Type systems for programming languages

Didier Rémy

Academic year 2020-2021
Version of August 25, 2022
Contents

1 Introduction ........................................... 7
   1.1 Overview of the course ............................ 7
   1.2 Requirements ..................................... 9
   1.3 About Functional Programming .................... 9
   1.4 About Types ....................................... 9
   1.5 Acknowledgment ................................... 11

2 The untyped \( \lambda \)-calculus ...................... 13
   2.1 Syntax ........................................... 13
   2.2 Semantics ........................................ 15
      2.2.1 Strong v.s. weak reduction strategies ....... 15
      2.2.2 Call-by-value semantics ..................... 16
   2.3 Answers to exercises ............................. 18

3 Simply-typed lambda-calculus ....................... 21
   3.1 Syntax ........................................... 21
   3.2 Dynamic semantics ................................ 21
   3.3 Type system ...................................... 22
   3.4 Type soundness ................................... 25
      3.4.1 Proof of subject reduction ................. 26
      3.4.2 Proof of progress ........................... 28
   3.5 Simple extensions ............................... 30
      3.5.1 Unit ......................................... 30
      3.5.2 Boolean ...................................... 30
      3.5.3 Pairs ......................................... 31
      3.5.4 Sums ......................................... 32
      3.5.5 Modularity of extensions .................... 32
      3.5.6 Recursive functions ......................... 33
      3.5.7 A derived construct: let-bindings ............ 33
   3.6 Exceptions ....................................... 35
      3.6.1 Semantics .................................... 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6.2 Typing rules</td>
<td>36</td>
</tr>
<tr>
<td>3.6.3 Variations</td>
<td>37</td>
</tr>
<tr>
<td>3.7 References</td>
<td>39</td>
</tr>
<tr>
<td>3.7.1 Language definition</td>
<td>39</td>
</tr>
<tr>
<td>3.7.2 Type soundness</td>
<td>41</td>
</tr>
<tr>
<td>3.7.3 Tracing effects with a monad</td>
<td>42</td>
</tr>
<tr>
<td>3.7.4 Memory deallocation</td>
<td>43</td>
</tr>
<tr>
<td>3.8 Ommitted proofs and answers to exercises</td>
<td>44</td>
</tr>
<tr>
<td>4 Polymorphism and System $\mathcal{F}$</td>
<td>49</td>
</tr>
<tr>
<td>4.1 Polymorphism</td>
<td>49</td>
</tr>
<tr>
<td>4.2 Polymorphic $\lambda$-calculus</td>
<td>51</td>
</tr>
<tr>
<td>4.2.1 Types and typing rules</td>
<td>51</td>
</tr>
<tr>
<td>4.2.2 Semantics</td>
<td>52</td>
</tr>
<tr>
<td>4.2.3 Extended System $\mathcal{F}$ with datatypes</td>
<td>54</td>
</tr>
<tr>
<td>4.3 Type soundness</td>
<td>58</td>
</tr>
<tr>
<td>4.4 Type erasing semantics</td>
<td>62</td>
</tr>
<tr>
<td>4.4.1 Implicitly-typed System $\mathcal{F}$</td>
<td>62</td>
</tr>
<tr>
<td>4.4.2 Type instance</td>
<td>65</td>
</tr>
<tr>
<td>4.4.3 Type containment in System $\mathcal{F}$</td>
<td>66</td>
</tr>
<tr>
<td>4.4.4 A definition of principal typings</td>
<td>68</td>
</tr>
<tr>
<td>4.4.5 Type soundness for implicitly-typed System $\mathcal{F}$</td>
<td>69</td>
</tr>
<tr>
<td>4.5 References</td>
<td>73</td>
</tr>
<tr>
<td>4.5.1 A counter example</td>
<td>73</td>
</tr>
<tr>
<td>4.5.2 Internalizing configurations</td>
<td>75</td>
</tr>
<tr>
<td>4.6 Damas and Milner's type system</td>
<td>77</td>
</tr>
<tr>
<td>4.6.1 Definition</td>
<td>78</td>
</tr>
<tr>
<td>4.6.2 Syntax-directed presentation</td>
<td>80</td>
</tr>
<tr>
<td>4.6.3 Type soundness for ML</td>
<td>82</td>
</tr>
<tr>
<td>4.7 Ommitted proofs and answers to exercises</td>
<td>84</td>
</tr>
<tr>
<td>5 Existential types</td>
<td>91</td>
</tr>
<tr>
<td>5.1 Towards typed closure conversion</td>
<td>92</td>
</tr>
<tr>
<td>5.2 Existential types</td>
<td>94</td>
</tr>
<tr>
<td>5.2.1 Existential types in Church style (explicitly typed)</td>
<td>94</td>
</tr>
<tr>
<td>5.2.2 Implicitly-typed existential types</td>
<td>97</td>
</tr>
<tr>
<td>5.2.3 Existential types in ML</td>
<td>99</td>
</tr>
<tr>
<td>5.2.4 Existential types in OCaml</td>
<td>100</td>
</tr>
<tr>
<td>5.3 Typed closure conversion</td>
<td>101</td>
</tr>
<tr>
<td>5.3.1 Environment-passing closure conversion</td>
<td>101</td>
</tr>
<tr>
<td>5.3.2 Closure-passing closure conversion</td>
<td>103</td>
</tr>
</tbody>
</table>
5 Type reconstruction

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>91</td>
</tr>
<tr>
<td>5.2</td>
<td>Type inference for simply-typed $\lambda$-calculus</td>
<td>92</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Constraints</td>
<td>93</td>
</tr>
<tr>
<td>5.2.2</td>
<td>A detailed example</td>
<td>94</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Soundness and completeness of type inference</td>
<td>96</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Constraint solving</td>
<td>96</td>
</tr>
<tr>
<td>5.3</td>
<td>Type inference for ML</td>
<td>98</td>
</tr>
</tbody>
</table>

6 Fomega: higher-kinds and higher-order types

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>109</td>
</tr>
<tr>
<td>6.2</td>
<td>From System F to System $F^{\omega}$</td>
<td>110</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Properties</td>
<td>111</td>
</tr>
<tr>
<td>6.3</td>
<td>Expressiveness</td>
<td>113</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Distrib pair in System $F^{\omega}$</td>
<td>113</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Abstracting over type operators</td>
<td>113</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Encoding of existential types</td>
<td>114</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Church encoding of non-regular ADT</td>
<td>115</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Encoding GADT—with explicit coercions</td>
<td>117</td>
</tr>
<tr>
<td>6.4</td>
<td>Beyond $F^{\omega}$</td>
<td>118</td>
</tr>
</tbody>
</table>

9 Logical Relations

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>183</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Parametricity</td>
<td>183</td>
</tr>
<tr>
<td>9.2</td>
<td>Normalization of simply-typed $\lambda$-calculus</td>
<td>185</td>
</tr>
<tr>
<td>9.3</td>
<td>Observational equivalence</td>
<td>187</td>
</tr>
<tr>
<td>9.4</td>
<td>Logical rel in simply-typed $\lambda$-calculus</td>
<td>189</td>
</tr>
<tr>
<td>9.4.1</td>
<td>Logical equivalence for closed terms</td>
<td>189</td>
</tr>
<tr>
<td>9.4.2</td>
<td>Logical equivalence for open terms</td>
<td>190</td>
</tr>
<tr>
<td>9.5</td>
<td>Logical rel in F</td>
<td>193</td>
</tr>
<tr>
<td>9.5.1</td>
<td>Logical equivalence for closed terms</td>
<td>194</td>
</tr>
<tr>
<td>9.6</td>
<td>Applications</td>
<td>199</td>
</tr>
<tr>
<td>9.7</td>
<td>Extensions</td>
<td>201</td>
</tr>
<tr>
<td>9.7.1</td>
<td>Natural numbers</td>
<td>201</td>
</tr>
<tr>
<td>9.7.2</td>
<td>Products</td>
<td>201</td>
</tr>
<tr>
<td>9.7.3</td>
<td>Sums</td>
<td>202</td>
</tr>
<tr>
<td>9.7.4</td>
<td>Lists</td>
<td>202</td>
</tr>
<tr>
<td>9.7.5</td>
<td>Existential types</td>
<td>202</td>
</tr>
<tr>
<td>9.7.6</td>
<td>Step-indexed logical relations</td>
<td>203</td>
</tr>
</tbody>
</table>

5.3.3 Mutually recursive functions

| Page | 105  |
5.3.1 Milner’s Algorithm \( \mathcal{I} \) .................................................. 98
5.3.2 Constraints ................................................................. 99
5.3.3 Constraint solving by example .......................................... 103
5.3.4 Type reconstruction ...................................................... 106
5.4 Type annotations ............................................................ 109
5.4.1 Explicit binding of type variables ...................................... 110
5.4.2 Polymorphic recursion .................................................... 113
5.4.3 mixed-prefix ............................................................... 114
5.5 Equi- and iso-recursive types ............................................ 115
5.5.1 Equi-recursive types ..................................................... 115
5.5.2 Iso-recursive types ...................................................... 117
5.5.3 Algebraic data types ..................................................... 118
5.6 \( \text{HM}(X) \) ................................................................. 119
5.7 Type reconstruction in System \( \mathcal{F} \) ........................................ 121
5.7.1 Type inference based on Second-order unification .................. 121
5.7.2 Bidirectional type inference ........................................... 122
5.7.3 Partial type inference in \( \text{MLF} \) ........................................ 124
5.8 Proofs and Solution to Exercises .......................................... 124

8 Overloading ................................................................. 159
8.1 An overview ................................................................. 159
8.1.1 Why use overloading? ................................................... 159
8.1.2 Different forms of overloading ........................................ 160
8.1.3 Static overloading ........................................................ 161
8.1.4 Dynamic resolution with a type passing semantics ............... 161
8.1.5 Dynamic overloading with a type erasing semantics .......... 162
8.2 Mini Haskell ................................................................. 162
8.2.1 Examples in \( \text{MH} \) ...................................................... 163
8.2.2 The definition of Mini Haskell ........................................ 164
8.2.3 Semantics of Mini Haskell ............................................ 165
8.2.4 Elaboration of expressions ............................................ 167
8.2.5 Summary of the elaboration .......................................... 168
8.2.6 Elaboration of dictionaries ........................................... 170
8.3 Implicitly-typed terms ...................................................... 172
8.4 Variations ................................................................. 177
8.5 Ommitted proofs and answers to exercises ................................ 181
Chapter 6

Fomega: higher-kinds and higher-order types

6.1 Introduction

Polymorphism in System F  Compare with simple types, which lacks polymorphism, and thus forces many functions to be duplicated at different types. ML style polymorphism is a considerable improvement by avoiding most of code duplication. Local let-bound polymorphism is also permitted in ML, but is less used in practice. However, core ML still lacks first-class polymorphism, which means higher-rank polymorphism and the lack of primitive existential types.

In ML, the module system allows for type abstraction, which for made the lack of existential types more sustainable—when programming in the large. First class-existential types are encodable in ML with first-class modules. They are now directly available via GADTs.

System F solves enables first-class existential and universal-types in the core language. This increases expressiveness by enabling encoding of data structures and many more programming patterns. Still, System F polymorphism is limited. . . .

Limits of System F: \( \lambda f x y. (f x, f y) \) Although System F has higher-rank polymorphism, this is still sometimes limited. For example, the map function on pairs whose untyped code if \( \lambda f. \lambda x. \lambda y. (f x, f y) \), says \texttt{distrib\_pair} can be given the following incompatible types in System F:

\[
\forall \alpha_1. \forall \alpha_2. \ (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2 \\
\forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2 
\]

The first one requires \( x \) and \( y \) to admit a common type, while the second one requires \( f \) to be polymorphic.

However, System F 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 $\forall \alpha. \tau[\alpha] \rightarrow \sigma[\alpha]$ for arbitrary one-hole types $\tau$ and $\sigma$. Hence, it cannot give a type to `distrib_pair` that subsumes both types above.

To solve this, we need to abstract over $\sigma$ and $\tau$, i.e. over type functions, of kind $\ast \rightarrow \ast$:

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

This is what System $F_\omega$ allows.

## 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 kinds in the presentation of System $F$—without changing expressiveness. That is, we write $\kappa$ for the single kind of types.

Well-formedness of types $\Gamma \vdash \tau$ may then be written as a well-kinding judgment $\Gamma \vdash \tau : \ast$, defined inductively as follows:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdash \emptyset$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\vdash \alpha : \kappa \in \Gamma$</td>
<td>$\Gamma$</td>
<td>$\alpha : \kappa$</td>
<td></td>
</tr>
<tr>
<td>$\Gamma \vdash \alpha : \kappa$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 : \ast$</td>
<td>$\Gamma$</td>
<td>$\tau_1$</td>
<td></td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_2 : \ast$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Gamma \vdash \tau_1 \rightarrow \tau_2 : \ast$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Gamma \vdash \forall \alpha :: \kappa, \tau : \ast$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We accordingly add kind annotations on type abstractions and type applications:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{TABS}$</td>
<td>$\Gamma, \alpha : \kappa$</td>
<td>$M : \tau$</td>
<td>$\Lambda\alpha :: \kappa. M : \forall \alpha :: \kappa. \tau$</td>
</tr>
<tr>
<td>$\text{TAPP}$</td>
<td>$\Gamma$</td>
<td>$M : \forall \alpha :: \kappa, \tau$</td>
<td>$\Gamma \vdash \tau_1 \vdash \tau_2 : \forall \alpha :: \kappa. \tau$</td>
</tr>
</tbody>
</table>

So far, this is an equivalent formalization of System $F$.

### Type functions

We now add type functions, moving form System $F$ to System $F_\omega$. For that purpose, we redefine kinds, so as to introduce kinds of type functions:

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

We may now introduce type function and type application in type expressions:

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

with the following kinding rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{WfTypeApp}$</td>
<td>$\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1$</td>
<td>$\Gamma \vdash \tau_2 : \kappa_2$</td>
<td>$\Gamma \vdash \tau_1 \tau_2 : \kappa_1$</td>
</tr>
<tr>
<td>$\text{WfTypeAbs}$</td>
<td>$\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2$</td>
<td></td>
<td>$\Gamma \vdash \lambda \alpha :: \kappa_1. \tau : \kappa_1 \Rightarrow \kappa_2$</td>
</tr>
</tbody>
</table>
Type reduction  Types are now equipped with $\beta$-reduction:

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

which is applicable in any type context.

Notice that type reduction is the same as (full reduction) in simply-typed $\lambda$-calculus when kinds and types now play the role of types and terms. It thus preserves well-kindness and kinds\(^1\). Hence, kinds are erasable. Kinds may only be checked when reading type expressions and ignored afterwards. As types, they do not contribute to the reduction.

Type reduction induces a notion of $\beta$-equivalence on types, which is decidable: for example, by normalization (which terminates) and comparison of normal forms—although an efficient implementation would reduce terms by need.

Typing of expressions is up to type equivalence, i.e. $\beta$-conversion:

$$\text{TConv} \quad \frac{\Gamma \vdash M : \tau \quad \tau \equiv_\beta \tau'}{\Gamma \vdash M : \tau'}$$

Notice that well-typedness $\Gamma \vdash M : \tau$ ensures well-kindness $\Gamma \vdash \tau : \ast$. Notice that decidability of type checking in System $F^\omega$ relies on decidability of type equivalence.

However, 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.

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).

\(^1\)We have only proved subject reduction for CBV, though in the previous lessons.
CHAPTER 6. FOMEGA: HIGHER-KINDS AND HIGHER-ORDER TYPES

Syntax

\[
\begin{align*}
\kappa & ::= * | \kappa \Rightarrow \kappa \\
\tau & ::= \alpha | \tau \rightarrow \tau | \forall \alpha :: \kappa. \tau | \lambda \alpha :: \kappa. \tau | \tau \tau \\
M & ::= x | \lambda \alpha :: \kappa. M | M M | \Lambda \alpha :: \kappa.M | M \tau
\end{align*}
\]

Kinding rules

\[
\begin{align*}
\varnothing & \vdash \\
\Gamma & \vdash \alpha \notin \text{dom}(\Gamma) \quad \alpha : \kappa \in \Gamma \\
\Gamma & \vdash x : \tau \\
\Gamma & \vdash \alpha : \kappa \\
\Gamma, \alpha : \kappa & \vdash \tau : * \\
\Gamma & \vdash \tau_{1} : * \quad \Gamma & \vdash \tau_{2} : * \\
\Gamma & \vdash \tau_{1} \rightarrow \tau_{2} : * \\
\Gamma, \alpha : \kappa & \vdash \tau : * \\
\Gamma, \alpha : \kappa & \vdash \tau_{1} : \kappa_{1} \quad \Gamma, \alpha : \kappa & \vdash \tau_{2} : \kappa_{2} \\
\Gamma & \vdash \tau_{1} \tau_{2} : \kappa_{1}
\end{align*}
\]

Typing rules

\[
\begin{align*}
\text{Var} & \quad \text{Abs} & \quad \text{App} \\
\Gamma & \vdash x : \tau & \Gamma, x : \tau_{1} & \vdash M : \tau_{2} & \Gamma & \vdash \lambda x : \tau_{1}. M : \tau_{1} \rightarrow \tau_{2} & \Gamma & \vdash M_{1} : \tau_{1} \rightarrow \tau_{2} & \Gamma & \vdash M_{2} : \tau_{1} \\
\Gamma, \alpha : \kappa & \vdash M : \tau & \Gamma & \vdash \Lambda \alpha :: \kappa.M : \forall \alpha :: \kappa. \tau & \Gamma & \vdash \tau' : \kappa & \Gamma & \vdash \tau' : \kappa & \Gamma & \vdash \tau : \tau' \\
\Gamma, \alpha : \kappa & \vdash M : \forall \alpha :: \kappa. \tau & \Gamma & \vdash \tau' : \kappa & \Gamma & \vdash M : \tau' & \Gamma & \vdash \tau \equiv_{\beta} \tau' & \Gamma & \vdash M : \tau'
\end{align*}
\]

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

\[
\begin{align*}
V & ::= \lambda x : \tau. M | \Lambda \alpha :: \kappa. V \\
E & ::= [ ] M | V [ ] | [] \tau | \Lambda \alpha :: \kappa. [ ] \\
(\lambda x : \tau. M) V & \rightarrow [x \mapsto V] M \\
(\Lambda \alpha :: \kappa. V) \tau & \rightarrow [\alpha \mapsto \tau] V \\
M & \rightarrow M' \\
E[M] & \rightarrow E[M']
\end{align*}
\]

Figure 6.1: System $F^{\omega}$, altogether
6.3 Expressiveness

System $F^\omega$ increases expressiveness and allows to solve the limitations of System $F$ discussed in the introduction.

Just adding more polymorphism on ad hoc examples such as `distrib_pair`, exploiting abstraction over type operators, such as examples with monads, or the encoding of existential types, or more advanced encodings such as non regular datatypes, and type equality.

### 6.3.1 Distrib pair in System $F^\omega$

We may now type the example of `distrib_pair`, 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 `distrib_pair` is:

$$\Lambda \varphi . \Lambda \psi . \Lambda \tau_1 . \Lambda \alpha_2 . \lambda \alpha . (\forall (\alpha :: \star) . \varphi \alpha \rightarrow \psi \alpha) . \lambda x . \forall \alpha_1 . \lambda y . \forall \alpha_2 . (f \ \alpha_1 x, f \ \alpha_2 y)$$

of type:

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

We may recover, the two incomparable types it had in System $F$:

$$\Lambda (\alpha_1 :: \star) . \Lambda (\alpha_2 :: \star) . \Lambda \tau_1 . \Lambda \alpha_2 . \lambda \alpha . (\forall (\alpha_1 :: \star) . \forall (\alpha_2 :: \star) . (\alpha_1 \rightarrow \alpha_2) \rightarrow (\alpha_1 \rightarrow \alpha_2) \rightarrow (\alpha_1 \rightarrow \alpha_2) \times \alpha_2$$

and

$$\forall (\alpha_1 :: \star) . \forall \alpha_2 . (\forall (\alpha :: \star) . \alpha \rightarrow \alpha) \rightarrow (\alpha_1 \rightarrow \alpha_2) \rightarrow (\alpha_1 \rightarrow \alpha_2) \times \alpha_2$$

While the type of `distrib_pair` in System $F^\omega$ is much more general than in System $F$, it is still not principal. For example, $\varphi$ and $\psi$ could depend on two variables, i.e. be of kind $\star \Rightarrow \star \Rightarrow \star$, or many other kinds.

### 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).

$$M \triangleq \lambda (\varphi :: \star \Rightarrow \star) . \{ \text{ret} : \forall (\alpha :: \star) . \alpha \rightarrow \varphi \alpha; \text{bind} : \forall (\alpha :: \star) . \forall (\beta :: \star) . \varphi \alpha \rightarrow (\alpha \rightarrow \varphi \beta) \rightarrow \varphi \beta \}$$

$$(\star \Rightarrow \star) \Rightarrow \star$$

(Notice that $M$ is itself of higher kind.)
For example, a generic map function, can then be defined as follows:

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

Abstraction over type operators is available in Haskell—but without \(\beta\)-reduction: type application \(\varphi \alpha\) is encoded as a first-order type \(\text{App} (\varphi, \alpha)\) where \(\text{App}\) is a binary (application) symbol of kind \((\kappa_1 \Rightarrow \kappa_2) \Rightarrow \kappa_1 \Rightarrow \kappa_2\).

Interestingly, this approach is compatible with type inference, which is based on first-order unification. However, there is no \(\beta\)-reduction at the level of types, that is:

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

Therefore, this does not have the expressiveness of System \(F^\omega\) at all.

Abstraction over type operators is also encodable with OCaml modules. See \(?\) (and also \(?\)). 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 Encoding of existential types

We saw the encoding of existential types in System \(F\):

\[\exists \alpha. \tau \equiv \forall \beta. (\forall \alpha. \tau \rightarrow \beta) \rightarrow \beta\]

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

\[\text{pack}_{3\alpha.\tau} \equiv \Lambda \alpha. \lambda x : [\tau]. \Lambda \beta. \lambda k : \forall \alpha. ([\tau] \rightarrow \beta). k \alpha x\]

(and a similar encoding for \(\text{unpack}_{3\alpha.\tau}\)).

Unfortunately, this requires a different code for each type \(\tau\). To have a unique code, we need to abstract over \(\tau\) which is not possible in System \(F\), but quite natural in System \(F^\omega\).

We first extend existential types to abstraction over higher kinding variables, letting \(\exists (\alpha : \kappa). \tau\) mean \(\forall (\beta : \ast). (\forall (\alpha : \kappa). \tau \rightarrow \beta) \rightarrow \beta\). In fact, we need not introduce a special construct \(\exists (\alpha : \kappa). \tau\) for that purpose, but just a new type constant \(\exists_k\) of kind \(\kappa \Rightarrow \ast\) and write \(\exists_k (\lambda (\alpha : \kappa). \tau)\) for \(\exists (\alpha : \kappa). \tau\).
Then, we may abstract the encodings over some type variable $\varphi$ of kind $\kappa \Rightarrow \ast$, as follows:

$\exists_{\kappa} = \lambda (\varphi :: \kappa \Rightarrow \ast). \forall (\alpha :: \kappa). \varphi \rightarrow \exists_{\kappa} \varphi$

$\text{pack}_{\kappa} = \lambda x : \varphi. \lambda (\beta :: \ast). \lambda k : \forall (\alpha :: \kappa). (\varphi \Rightarrow \beta). k \alpha x$

$\text{unpack}_{\kappa} = \lambda x : \exists_{\kappa} \varphi. \lambda (\tau :: \kappa). \tau \Rightarrow \beta$

The interest is that the encoding need not be defined at the metalevel, but directly provided as two terms of System $F^\omega$—with may be defined once for all.

This idea of exploiting kinds once we have type functions, the language of types could be reduced to $\lambda$-calculus with constants (plus the arrow types kept as primitive):

$$\tau = \alpha \mid \lambda \alpha :: \kappa. \tau \mid \tau \tau \mid \tau \rightarrow \tau \mid G$$

where type constants $G \in G$ are given with their kind and syntactic sugar:

$$\begin{align*}
\times :: \ast & \Rightarrow \ast \Rightarrow \ast & (\tau \times \tau) & \triangleq (\times) \tau_1 \tau_2 \\
+ :: \ast & \Rightarrow \ast \Rightarrow \kappa & (\tau + \tau) & \triangleq (+) \tau_1 \tau_2 \\
\forall :: (\kappa \Rightarrow \ast) \Rightarrow \ast & & \forall \varphi : \kappa. \tau & \triangleq \forall_{\kappa} \varphi :: \kappa \Rightarrow \ast. \tau \\
\exists :: (\kappa \Rightarrow \ast) \Rightarrow \ast & & \exists \varphi : \kappa. \tau & \triangleq \exists_{\kappa} \varphi :: \kappa \Rightarrow \ast. \tau
\end{align*}$$

This is even nicer if System $F^\omega$ were extended with kind abstraction (see §6.4), as we could then just write:

$$\begin{align*}
\hat{\forall} :: \forall_{\kappa}. (\kappa \Rightarrow \ast) \Rightarrow \ast & \quad \forall \varphi : \kappa. \tau & \triangleq \hat{\forall}_{\kappa} \varphi :: \kappa \Rightarrow \ast. \tau \\
\hat{\exists} :: \forall_{\kappa}. (\kappa \Rightarrow \ast) \Rightarrow \ast & \quad \exists \varphi : \kappa. \tau & \triangleq \hat{\exists}_{\kappa} \varphi :: \kappa \Rightarrow \ast. \tau
\end{align*}$$

where the right hand sides are no more syntactic forms but the application of the type constants $\hat{\forall}$ and $\hat{\exists}$ to a kind a type.

### 6.3.4 Church encoding of non-regular ADT

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

```plaintext
| type List α =
| Nil : ∀α. List α
| Cons : ∀α. α → List α → List α
```

has the following Church (CPS style) encoding:
CHAPTER 6. FOMEGA: HIGHER-KINDS AND HIGHER-ORDER TYPES

```
List  △  λα. ∀β. β → (α → β → β) → β
Nil   △  Λα. Λβ. λn : β. λc : (α → β → β). n
Cons  △  Λα. λx : α. λℓ : List α.
      Λβ. λn : β. λc : (α → β → β). cx (ℓ β n c)
fold  △  Λα. Λβ. λn : β. λc : (α → β → β). ℓ : List α. ℓ β n c
```

In fact, we may give use following signature in System $F^\omega$:

```
List  △  λα. ∀φ. α → (α → φ α → φ α) → φ α
Nil   △  Λα. λφ. λn : φ α. λc : (α → φ α → φ α). n
Cons  △  Λα. λx : α. λℓ : List α.
      λφ. λn : φ α. λc : (α → φ α → φ α). cx (ℓ φ n c)
fold  △  Λα. λφ. λn : φ α. λc : (α → φ α → φ α). ℓ : List α. ℓ φ n c
```

This seems more abstract since $β$ is now $φ α$ which may depend on $α$.

Actually not! Be aware of useless over-generalization! For regular ADTs, since all uses of $φ$ are applied to the same $α$, this interface is actually no more general that the previous one. However, this additional degree of liberty will be the key to then encoding of non regular ADTs.

A simpler example of over generalization is the type of the identity. $∀α. α → α$ could be generalized as $∀φ. ∀α. φ → φ α$: it is easy to check that there are retyping functions ( typable in System $F^\omega$ that are $βη$ convertible to the identity).

By contrast type abstraction at higher-rank was a key for the typing of distrib.pair.

Let us consider Okasaki’s Seq non-regular ADT:

```
type Seq α =
    Nil : ∀α. Seq α
    Zero : ∀α. Seq (α × α) → Seq α
    One : ∀α. α → Seq (α × α) → Seq α
```
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

This may be encoded in System $F^\omega$ as:

\[
\begin{align*}
\text{Seq} & \triangleq \lambda a. \forall F. F a \rightarrow (F(\alpha \times \alpha) \rightarrow F \alpha) \rightarrow (\alpha \rightarrow F(\alpha \times \alpha) \rightarrow F \alpha) \rightarrow F \alpha \\
\text{Nil} & \triangleq \lambda n. \lambda z. \lambda s. n \\
\text{Zero} & \triangleq \lambda \alpha. \lambda F. \lambda n : F \alpha. \lambda z : F(\alpha \times \alpha) \rightarrow F \alpha. \lambda s : \alpha \rightarrow F(\alpha \times \alpha) \rightarrow F \alpha. n \\
\text{One} & \triangleq \lambda x. \lambda \ell. \lambda n. \lambda z. \lambda s. s \ x (\ell \ n \ z) \\
\text{fold} & \triangleq \lambda n. \lambda z. \lambda s. \lambda \ell. n \ z \ s \\
\text{Fold} & \triangleq \lambda a. \lambda F. \lambda n : F \alpha. \lambda z : F(\alpha \times \alpha) \rightarrow F \alpha. \lambda s : \alpha \rightarrow F(\alpha \times \alpha) \rightarrow F \alpha. s \ x (\ell \ n \ z)
\end{align*}
\]

Indeed, higher-rank is mandatory 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 $F$.

### 6.3.5 Encoding GADT—with explicit coercions

We have seen that 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 $(\tau, \sigma) \text{Eq.eq}$ 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 is not given.
CHAPTER 6. FOMEGA: HIGHER-KINDS AND HIGHER-ORDER TYPES

\[
\begin{align*}
\text{Eq} & \triangleq \lambda \alpha. \lambda \beta. \forall \varphi. \varphi \alpha \to \varphi \beta \\
\text{coerce} & \triangleq \lambda p. \lambda x. p x \\
\text{refl} & \triangleq \lambda x. x \\
\text{symm} & \triangleq \lambda p. p (\text{refl}) \\
\text{trans} & \triangleq \lambda p. \lambda q. q (\text{refl}) \\
\text{lift} & \triangleq \lambda p. p (\text{refl}) \\
\end{align*}
\]

hence, \( \text{Eq} \alpha \beta \equiv \forall \varphi. \varphi \alpha \to \varphi \beta \)

\[
\begin{align*}
\Lambda \alpha. \Lambda \beta. \lambda p. \text{Eq} \alpha \beta. \lambda x. \alpha. p (\lambda \gamma. \gamma) x \\
\end{align*}
\]

\[
\begin{align*}
\forall \alpha. \forall \varphi. \varphi \alpha \to \varphi \alpha & \equiv \forall \alpha. \text{Eq} \alpha \alpha \\
\end{align*}
\]

\[
\begin{align*}
\forall \alpha. \forall \beta. \forall \gamma. \text{Eq} \alpha \beta \to \text{Eq} \gamma \to \text{Eq} \alpha \gamma & : \text{Eq} \beta \to \text{Eq} \alpha \gamma \\
\forall \alpha. \forall \beta. \forall \varphi. \text{Eq} \alpha \beta \to \text{Eq} (\varphi \alpha) (\varphi \beta) & : \text{Eq} (\varphi \alpha) (\varphi \alpha) \to \text{Eq} (\varphi \alpha) (\varphi \beta)
\end{align*}
\]

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

Equivalences and liftings must be written explicitly, while they are implicit with GADTs.

Some GATDs can be encoded, using equality plus existential types.

6.4 Beyond \( F^\omega \)

Let us define the rank of a kind as usual: the base kind \( * \) 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 \( * \Rightarrow * \) taking type parameters of base kind have rank 1 and type functions taking such type functions as arguments have rank 2.

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 \( n \), whose whose limit is System \( F^\omega \)—and where System F is just \( F^0 \) (ranks are sometimes shifted by one, starting with \( F = F^2 \)).

Most examples in practice (and those we wrote) lies in \( F^2 \), just above \( F \).

Kind abstraction In section §??, we have used abstraction over kinds. Strictly speaking, this goes beyond System \( F^\omega \), but this all properties are preserved.

\[
\begin{align*}
L \forall (\varphi \Rightarrow *). \forall (\psi \Rightarrow *). \forall (\alpha_1 \Rightarrow *). \forall (\alpha_2 \Rightarrow *). \\
(\forall (\alpha \Rightarrow *). \varphi \alpha \to \psi \alpha) \to \varphi \alpha_1 \to \varphi \alpha_2 \to \psi \alpha_1 \times \psi \alpha_2
\end{align*}
\]

One applications is the use of constants instead of encodings as in section §??. Another application could be having even more general types. See for example, this discussion on distrib pair (§6.3.1).
**Multiple base kinds**  We could have several base kinds, e.g. ∗ and field with type constructors:

\[
\text{filled} : ∗ \Rightarrow \text{field} \\
\text{empty} : \text{field} \Rightarrow ∗
\]

Prevents ill-formed types such as \( \text{box} (α \rightarrow \text{filled} α) \).

This allows to build values \( v \) of type \( \text{box} θ \) where \( θ \) of kind field statically tells whether \( v \) is filled with a value of type \( τ \) or empty. This is used in OCaml for rows of object types, although kinds are hidden from the user using superficial syntax:

\[
\text{let get} (x : \langle \text{get} : 'a ; .. \rangle) : 'a = x#\text{get}
\]

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

**Equirecursive types**  Checking equality of equirecursive types in System F is already non obvious, since unfolding may require \( α \)-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 ?.

**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, equivalent to the untyped \( λ \)-calculus—every term is typable. Without further restrictions reduction of types does not terminate 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. Type equality become well-defined and semi-decidable. This extension has been used to show that references in System F can be translated away in System \( F^ω \) with guarded recursive kinds.

**Encoding of functors**  In OCaml functions are by default generative: when a functor return abstract types, two applications of this functor to same structures will produce new incompatible abstract types. By contrast, applicative functors would return two structures with compatible abstract types.

While generative functors can be encoded in System F with existential types (as long as we ignore parametric types—or treat add as primitive). The idea to give functor \( F \) a type of the
form \( \forall \alpha. \tau[\alpha] \rightarrow \exists \beta. \sigma[\alpha, \beta] \). Then, if \( X, Y \) has type \( \tau[\rho] \), two successive applications \( F(X) \) and \( F(X) \) have types \( \exists \beta. [\rho, \beta] \) with different abstract types \( \beta \) and cannot interoperate (on components involving \( \beta \)).

Indeed, the program

\[
\begin{align*}
&\text{let } Y = \text{unpack } FX \text{ in} \\
&\text{let } Z = \text{unpack } FX \text{ in} \\
&Y = Z
\end{align*}
\]

is ill-typed.

To allow two identical applications of the functor \( F \) to be compatible, a solution is to give \( F \) a type of the form: \( \exists \varphi. \forall \alpha. \tau[\alpha] \rightarrow \sigma[\alpha, \varphi \alpha] \). Then, when \( F \) is applied it is first open and given type \( \forall \alpha. \tau[\alpha] \rightarrow \sigma[\alpha, \psi \alpha] \) for some unknown \( \psi \). Then the result of the application to an argument of type \( \sigma[\rho] \) will have type \( \sigma[\rho, \psi \rho] \) where the abstract type (application) \( \psi \rho \) describes the abstract types created by the application. Hence two applications to the same argument will have the same type, as the same abstract type \( \psi \) has just been open once and all occurrences of \( \psi \rho \) are equal hence compatible.

The code would look like

\[
\begin{align*}
&\text{let } \psi, f = \text{unpack } F \text{ in} \\
&\text{let } Y = FX \text{ in} \\
&\text{let } Z = FX \text{ in} \\
&Y = Z
\end{align*}
\]

The encoding heavily relies on higher-rank types and may only be implemented in System \( F^\omega \). See \? and \? for details.

**System \( F^\omega \) in OCaml.** Second-order polymorphism is not primitive but encodable in OCaml, using polymorphic methods

\[
\begin{align*}
&\text{let } id = \text{object method } f : 'a \rightarrow 'a = \text{fun } x \rightarrow x \text{ end} \\
&\text{let } y (x : <f : 'a \rightarrow 'a>) = x#f x \text{ in } y id
\end{align*}
\]

or first-class modules

\[
\begin{align*}
&\text{module type } S = \text{sig val } f : 'a \rightarrow 'a \text{ end} \\
&\text{let } id = (\text{module struct let } f x = x \text{ end : } S) \\
&\text{let } y (x : (\text{module } S)) = \text{let module } X = (\text{val } x) \text{ in } X.f x \text{ in } y id
\end{align*}
\]

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\footnote{Available at [git@github.com:mrmr1993/ocaml.git](https://git@github.com/mrmr1993/ocaml.git)}, 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 distrib_pair with modular explicit is presented in Figure \ref{fig:dist_pair} with its two specialized instances.
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: distrib_pair with modular implicit

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]
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

See [https://www.baeldung.com/scala/dotty-σ2/scala-σ2](https://www.baeldung.com/scala/dotty-σ2/scala-σ2).

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 obfuscating.

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

![Diagram of the $\lambda$-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.

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. 

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. 

▷ Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. 


▷ Nick Benton and Andrew Kennedy. Exceptional syntax journal of functional programming. 

▷ Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing Polymorphic Properties, 


November 2005.


