha. \alpha) (\lambda \alpha. \alpha) \]

: \forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2

Actually, the former is not quite the expected typed, which should be \( \alpha_1 \to \alpha_2 \) rather than \( \forall \alpha. \alpha_1 \to \alpha_2 \), where the useless quantifier has been removed. This can be obtained by \( \eta \)-expansion: instantiation:

\[ \Lambda \alpha_1. \Lambda \alpha_2. \lambda f: \alpha_1 \to \alpha_2. \text{pair}_{\text{map}} (\lambda \alpha. \alpha_1) (\lambda \alpha. \alpha_2) \alpha_1 \alpha_2 (\Lambda \alpha. f) \]

: \forall \alpha_1. \forall \alpha_2. (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2

Notice that while the type of \( \text{pair}_{\text{map}} \) in System \( F^\omega \) is more general than both of these types, it is still not principal. For example, \( \varphi \) and \( \psi \) could depend on two variables, i.e. be of kind \(* \Rightarrow * \Rightarrow *\).

### 6.3.2 Abstracting over type operators

Given a type operator \( \varphi \), a monad is given by a pair of two functions of the following type (satisfying certain laws).

\[
\text{monad} \triangleq \lambda \varphi. \{ \text{ret} : \forall \alpha. \alpha \to \varphi \alpha; \text{bind} : \forall \alpha. \forall \beta. \varphi \alpha \to (\alpha \to \varphi \beta) \to \varphi \beta \} \\
: \ast \Rightarrow \ast \Rightarrow \ast
\]

Notice that \( \text{monad} \) is of higher-order kind—not just \( \ast \to \ast \).

For example, a generic map function, parameterized by some monad \( m \) can then be defined as follows:

\[
\text{fmap} \triangleq \Lambda \varphi. \lambda m: \text{monad} \varphi. \\
\Lambda \alpha. \Lambda \beta. \lambda f: (\alpha \to \beta). \lambda x: \varphi \alpha. m.\text{bind} \alpha \beta x (\lambda x: \alpha. m.\text{ret} \alpha (f x)) \\
: \forall \varphi. \forall m. \text{monad} \varphi \to \forall \alpha. \forall \beta. \alpha \Rightarrow \beta \Rightarrow \varphi \alpha \Rightarrow \varphi \beta
\]

**Abstraction over type operators without reduction.** In fact type abstraction over type operators is already available in Haskell, but does not handle \( \beta \)-reduction. In this case, type application \( \varphi \alpha \) behaves as a first-order type \( \text{App} (\varphi, \alpha) \) where \( \text{App} \) is a binary (application) symbol of kind \( (\kappa_1 \Rightarrow \kappa_2) \Rightarrow \kappa_1 \Rightarrow \kappa_2 \). That is,

\[
\varphi \alpha = \psi \beta \iff \varphi = \psi \land \alpha = \beta
\]

The expressiveness is then closer to System F than to System \( F^\omega \). As a counterpart of this limitation, this approach is compatible with type inference, based on first-order unification.
Such abstraction over type operators is actually encodable with OCaml modules. See Yallop and White (2014) (and also Kiselyov). As in Haskell, the encoding does not handle type $\beta$-reduction and as a consequence is compatible with type inference at higher kinds.

6.3.3 Existential types

We saw the encoding of existential types in System $\text{F}$:

$$\exists \alpha. \tau \triangleq \forall \beta. (\forall \alpha. \tau \rightarrow \beta) \rightarrow \beta$$

Hence, existential types could be provided as a family of primitives

$$\text{pack}_{\exists \alpha. \tau} \triangleq \Lambda \alpha. \lambda x : [\tau]. \Lambda \beta. \lambda k : \forall \alpha. ([\tau] \rightarrow \beta). k \alpha x$$

(and a similar encoding for $\text{unpack}_{\exists \alpha. \tau}$). In System $\text{F}$, this requires a different code for each type $\tau$. Indeed, sharing this code when $\tau$ varies requires to abstract over $\tau$, which is not possible in System $\text{F}$—but quite natural in System $\text{F}^\omega$!

In System $\text{F}^\omega$, we allow existential types to abstract over higher-kinded variables:

$$\exists \alpha. \tau \triangleq \forall \beta. (\forall \alpha. \tau \rightarrow \beta) \rightarrow \beta$$

In fact, we need not introduce a special construct $\exists \alpha. \tau$ for that purpose. We may instead introduce a family of type constants $\exists_\kappa$ indexed by $\kappa$ of respective kind $\kappa \Rightarrow \ast$. We then write $\exists_\kappa (\lambda \alpha. \tau)$ for $\exists \alpha. \tau$.

Revisiting the encoding, we may now abstract over some type variable $\varphi$ of kind $\kappa \Rightarrow \ast$, as follows:

$$\exists (\varphi :: \kappa). \tau \triangleq \forall (\beta :: \ast). (\forall (\varphi :: \kappa). \tau \rightarrow \beta) \rightarrow \beta$$

$$\exists_\kappa \triangleq \lambda (\psi :: \kappa \Rightarrow \ast). \forall (\beta :: \ast). (\forall (\varphi :: \kappa). \psi \varphi \rightarrow \beta) \rightarrow \beta$$

$$\text{pack}_\kappa : \forall (\psi :: \kappa \Rightarrow \ast). \forall (\varphi :: \kappa). \psi \varphi \rightarrow \exists_\kappa \psi$$

$$\triangleq \Lambda (\psi :: \kappa \Rightarrow \ast). \Lambda (\varphi :: \kappa). \lambda x : \psi \varphi. \lambda (\beta :: \ast). \lambda k : \forall (\varphi :: \kappa). (\psi \varphi \rightarrow \beta). k \varphi x$$

$$\text{unpack}_\kappa : \forall (\psi :: \kappa \Rightarrow \ast). \exists_\kappa \psi \rightarrow \forall (\beta :: \ast). (\forall (\varphi :: \kappa). (\psi \varphi \rightarrow \beta)) \rightarrow \beta$$

$$\triangleq \Lambda (\psi :: \kappa \Rightarrow \ast). \lambda x : \exists_\kappa \psi . x$$

The interest is that the encoding need not be defined at the metalevel, but directly provided as terms in System $\text{F}^\omega$ defined once for all.

**Exploiting abstraction over type operators.** The encoding of existential types in System $\text{F}^\omega$ is simplified by abstraction over type functions, which allows replacing primitive type constructs by type constants—just keeping arrow types as primitive as they play a particular role.

$$\tau = \alpha | \lambda \alpha : \kappa. \tau | \tau \tau | \tau \rightarrow \tau | \text{G}$$
where a family of type constants \( \mathcal{G} \in \mathcal{G} \) given with their kinds (left column):

\[
\begin{align*}
\times & : \star \Rightarrow \star \Rightarrow \star \quad (\tau \times \tau) \triangleq (\times) \tau_1 \tau_2 \\
+ & : \star \Rightarrow \star \Rightarrow \kappa \quad (\tau + \tau) \triangleq (+) \tau_1 \tau_2 \\
\forall \kappa & : (\kappa \Rightarrow \star) \Rightarrow \star \quad \forall (\varphi : \kappa) . \tau \triangleq \forall \kappa (\lambda (\varphi : \kappa) . \tau) \\
\exists \kappa & : (\kappa \Rightarrow \star) \Rightarrow \star \quad \exists (\varphi : \kappa) . \tau \triangleq \exists \kappa (\lambda (\varphi : \kappa) . \tau)
\end{align*}
\]

For convenience, we may still provide some notations as shown on the right column, but it is then just syntactic sugar!

**Abstraction over kinds.** Although not in System \( F^\omega \) \textit{per se}, we could also allow abstraction over kinds (see §6.4.2), and then just write:

\[
\begin{align*}
\hat{\forall} & : \forall \kappa. (\kappa \Rightarrow \star) \Rightarrow \star \quad \forall \varphi : \kappa . \tau \triangleq \hat{\forall} \kappa (\lambda (\varphi : \kappa) . \tau) \\
\hat{\exists} & : \forall \kappa. (\kappa \Rightarrow \star) \Rightarrow \star \quad \exists \varphi : \kappa . \tau \triangleq \hat{\exists} \kappa (\lambda (\varphi : \kappa) . \tau)
\end{align*}
\]

where the middle column are the application of the type constants \( \hat{\forall} \) and \( \hat{\exists} \) to a kind followed by a type, and the right column is the same when kinds are inferred.

### 6.3.4 Church encoding of non-regular ADT

Regular ADTs can be encoded in System F. For instance, the list datatype

\[
\text{type} \quad \text{List} \alpha = \begin{align*}
| \text{Nil} & : \forall \alpha. \text{List} \alpha \\
| \text{Cons} & : \forall \alpha. \alpha \rightarrow \text{List} \alpha \rightarrow \text{List} \alpha
\end{align*}
\]

has the following Church (CPS style) encoding:

\[
\begin{align*}
\text{List} & \triangleq \lambda \alpha. \forall \beta. \beta \rightarrow (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \\
\text{Nil} & \triangleq \Lambda \alpha. \Lambda \beta. \Lambda n : \beta . \lambda c : (\alpha \rightarrow \beta \rightarrow \beta) . n \\
\text{Cons} & \triangleq \Lambda \alpha. \lambda x : \alpha . \Lambda \ell : \text{List} \alpha . \\
& \quad \Lambda \beta . \lambda n : \beta . \lambda c : (\alpha \rightarrow \beta \rightarrow \beta) . c x (\ell \beta n c) \\
\text{fold} & \triangleq \Lambda \alpha. \Lambda \beta. \Lambda n : \beta . \lambda c : (\alpha \rightarrow \beta \rightarrow \beta) . \lambda \ell : \text{List} \alpha . \ell \beta n c
\end{align*}
\]

which is well-typed in System F.

In fact, one may attempt to generalize this signature to allow \( \beta \) to depend on \( \alpha \), hence
replacing $\beta$ by a type operator $\varphi$ of kind $* \Rightarrow *$:

\[
\begin{align*}
\text{List} & \triangleq \lambda \alpha. \forall \varphi . \varphi \alpha \rightarrow (\alpha \rightarrow \varphi \alpha \rightarrow \varphi \alpha) \rightarrow \varphi \alpha \\
\text{Nil} & \triangleq \Lambda \alpha . \Lambda \varphi . \lambda n : \varphi \alpha . \lambda c : (\alpha \rightarrow \varphi \alpha \rightarrow \varphi \alpha) . n \\
\text{Cons} & \triangleq \Lambda \alpha . \Lambda x : \alpha . \lambda \ell : \text{List} \alpha . \\
& \hspace{1cm} \Lambda \varphi . \lambda n : \varphi \alpha . \lambda c : (\alpha \rightarrow \varphi \alpha \rightarrow \varphi \alpha) . c \ x (\ell \varphi n c) \\
\text{fold} & \triangleq \Lambda \alpha . \Lambda \varphi . \lambda n : \varphi \alpha . \lambda c : (\alpha \rightarrow \varphi \alpha \rightarrow \varphi \alpha) . \lambda \ell : \text{List} \alpha . \ell \varphi n c
\end{align*}
\]

This seems more abstract since $\beta$ is now $\varphi \alpha$ which may depend on $\alpha$.

Actually not! Be aware of useless over-generalization! For regular ADTs, since all uses of $\varphi$ are applied to the same $\alpha$, this interface is actually no more general than the previous one. (One can then easily recover this interface by instantiation of the previous one.) However, this additional degree of liberty will be the key to then encoding of non-regular ADTs.

**Example 1** For a simpler example of over generalization, take the identity function \(id\) of type $\forall \alpha . \alpha \rightarrow \alpha$. If gave \(id\) the more general type $\forall \varphi . \forall \alpha . \varphi \alpha \rightarrow \varphi \alpha$, we may then recover the former by type instantiation: $\Lambda \varphi . \Lambda \alpha . id (\varphi \alpha)$.

Of course, raising type abstraction at higher rank is sometimes a key, as illustrated by the typing of \texttt{pair\_map}.

This is also the case for encoding non-regular ADT. Let us consider Okasaki’s datatype \texttt{Seq} for his purely functional efficient implementation of sequences:

\[
\begin{align*}
\text{type} & \quad \texttt{Seq} \ \alpha = \\
& | \text{Nil} : \forall \alpha . \texttt{Seq} \ \alpha \\
& | \text{Zero} : \forall \alpha . \texttt{Seq} (\alpha \times \alpha) \rightarrow \texttt{Seq} \ \alpha \\
& | \text{One} : \forall \alpha . \alpha \rightarrow \texttt{Seq} (\alpha \times \alpha) \rightarrow \texttt{Seq} \ \alpha
\end{align*}
\]
This may be encoded in System $\mathcal{F}_\omega$ as follows.

\begin{align*}
\text{Seq} & \triangleq \lambda \alpha. \forall \varphi. (\forall \alpha. \varphi \alpha) \to (\forall \alpha. \varphi (\alpha \times \alpha) \to \varphi \alpha) \to (\forall \alpha. \alpha \to \varphi (\alpha \times \alpha) \to \varphi \alpha) \to \varphi \alpha \\
\text{Nil} & \triangleq \lambda n. \lambda z. \lambda s. n \\
& : \forall \alpha. \text{Seq} \alpha \\
& = \Lambda \alpha. \Lambda \varphi. \lambda n : \forall \alpha. \varphi \alpha. \lambda z : \forall \alpha. \varphi (\alpha \times \alpha) \to \varphi \alpha. \lambda s : \forall \alpha. \alpha \to \varphi (\alpha \times \alpha) \to \varphi \alpha. n \\
\text{Zero} & \triangleq \lambda \ell. \lambda n. \lambda z. \lambda s. z \alpha (\ell n z s) \\
& : \forall \alpha. \text{Seq} (\alpha \times \alpha) \to \text{Seq} \alpha \\
& = \Lambda \alpha. \lambda \ell : \text{Seq} (\alpha \times \alpha). \\
& \quad \Lambda \varphi. \lambda n : \forall \alpha. \varphi \alpha. \lambda z : \forall \alpha. \varphi (\alpha \times \alpha) \to \varphi \alpha. \lambda s : \forall \alpha. \alpha \to \varphi (\alpha \times \alpha) \to \varphi \alpha. z \alpha (\ell \varphi n z s) \\
\text{One} & \triangleq \lambda x. \lambda \ell. \lambda n. \lambda z. \lambda s. x (\ell n z s) \\
& : \forall \alpha. \alpha \to \text{Seq} (\alpha \times \alpha) \to \text{Seq} \alpha \\
& = \Lambda \alpha. \lambda x : \alpha. \lambda \ell : \text{Seq} (\alpha \times \alpha). \\
& \quad \Lambda \varphi. \lambda n : \forall \alpha. \varphi \alpha. \lambda z : \forall \alpha. \varphi (\alpha \times \alpha) \to \varphi \alpha. \lambda s : \forall \alpha. \alpha \to \varphi (\alpha \times \alpha) \to \varphi \alpha. s x (\ell \varphi n z s) \\
\text{fold} & \triangleq \lambda \ell. \lambda n. \lambda z. \lambda s. \ell n z s \\
& : \forall \alpha. \text{Seq} \alpha \to \forall \varphi. (\forall \alpha. \varphi \alpha) \to (\forall \alpha. \varphi (\alpha \times \alpha) \to \varphi \alpha) \to (\forall \alpha. \alpha \to \varphi (\alpha \times \alpha) \to \varphi \alpha) \to \varphi \alpha \\
& = \Lambda \alpha. \lambda \ell : \text{Seq} \alpha. \Lambda \varphi. \lambda n : \forall \alpha. \varphi \alpha. \lambda z : \forall \alpha. \varphi (\alpha \times \alpha) \to \varphi \alpha. \lambda s : \forall \alpha. \alpha \to \varphi (\alpha \times \alpha) \to \varphi \alpha. \ell \varphi n z s
\end{align*}

To reconstruct the typed Church encoding—if it were not given, one should proceed as follows:

- First, build the untyped Church encoding:
  
  - The type is the type of deconstruction by cases: it is parametric (polymorphic in $\alpha$) and abstract over the type of sequences (polymorphic in $\varphi$) then receives as many functions as there are constructors in the datatype, if then returns a term of type $\varphi \alpha$, hence where both $\varphi$ and $\alpha$ will be chosen by the user.
  
  - We may then write the untyped encoding: each constructor just waits for three destruction functions (the three actions that will be passed depending on the constructor) and applies its corresponding action to its arguments.
  
  - Fold is just the $\eta$-expansion of the type of $\text{Seq} \alpha$. It is polymorphic in $\alpha$ and first receives an argument $\ell$ of type $\lambda \alpha. \forall \varphi. (\forall \alpha. \varphi \alpha) \to (\forall \alpha. \varphi (\alpha \times \alpha) \to \varphi \alpha) \to (\forall \alpha. \alpha \to \varphi (\alpha \times \alpha) \to \varphi \alpha) \to \varphi \alpha A$; then it expects as many actions as there are constructors and just pass them to $\ell$.

- Second, the types of the construction functions are exactly the same as those of the constructors.

- Third, the explicitly typed encoding can be derived mechanically by combining the untyped encoding and the types of the encoding.
module Eq : EQ = struct
  type ('a, 'b) eq = Eq : ('a, 'a) eq
  let coerce (type a) (type b) (ab : (a,b) eq) (x : a) : b = let Eq = ab in x
  let refl : ('a, 'a) eq = Eq

(* all these are propagation are automatic with GADTs *)
let symm (type a) (type b) (ab : (a,b) eq) : (b,a) eq = let Eq = ab in ab
let trans (type a) (type b) (type c)
  (ab : (a,b) eq) (bc : (b,c) eq) : (a,c) eq = let Eq = ab in bc
let lift (type a) (type b) (ab : (a,b) eq) : (a list, b list) eq =
  let Eq = ab in Eq
end

Figure 6.2: Leibnitz equality with GADT in OCaml

Notice that higher-rank is mandatory here as for each constructor \( \varphi \) is applied to both \( \alpha \) and \( \alpha \times \alpha \). This is why non-regular ADTs cannot be encoded in System F.

The encoding of the list datatypes could be obtained similarly, and then realize a posteriori that there is no gain in being polymorphic in \( \varphi \) since all occurrences of \( \varphi \) are always applied to the same variable \( \alpha \). This is always the case for church encodings or regular datatypes, hence, there is not need for such over generalization in the first place.

6.3.5 Encoding GADT—with explicit coercions

GADT can be encoded with a single equality type, existential types and non regular datatypes. Figure 6.2 gives an implementation of Leibnitz equality with a GADT in OCaml. We may then use a value of type \( (\tau, \sigma) \) Eq.eq as a proof of equality of the types \( \tau \) and \( \sigma \).

Leibnitz equality can also be defined in System \( F^{\omega} \) (Figure 6.3. In the figure, we have overlined proof terms and their types (respectively on the left and right columns) so as to help check typechecking.

We only implemented parts of the coercions of System \( Fc \): we do not have decomposition of equalities (the inverse of Lift), as this requires injectivity of the type operator, which cannot be assumed. Hence, some equality proofs are still missing. Notice that equivalences and liftings must be written explicitly in this encoding, which is cumbersome and obfuscating, while they are implicit with GADTs.
\[
\text{Eq} \triangleq \lambda \alpha. \lambda \beta. \forall \varphi. \varphi \alpha \to \varphi \beta
\]
hence, \(\text{Eq} \alpha \beta \equiv \forall \varphi. \varphi \alpha \to \varphi \beta\)

\text{coerce} \triangleq \lambda p. \lambda x. p \ x

\text{refl} \triangleq \lambda x. x

\text{symm} \triangleq \lambda p. p (\text{refl})

\text{trans} \triangleq \lambda p. \lambda q. q \ p

\text{lift} \triangleq \lambda p. p (\text{refl})

Figure 6.3: Leibnitz equality in System \(F^\omega\)
6.4 Beyond $F^\omega$

6.4.1 Stratification

Let us define the rank of a kind as usual: the base kind $\ast$ is of rank 1 and $\text{rank}(\kappa_1 \Rightarrow \kappa_2)$ is recursively defined as $\max(1 + \text{rank}\,\kappa_1, \text{rank}\,\kappa_2)$. Hence, type functions of kind $\ast \Rightarrow \ast$ or $\ast \Rightarrow \ast \Rightarrow \ast$ taking type parameters of base kind have rank 2 and type functions taking such type functions as arguments, e.g. of kind $(\ast \Rightarrow \ast) \Rightarrow \ast$, have rank 3.

We may define a hierarchy $F^1 \subseteq F^2 \subseteq F^3 \ldots \subseteq F^\omega$ of type systems of increasing expressiveness, where $F^n$ only uses kinds of rank $n$ and whose limit is $F^\omega$. Hence, System $F$ is just $F^1$. Most examples used in practice (and most of those we wrote) lie in $F^2$, just above System $F$. (Sometimes, ranks are shifted by one, starting with $F^2$ for System $F$.)

6.4.2 Kinds

Kind abstraction. In section §6.3.3 we used abstraction over kinds. Strictly speaking, this goes beyond System $F^\omega$, but System $F^\omega$ can easily be extended with kind abstraction and properties are preserved.

$$\forall \varphi.\forall \psi.\forall \alpha_1.\forall \alpha_2. \\
(\forall \alpha. \varphi \, \alpha \rightarrow \psi \, \alpha) \rightarrow \varphi \, \alpha_1 \rightarrow \varphi \, \alpha_2 \rightarrow \psi \, \alpha_1 \times \psi \, \alpha_2$$

One application is the use of constants instead of encodings as in section §6.3.3. Another application is to have even more general types. See the discussion on pair_map in §6.3.1.

Multiple base kinds We have used a single base kind $\ast$. Allow several base kinds raise no problems. For example, we may introduce an additional kind field and declare type constructors:

\[
\begin{align*}
\text{filled} & : \ast \Rightarrow \text{field} \\
\text{empty} & : \text{field} \\
\text{box} & : \text{field} \Rightarrow \ast
\end{align*}
\]

The will prevents the formation of types such as box ($\alpha \rightarrow \text{filled} \, \alpha$). This allows to build values $v$ of type box $\theta$ where $\theta$ of kind field statically tells whether $v$ is filled with a value of type $\tau$ or empty. Such kinding is actually used in OCaml for rows of object types, although kinds are hidden from the user using superficial syntax:

```
let get (x : ('a'; ..)) : 'a = x#get
```

The dots “..” here stands for a variable of another base kind representing a row of types.

6.4.3 Recursion

Equirecursive types Checking equality of equirecursive types in System $F$ is already non obvious, since unfolding may require $\alpha$-conversion to avoid variable capture. (See also
With higher-order types, it is even trickier, since unfolding at functional kinds could expose new type redexes.

Besides, the language of types would be the simply type λ-calculus with a fix-point operator: type reduction would not terminate. Therefore type equality would be undecidable, as well as type checking.

A solution is to restrict to recursion at the base kind *. This allows to define recursive types but not recursive type functions. Such an extension has been proven sound and decidable, but only for the weak form or equirecursive types (with the unfolding but not the uniqueness rule)—see Cai et al. (2016).

Equirecursive kinds Recursion could also occur just at the level of kinds, allowing kinds to be themselves recursive. Then, the language of types is the simply type λ-calculus with recursive types, which is equivalent to the untyped λ-calculus: every term is typable. Hence, without further restrictions reduction of types no longer terminates and type equality is ill-defined.

A solution proposed by Pottier is to force recursive kinds to be productive, reusing an idea from Nakano (2000, 2001) for controlling recursion on terms, but pushing it one level up. Then, type equality is well-defined, but only semi-decidable. This extension has been used to show that references in System F can be translated away in System $F^\omega$ with guarded recursive kinds Pottier (2011).

6.4.4 Encoding of functors

In early versions of OCaml, functors were generative: when a functor returns an abstract type, two applications of this functor to the very same structure produce new incompatible abstract types. By contrast, applicative functors would return two structures with compatible abstract types, allowing then to interoperate.

Generative functors Generative functors can be encoded in System F with existential types (as long as we ignore parametric types—or treat them as primitive). The idea to give functor $F$ a type of the form

$$\forall \alpha. \tau[\bar{\alpha}] \rightarrow \exists \beta. \sigma[\bar{\alpha}, \bar{\beta}]$$

Here $\tau[\bar{\alpha}]$ represents the signature of the argument with some abstract types $\bar{\alpha}$ while $\exists \beta. \sigma[\bar{\alpha}, \bar{\beta}]$ represents the signature of the result of the functor application. That is the abstract types $\bar{\alpha}$ appearing in the result signature are those taken from and shared with the argument. By contrast, $\bar{\beta}$ are the abstract types created by the functor application, and have fresh identities independent of the argument.

Therefore two successive applications with the very same argument (hence the same $\bar{\alpha}$) will create two signatures with incompatible abstract types $\bar{\beta}_1$ and $\bar{\beta}_2$, once the existential have been open.
Schematically, two applications of a functor $F$ to the very same structure argument $X$ as on the left column will be typed as on the right-column:

\[
\begin{align*}
\text{let module } Z_1 &= F(X) \text{ in} \\
\text{let module } Z_2 &= F(X) \text{ in} \\
\text{let } \beta_1, Z_1 &= \text{unpack } (F \bar{\rho} X) \text{ in} \\
\text{let } \beta_2, Z_2 &= \text{unpack } (F \bar{\rho} X) \text{ in} \ldots
\end{align*}
\]

Hence, the two resulting structures $Z_1$ and $Z_2$ have incompatible abstract types. (Typically, they contain a field of respective types $\beta_1$ and $\beta_2$ so that $Z.\ell = Z'.\ell$ is ill-typed.)

**Applicative functors.** Applicative functors can also be encoded, but in System $F^\omega$, using higher-order existential types.

To allow two identical applications of the functor $F$ to be compatible, we give it a type of the form:

\[
\exists \phi . \forall \bar{\alpha}. \tau[\bar{\alpha}] \to \sigma[\bar{\alpha}, \phi \bar{\alpha}]
\]

moving the existential $\beta$ across the arrow and universal quantifier. This requires skolemizing $\beta$ into a type function $\phi$ abstracted over the type variables $\bar{\beta}$.

The functor $F$ is first opened before being applied, becoming of type $\forall \bar{\alpha}. \tau[\bar{\alpha}] \to \sigma[\bar{\alpha}, \phi \bar{\alpha}]$ for some unknown $\phi$. As before, we specialize it to the abstract types, say $\bar{\rho}$, of the argument followed by the structure argument $X$ and get back a structure $Z$ of type $\sigma[\bar{\rho}, \phi \bar{\rho}]$.

Here $\phi \bar{\rho}$ are the abstract types created by the application. Each $\phi \rho$ is a new abstract type—one we know nothing about, as it is the application of an abstract type to $\bar{\rho}$. However, two successive applications with the same arguments (hence the same $\bar{\rho}$) will create two compatible structures whose signatures have the same shared abstract types $\phi \bar{\rho}$, as long as the functor has just been opened once for performing the two applications.

Schematically, the previous encoding on the left-hand side has been replaced by the one on the right-hand side:

\[
\begin{align*}
\text{let } \beta_1, Z_1 &= \text{unpack } (F(X)) \text{ in} \\
\text{let } \beta_2, Z_2 &= \text{unpack } (F(X)) \text{ in} \ldots \quad \text{let } \phi, F &= \text{unpack } F \text{ in} \\
\text{let } Z_1 &= F \bar{\rho} X \text{ in} \\
\text{let } Z_2 &= F \bar{\rho} X \text{ in} \ldots
\end{align*}
\]

More generally, functors could have both an applicative and a generative part, and have a type of the form:

\[
\exists \phi . \forall \bar{\alpha}. \tau[\bar{\alpha}] \to \exists \beta. \sigma[\bar{\alpha}, \phi \bar{\alpha}, \beta]
\]

Where $\phi \bar{\alpha}$ are the applicative shared abstract types and $\beta$ are the generative abstract types produced by the application. Or we may just have both forms and alternate between generative and applicative functors.

**Remarks:**

- We have used skolemization and therefore type functions to move the existential type across the universal type.
The application of an abstract type of higher-order kind to abstract types can be used to generate new (partially) abstract types!

The encoding of applicative functors in System $F^\omega$ uses these mechanisms to generate abstract types that can be shared. See ? and ? for more details and also ? for ongoing work.

### 6.4.5 System $F^\omega$ in OCaml

Second-order polymorphism is not primitive but encodable in OCaml, using polymorphic methods

```ocaml
def id = object method f : α. α → α = fun x → x end
let y (x : (f : α. α → α)) = x#f x in y id
```

or first-class modules

```ocaml
module type S = sig val f : α → α end
let id = (module struct let f x = x end : S)
let y (x : (module S)) = let module X = (val x) in X.f x in y id
```

Both solutions are quite verbose, though. Besides, second-order types are not first-class.

In principle, one can also reach higher-rank types OCaml, using first-class modules. However, this is not currently possible, due to (unnecessary) restrictions in the module language.

Modular explicits, a prototype extension\(^2\), leaves some of these restrictions, easing abstraction over first-class modules and allow a light-weight encoding of System $F^\omega$—with still some boiler-plate glue code. The encoding of \texttt{pair\_map} with modular explicit is presented in Figure 6.3 with its two specialized instances.

Higher-order polymorphism a la System $F^\omega$ is now also accessible in Scala-3. For instance, the monad example (with some variation on the signature) can be defined as:

```scala
trait Monad [F[_]] {
  def pure [A] (x: A) : F[A]
}
```


Still, this feature of Scala-3 is not emphasized and was not directly accessible in previous versions of Scala. Besides, Scala’s syntax and other complex features of Scala are somewhat obfuscating.

\(^2\)Available at [https@github.com:mrmr1993/ocaml](https@github.com:mrmr1993/ocaml)
6.4. BEYOND $F^\omega$

module type s = sig type t end
module type op = functor (A:s) -> s

let dp {F:op} {G:op} {A:s} {B:s} (f:{C:s} -> F(C).t -> G(C).t)
  (x : F(A).t) (y : F(B).t) : G(A).t * G(B).t = f {A} x, f {B} y

let dp1 (type a) (type b) (f : {C:s} -> C.t -> C.t) : a -> b -> a * b =
let module F(C:s) = C in let module G = F in
let module A = struct type t = a end in
let module B = struct type t = b end in
dp {F} {G} {A} {B} f

let dp2 (type a) (type b) (f : a -> b) : a -> a -> b * b =
let module A = struct type t = a end in
let module B = struct type t = b end in
let module F(C:s) = A in let module G(C:s) = B in
dp {F} {G} {A} {B} (fun {C:s} -> f)

Figure 6.4: pair_map with modular implicits

What’s next? The next step in expressiveness are dependent types, as illustrated in the Barendregt’s $\lambda$-cube:

![Barendregt’s $\lambda$-cube](image)

(1) Term abstraction on Types, as in System F;
(2) Type abstraction on Types, as in System $F^\omega$;
(3) Type abstraction on Terms: dependent types $\lambda\Pi$, $\lambda\Pi2$, $\lambda\Pi\omega$.

A form of dependent types is available in Haskell, but not in OCaml.
Bibliography

▷ A tour of scala: Implicit parameters. Part of scala documentation.


