


Bibliography


<table>
<thead>
<tr>
<th>Type Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>type checking</td>
<td>50</td>
</tr>
<tr>
<td>explicit types</td>
<td>54</td>
</tr>
<tr>
<td>implicit types</td>
<td>54</td>
</tr>
<tr>
<td>explicit polymorphism</td>
<td>54</td>
</tr>
<tr>
<td>implicit polymorphism</td>
<td>54</td>
</tr>
<tr>
<td>type inference</td>
<td>54</td>
</tr>
<tr>
<td>partially typed</td>
<td>54</td>
</tr>
<tr>
<td>untyped</td>
<td>54</td>
</tr>
<tr>
<td>fully typed</td>
<td>54</td>
</tr>
<tr>
<td>Curry view</td>
<td>55</td>
</tr>
<tr>
<td>Church view</td>
<td>55</td>
</tr>
<tr>
<td>principle types</td>
<td>55</td>
</tr>
<tr>
<td>Milner style</td>
<td>55</td>
</tr>
<tr>
<td>Girard/Reynolds style</td>
<td>56</td>
</tr>
<tr>
<td>partial type inference</td>
<td>55</td>
</tr>
</tbody>
</table>
## Concepts

<table>
<thead>
<tr>
<th>Concept</th>
<th>See page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>simply-typed lambda calculus</td>
<td>6</td>
</tr>
<tr>
<td>λ-abstraction</td>
<td>6</td>
</tr>
<tr>
<td>application</td>
<td>6</td>
</tr>
<tr>
<td>scope</td>
<td>7</td>
</tr>
<tr>
<td>body</td>
<td>7</td>
</tr>
<tr>
<td>bound</td>
<td>7,21</td>
</tr>
<tr>
<td>free</td>
<td>7,21</td>
</tr>
<tr>
<td>type judgment</td>
<td>7</td>
</tr>
<tr>
<td>extension</td>
<td>7</td>
</tr>
<tr>
<td>inference rules</td>
<td>8</td>
</tr>
<tr>
<td>β-reduct,β-reducible,β-reduction,β-conversion</td>
<td>8</td>
</tr>
<tr>
<td>η-reduct,η-reducible,η-reduction,η-conversion</td>
<td>8</td>
</tr>
<tr>
<td>βη-reductible,βη-reduction,βη-conversion</td>
<td>8</td>
</tr>
<tr>
<td>βη-normal form</td>
<td>8</td>
</tr>
<tr>
<td>equivalence</td>
<td>8</td>
</tr>
<tr>
<td>α-equivalent</td>
<td>9</td>
</tr>
<tr>
<td>α-convertible</td>
<td>9</td>
</tr>
<tr>
<td>Strong Normalization Theorem</td>
<td>9</td>
</tr>
<tr>
<td>Church-Rosser Theorem</td>
<td>9</td>
</tr>
<tr>
<td>currying</td>
<td>11</td>
</tr>
<tr>
<td>inductive type definitions</td>
<td>11</td>
</tr>
<tr>
<td>global environment</td>
<td>13</td>
</tr>
<tr>
<td>positive occurrence</td>
<td>13,35</td>
</tr>
<tr>
<td>negative occurrence</td>
<td>35,35</td>
</tr>
<tr>
<td>underscore notation</td>
<td>15</td>
</tr>
<tr>
<td>polymorphism</td>
<td>20</td>
</tr>
<tr>
<td>polymorphic λ-calculus</td>
<td>20</td>
</tr>
<tr>
<td>term abstraction</td>
<td>21</td>
</tr>
<tr>
<td>type abstraction</td>
<td>21</td>
</tr>
<tr>
<td>term application</td>
<td>21</td>
</tr>
<tr>
<td>type application</td>
<td>21</td>
</tr>
<tr>
<td>closed</td>
<td>21</td>
</tr>
<tr>
<td>primitive recursion</td>
<td>29</td>
</tr>
<tr>
<td>type constructor</td>
<td>33</td>
</tr>
<tr>
<td>meta-programming</td>
<td>41</td>
</tr>
<tr>
<td>homogeneous, nonhomogeneous</td>
<td>42</td>
</tr>
</tbody>
</table>
**Example Functions**

<table>
<thead>
<tr>
<th>NAME</th>
<th>PURPOSE</th>
<th>See page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iter</td>
<td>$F^+_1$ iteration on naturals</td>
<td>6,10</td>
</tr>
<tr>
<td>zero</td>
<td>zero element of naturals</td>
<td>6,12,26</td>
</tr>
<tr>
<td>succ</td>
<td>successor function on naturals</td>
<td>6,12,26</td>
</tr>
<tr>
<td>plus</td>
<td>addition of natural numbers</td>
<td>11,12,12,28</td>
</tr>
<tr>
<td>times</td>
<td>multiplication of natural numbers</td>
<td>11,28</td>
</tr>
<tr>
<td>unit</td>
<td>unique object of type Unit</td>
<td>15,24</td>
</tr>
<tr>
<td>true, false</td>
<td>boolean constants</td>
<td>15,24</td>
</tr>
<tr>
<td>pairNat</td>
<td>pair of natural numbers (in $F^+_1$)</td>
<td>16</td>
</tr>
<tr>
<td>fstNat</td>
<td>first of a pair of natural numbers</td>
<td>16</td>
</tr>
<tr>
<td>sndNat</td>
<td>second of a pair of natural numbers</td>
<td>16</td>
</tr>
<tr>
<td>nilNat</td>
<td>empty list of natural numbers (in $F^+_1$)</td>
<td>16</td>
</tr>
<tr>
<td>consNat</td>
<td>cons of a natural number and a list of naturals</td>
<td>16</td>
</tr>
<tr>
<td>carNat</td>
<td>car of a list of naturals</td>
<td>16</td>
</tr>
<tr>
<td>cdrNat</td>
<td>cdr of a list of naturals</td>
<td>16</td>
</tr>
<tr>
<td>pairListNat</td>
<td>pair of lists of naturals</td>
<td>16</td>
</tr>
<tr>
<td>empty</td>
<td>polymorphic leaf node of a tree</td>
<td>17</td>
</tr>
<tr>
<td>node</td>
<td>polymorphic internal node of a tree</td>
<td>17</td>
</tr>
<tr>
<td>build</td>
<td>polymorphic function to construct trees</td>
<td>18</td>
</tr>
<tr>
<td>id</td>
<td>polymorphic identity</td>
<td>19</td>
</tr>
<tr>
<td>double</td>
<td>polymorphic double application</td>
<td>19</td>
</tr>
<tr>
<td>iterBool</td>
<td>iteration on booleans</td>
<td>25</td>
</tr>
<tr>
<td>not</td>
<td>boolean negation</td>
<td>24,26</td>
</tr>
<tr>
<td>zero?</td>
<td>test for zero on natural numbers</td>
<td>28</td>
</tr>
<tr>
<td>and, or</td>
<td>boolean conjunction/disjunction</td>
<td>26</td>
</tr>
<tr>
<td>iterNat</td>
<td>iteration on anturals</td>
<td>27</td>
</tr>
<tr>
<td>primrec</td>
<td>primitive recursion</td>
<td>30</td>
</tr>
<tr>
<td>pair</td>
<td>polymorphic pairing</td>
<td>36</td>
</tr>
<tr>
<td>fst</td>
<td>polymorphic first of a pair</td>
<td>36</td>
</tr>
<tr>
<td>snd</td>
<td>polymorphic second of a pair</td>
<td>36</td>
</tr>
<tr>
<td>nil</td>
<td>polymorphic empty list</td>
<td>38</td>
</tr>
<tr>
<td>cons</td>
<td>polymorphic list constructor</td>
<td>38</td>
</tr>
<tr>
<td>car</td>
<td>polymorphic list car</td>
<td>38</td>
</tr>
<tr>
<td>cdr</td>
<td>polymorphic list cdr</td>
<td>38</td>
</tr>
<tr>
<td>rep</td>
<td>representation function for meta-programming</td>
<td>41</td>
</tr>
<tr>
<td>app</td>
<td>meta-level term application</td>
<td>41</td>
</tr>
<tr>
<td>eval</td>
<td>evaluation function for metaprogramming</td>
<td>43</td>
</tr>
<tr>
<td>lam</td>
<td>meta-level $\lambda$-abstraction</td>
<td>44</td>
</tr>
<tr>
<td>typapp</td>
<td>meta-level type application</td>
<td>45</td>
</tr>
<tr>
<td>typlam</td>
<td>meta-level type abstraction</td>
<td>45</td>
</tr>
<tr>
<td>typapp1, typapp2</td>
<td>$F_2$ and $F_3$ type application in $F_4$</td>
<td>46</td>
</tr>
<tr>
<td>typlam1, typlam2</td>
<td>$F_2$ and $F_3$ type abstraction in $F_4$</td>
<td>46</td>
</tr>
</tbody>
</table>
Type and Kind Deduction Rules

<table>
<thead>
<tr>
<th>RULE NAME</th>
<th>PURPOSE</th>
<th>SEE PAGE(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Succ</td>
<td>check type of use of succ in $F_1^+$</td>
<td>8</td>
</tr>
<tr>
<td>Zero</td>
<td>check type of use of zero in $F_1^+$</td>
<td>8</td>
</tr>
<tr>
<td>Iter</td>
<td>check type of use of iter in $F_1^+$</td>
<td>8</td>
</tr>
<tr>
<td>Var</td>
<td>checks that the type of a variable is of kind *</td>
<td>8,22,51</td>
</tr>
<tr>
<td>→I</td>
<td>→ introduction in type of a term</td>
<td>8,22,51</td>
</tr>
<tr>
<td>→E</td>
<td>→ elimination in type of a term</td>
<td>8,22,51</td>
</tr>
<tr>
<td>Tvar</td>
<td>checks the kind of a type variable</td>
<td>22,51</td>
</tr>
<tr>
<td>ENV-()</td>
<td>asserts than an empty environment is well-formed</td>
<td>22,51</td>
</tr>
<tr>
<td>ENV-term</td>
<td>checks well-formedness of term/type pair in an environment</td>
<td>22,51</td>
</tr>
<tr>
<td>ENV-type</td>
<td>checks well-formedness of type/kind pair in an environment</td>
<td>22,51</td>
</tr>
<tr>
<td>WF-→</td>
<td>checks that → types are well-formed</td>
<td>22,51</td>
</tr>
<tr>
<td>WF-Δ</td>
<td>checks that Δ types are well-formed</td>
<td>22,51</td>
</tr>
<tr>
<td>ΔI</td>
<td>Δ introduction in type of a term</td>
<td>22,51</td>
</tr>
<tr>
<td>ΔE</td>
<td>Δ elimination in type of a term</td>
<td>22,51</td>
</tr>
<tr>
<td>WF-λ</td>
<td>checks that λ types are well-formed</td>
<td>51</td>
</tr>
<tr>
<td>WF-app</td>
<td>checks that a type-to-type application is well-formed</td>
<td>51</td>
</tr>
<tr>
<td>(≈)</td>
<td>$\beta\eta$-conversion for type expressions</td>
<td>51</td>
</tr>
</tbody>
</table>

Example Types

<table>
<thead>
<tr>
<th>NAME</th>
<th>PURPOSE</th>
<th>SEE PAGE(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nat</td>
<td>natural numbers</td>
<td>6,12,26</td>
</tr>
<tr>
<td>Bool</td>
<td>booleans</td>
<td>15,24</td>
</tr>
<tr>
<td>Void</td>
<td>type without any elements</td>
<td>15,23</td>
</tr>
<tr>
<td>Unit</td>
<td>type with exactly one element modulo $\alpha$-equivalence</td>
<td>15,24</td>
</tr>
<tr>
<td>PairNat</td>
<td>pair of natural numbers</td>
<td>16</td>
</tr>
<tr>
<td>ListNat</td>
<td>list of natural numbers</td>
<td>16</td>
</tr>
<tr>
<td>PairListNat</td>
<td>pair of lists of natural numbers</td>
<td>17</td>
</tr>
<tr>
<td>Tree</td>
<td>polymorphic tree</td>
<td>17</td>
</tr>
<tr>
<td>Pair</td>
<td>polymorphic pairs</td>
<td>36</td>
</tr>
<tr>
<td>List</td>
<td>polymorphic lists</td>
<td>38</td>
</tr>
<tr>
<td>Term</td>
<td>$\lambda$-calculus terms</td>
<td>41</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>(\beta)</td>
<td>(\beta)-reduction</td>
<td>8</td>
</tr>
<tr>
<td>(\eta)</td>
<td>(\eta)-reduction</td>
<td>8</td>
</tr>
<tr>
<td>(\beta\eta)</td>
<td>(\beta\eta)-equivalence</td>
<td>8</td>
</tr>
<tr>
<td>(\text{iterT}[V])</td>
<td>iteration</td>
<td>12</td>
</tr>
<tr>
<td>(\Lambda\alpha:K.T)</td>
<td>type-to-term abstraction</td>
<td>19, 21, 34, 50</td>
</tr>
<tr>
<td>(\ast)</td>
<td>the kind of types</td>
<td>21, 34, 50</td>
</tr>
<tr>
<td>(\Delta\alpha:K.T)</td>
<td>universal type</td>
<td>20, 21, 34, 50</td>
</tr>
<tr>
<td>(\text{e}[\alpha])</td>
<td>term-to-type application</td>
<td>21, 34, 50</td>
</tr>
<tr>
<td>(F_1^i)</td>
<td>(F_1) with inductive type definitions</td>
<td>12</td>
</tr>
<tr>
<td>(F_2)</td>
<td>second-order typed (\lambda)-calculus</td>
<td>21</td>
</tr>
<tr>
<td>(F_2^i)</td>
<td>(F_2) with inductive type definitions</td>
<td>33, 41</td>
</tr>
<tr>
<td>(F_3)</td>
<td>third-order typed (\lambda)-calculus</td>
<td>34</td>
</tr>
<tr>
<td>(\Pi[\alpha/K])</td>
<td>extension of (\Pi) with the kind of a type variable</td>
<td>35</td>
</tr>
<tr>
<td>(\Pi \vdash T \in K)</td>
<td>kind judgment</td>
<td>35</td>
</tr>
<tr>
<td>(\lambda\alpha:K.T)</td>
<td>type-to-type abstraction</td>
<td>34, 50</td>
</tr>
<tr>
<td>(T \vdash T')</td>
<td>type-to-type application</td>
<td>34, 50</td>
</tr>
<tr>
<td>(K \rightarrow K')</td>
<td>kind of a type-to-type abstraction</td>
<td>34</td>
</tr>
<tr>
<td>([\alpha]\Pi)</td>
<td>syntactic sugar for meta-representation</td>
<td>42</td>
</tr>
<tr>
<td>(F_4)</td>
<td>fourth-order typed (\lambda)-calculus</td>
<td>46</td>
</tr>
<tr>
<td>(F_\omega)</td>
<td>(\omega)-order typed (\lambda)-calculus</td>
<td>50</td>
</tr>
<tr>
<td>(F_\omega^i)</td>
<td>(F_\omega) with inductive type definitions</td>
<td>50</td>
</tr>
<tr>
<td>(\approx)</td>
<td>(\beta\eta)-equivalence of type expressions</td>
<td>50</td>
</tr>
<tr>
<td>(\text{e}[\alpha/T'])</td>
<td>type substitution</td>
<td>50</td>
</tr>
<tr>
<td>LEAP</td>
<td>the LEAP programming language</td>
<td>56</td>
</tr>
</tbody>
</table>
Appendix B

Symbols and Terminology

Notational Conventions

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>USAGE</th>
<th>STYLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>x, y, map, car, ...</td>
<td>term variables and constants</td>
<td>l.c.</td>
</tr>
<tr>
<td>List, Bool, Nat, ...</td>
<td>globally-defined types</td>
<td>u.c.</td>
</tr>
<tr>
<td>α, β, θ, σ, ...</td>
<td>bound type variables of kind *</td>
<td>l.c. greek</td>
</tr>
<tr>
<td>Φ, Ψ, Θ</td>
<td>bound type variables of higher kind</td>
<td>u.c. greek</td>
</tr>
<tr>
<td>γ</td>
<td>result type of an iteration</td>
<td></td>
</tr>
<tr>
<td>Γ</td>
<td>result type of a higher-kind iteration</td>
<td></td>
</tr>
<tr>
<td>pair, cons, succ</td>
<td>globally-defined constructors</td>
<td>l.c. spelled-out</td>
</tr>
<tr>
<td>p, c, n, s</td>
<td>λ-bound placeholders for constructors</td>
<td>l.c. first letter</td>
</tr>
<tr>
<td>K</td>
<td>kind variable</td>
<td></td>
</tr>
</tbody>
</table>

Symbols

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>MEANING</th>
<th>SEE PAGE(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>≡</td>
<td>global definitions</td>
<td>6</td>
</tr>
<tr>
<td>λx:T.e</td>
<td>term-to-term abstraction</td>
<td>6,6,12,21,34,50</td>
</tr>
<tr>
<td>ee'</td>
<td>term-to-term application</td>
<td>6,6,21,34,50</td>
</tr>
<tr>
<td>T→T'</td>
<td>type of a term-to-term abstraction</td>
<td>6,6,12,21,34</td>
</tr>
<tr>
<td>F₁</td>
<td>the simply-typed λ-calculus</td>
<td>6</td>
</tr>
<tr>
<td>F₁⁺</td>
<td>F₁ with primitive types and iteration</td>
<td>6</td>
</tr>
<tr>
<td>Π[α/β]</td>
<td>extension of Π with the type of a variable</td>
<td>7</td>
</tr>
<tr>
<td>Π ⊢ e ∈ T</td>
<td>type judgment</td>
<td>7</td>
</tr>
<tr>
<td>=β</td>
<td>β-equivalence</td>
<td>8</td>
</tr>
</tbody>
</table>
betalam \equiv \lambda t:\text{Term}. \lambda u:\text{Term}.
\begin{array}{l}
\text{fst (iterTerm[Pair_Term] t }
\begin{array}{l}
(\lambda x:\text{Var}.
\begin{array}{l}
\text{pair (app t u)}
\end{array}
\begin{array}{l}
(\text{var } x)
\end{array})
\begin{array}{l}
(\lambda r:\text{Pair_Term}. \lambda s:\text{Pair_Term}.
\begin{array}{l}
\text{pair (app t u)}
\end{array}
\begin{array}{l}
(\text{app (snd r) (snd s)})
\end{array})
\begin{array}{l}
(\lambda x:\text{Var}. \lambda r:\text{Pair_Term}.
\begin{array}{l}
\text{pair (subst (snd r) x u)}
\end{array}
\begin{array}{l}
(\text{lam x (snd r)})
\end{array})
\end{array}
\end{array}
\end{array}
\end{array}
\begin{array}{l}
\end{array}
\end{array}
\end{array}

This can easily be extended to perform any finite number of "simultaneous" \( \beta \)-reductions:

\text{multibeta} \equiv \lambda n:\text{Nat}. \lambda t:\text{Term}.
\begin{array}{l}
\text{iterNat[Term] n t beta}
\end{array}
\end{array}
\end{array}
\end{array}

\text{Exercises A.2:}

1. Define an encoding of type expressions in F\(_1\) as an inductive type.
2. Extend the definition of Term above to a representation of F\(_1\), where types appear explicitly in the data structure (rather than being represented as types in the metalanguage as in Chapter 5).
3. Define substitution and \( \beta \)-reduction for this representation of F\(_1\).
4. (Difficult.) Define a typechecker for this representation.
\[\text{alphaconv} \equiv \lambda t : \text{Term. } \lambda d : \text{Nat.} \]
\[\quad (\text{iterTerm}[(\text{Var} \mapsto \text{Bool}) \mapsto \text{Term}] t)\]
\[\quad (\lambda x : \text{Var. } \lambda b : \text{Var} \mapsto \text{Bool.} \]
\[\quad \text{ife } (b \ x) \]
\[\quad \quad (\text{var} (\text{plus} \ x \ d)) \]
\[\quad \quad (\text{var} x))\]
\[\quad (\lambda r : (\text{Var} \mapsto \text{Bool}) \mapsto \text{Term. } \lambda s : (\text{Var} \mapsto \text{Bool}) \mapsto \text{Term.} \]
\[\quad \lambda b : \text{Var} \mapsto \text{Bool.} \]
\[\quad \text{app } (r \ b) \ (s \ b)) \]
\[\quad (\lambda x : \text{Var. } \lambda r : (\text{Var} \mapsto \text{Bool}) \mapsto \text{Term. } \lambda b : \text{Var} \mapsto \text{Bool.} \]
\[\quad \text{lam } (\text{plus} \ x \ d) \]
\[\quad \quad (r \ (\lambda z : \text{Var.} \]
\[\quad \quad \text{ife } (= z x) \]
\[\quad \quad \quad \text{true} \]
\[\quad \quad \quad (b z)))) \]
\[\quad ) \ (\lambda x : \text{Var. } \text{false})\]

With substitution under control, it is now fairly easy to define \(\beta\)-reduction on terms. Of course, we cannot hope to completely normalize an arbitrary term of the untyped \(\lambda\)-calculus since any attempt to do so may diverge, which would contradict the strong normalization property of the metalanguage. The following pair of functions implements one-step \(\beta\)-reduction of all redices in a term, from bottom to top. (It should also be possible to reduce just one redex at a time, as defined in Section 2.1, but this seems harder than reducing several at once.)

The iteration in the first function does not alter the \text{Term} except at \text{app} nodes, which it hands off to the second function. This, in turn, does nothing unless the top-level constructor of the term on the left hand side of the application is a \(\lambda\) abstraction, in which case it uses \text{subst} to reduce the application to a substitution instance of the body of the \(\lambda\).

\[\text{beta} \equiv \lambda t : \text{Term.} \]
\[\quad \text{iterTerm}[\text{Term}] t \]
\[\quad (\lambda x : \text{Var. } \text{var } x) \]
\[\quad (\lambda r : \text{Term. } \lambda s : \text{Term.} \]
\[\quad \quad \text{beta} \lambda r \ s) \]
\[\quad (\lambda x : \text{Var. } \lambda r : \text{Term.} \]
\[\quad \quad \text{lam } x \ r)\]
\[
\text{subst}_1 \equiv \lambda t:\text{Term}. \lambda v:\text{Var}. \lambda s:\text{Term}. \\
\text{fst}\left(\text{iterTerm}[\text{Pair}\_\text{Term}]\ t\right) \\
\quad (\lambda x:\text{Var}. \\
\quad \text{pair} \\
\quad \quad (\text{ife} (= x v) s (\text{var} x)) \\
\quad \quad (\text{var} x)) \\
\quad (\lambda r:\text{Pair}\_\text{Term}. \lambda s:\text{Pair}\_\text{Term}. \\
\quad \text{pair} \\
\quad \quad (\text{app} (\text{fst} r) (\text{fst} s)) \\
\quad \quad (\text{app} (\text{snd} r) (\text{snd} s))) \\
\quad (\lambda x:\text{Var}. \lambda r:\text{Pair}\_\text{Term}. \\
\quad \text{pair} \\
\quad \quad (\text{ife} (= x v) \\
\quad \quad \quad (\text{lam} x (\text{snd} r)) \\
\quad \quad \quad (\text{lam} x (\text{fst} r))) \\
\quad \quad (\text{lam} x (\text{snd} r))) 
\)
\]

This version is not quite correct, however: it allows capture of free variables in \(s\) by bound variables in \(t\). To use it safely, we need to first make sure that no such capture can occur, by \(\alpha\)-converting \(t\) so that its bound variables all have greater indices than any variable used in \(s\).

**Exercise A.1:** Define a function \(\text{maxvar} : \text{Term} \rightarrow \text{Var}\) that calculates the maximum index of the variables used in its argument.

Taking \(\text{maxvar}\) as given, we can write:

\[
\text{subst} \equiv \lambda t:\text{Term}. \lambda v:\text{Var}. \lambda s:\text{Term}. \\
\text{subst}_1 (\text{alphaconv} t (\text{maxvar} s)) v s
\]

To define \(\text{alphaconv}\), we need to be able to keep track of the set of variables bound by enclosing \(\lambda s\) so that we can decide whether or not to change the index of a variable when we come to it. But the iteration construct makes this hard: since it starts at the leaves of the term and builds up the result, we see the variable at the leaf “before” we get to any enclosing \(\lambda s\). The trick is to have the iteration return not a term, but rather a function from variables to terms. (“You show me what the bound variables are and I’ll show you what the \(\alpha\)-converted term is.”)
Appendix A

Representing the Untyped \(\lambda\)-Calculus

This appendix presents a representation of the untyped \(\lambda\)-calculus in \(F_2\). Because it is homogeneous (in the sense discussed in Section 5.1) the untyped \(\lambda\)-calculus should be easier to represent than the typed \(\lambda\)-calculi. This makes it a good starting point for developing techniques, tools, and intuitions that may perhaps be applied to the more difficult situations. It also provides a good extended programming example.

We consider the simplest formulation of the untyped \(\lambda\)-calculus, with just variables, \(\lambda\)-abstractions, and applications:

\[
\text{indtype Term:* with} \\
\text{var: Var \rightarrow Term} \\
\text{app: Term \rightarrow Term \rightarrow Term} \\
\text{lam: Var \rightarrow Term \rightarrow Term}
\]

where natural numbers are used to represent variables:

\[
\text{Var} \equiv \text{Nat}
\]

Our first task is to implement substitution. This turns out to require primitive recursion rather than simple iteration, because we need to be able to inhibit substitution in the scope of a \(\lambda\) whose bound variable is the same as the one being substituted.

The iterator \text{iterTerm}[\text{Pair Term}] takes a Term to a pair of Terms. The first projection of the resulting pair is the substituted term; the second projection is the original term. (In the following, we omit some type tags when it is clear from context how they should be filled in. For example, write ife instead of ife Term.)
of empty type applications are inserted after it. These empty applications are then filled by the partial type inference algorithm, as before.
• As discussed in the previous section, a good type inference system can enormously reduce the redundancy and verbosity of programs in polymorphic λ-calculi.

• Most functional languages include some form of general recursive function definitions. Although we have left it out of our formulation of $F_\omega$—partly to highlight the interesting programming style that arises without it and partly to preserve the theoretical properties of Girard’s system—there is no reason why it cannot be added.

• Some “functional-style” languages also include imperative features like updateable references and powerful control constructs like exceptions or a “call with current continuation” operator. It appears possible to add these to $F_\omega$ as well.

Pfenning and Lee’s work is currently focused on a family of languages called LEAP (a “Language with Eval and Polymorphism”). In addition to some of the ideas mentioned above, key features include:

• An extension of $F_\omega$’s polymorphism that makes it possible to write an eval function for an inductive representation of all of $F_\omega$ (something that does not appear to be possible in $F_\omega$ itself). Global definitions in LEAP are not considered to be simple abbreviations (like the ones in this tutorial), but rather global let bindings. In particular, kind variables are allowed to appear free in the right hand side of a global definition, and these variables are considered generic in exactly the sense of ML’s generic type variables introduced by let statements [15]—they can be instantiated to different kinds at each point where the global definition is used. This introduces enough “additional polymorphism" to allow the definition of an eval function for a representation of terms essentially the same as that given in Section 5.3:

\[
\begin{align*}
\text{eval} & \equiv \Lambda\alpha:*. \lambda t:\text{Term}\,\alpha. \\
& \quad \text{iterTerm}[\lambda\sigma:*.*] t \\
& \quad (\Lambda\sigma:*.*. \lambda u:\sigma. u) \\
& \quad (\lambda t:*.*. \lambda u:\sigma\rightarrow\tau. \lambda v:\sigma. uv) \\
& \quad (\lambda t:*.*. \lambda u:\sigma\rightarrow\tau. u) \\
& \quad (\lambda\Theta:K\rightarrow*. \lambda u:(\Delta\alpha:K. \Theta\alpha). u) \\
& \quad (\lambda\Theta:K'\rightarrow*. \lambda u:(\Delta\alpha:K'. \Theta\alpha).\Delta\alpha. u[\alpha])
\end{align*}
\]

Every time eval is used in a program, the appropriate values of $K$ and $K'$ are computed by type/kind inference. Looking at the earlier definition of eval for $F_\lambda$ in terms of $F_\delta$, it is clear that the above is a schema for a definition that would otherwise have to be infinite.

• Even with a good partial type inference algorithm, the amount of syntactic “noise" introduced by placeholders for type applications can be substantial. Pfenning and Lee have introduced a very useful shorthand, which they call “star syntax." When an identifier is declared on the left hand side of a let, it may be annotated with some number of *'s to indicate how many type parameters it expects. Now when the defined variable is used (unstarred) elsewhere in the program, the correct number
"Girard/Reynolds" polymorphism of our calculi does not so constrain types. Milner's restriction disallows the passing of polymorphic functions as parameters, since nested $\Delta$'s are required to represent the type of the polymorphic argument. In fact, it is this restriction which makes the way for principle types. Consider the function double of Section 3. From an untyped version
\[
\lambda f. \lambda x. f(f \, x)
\]
type inference might derive either the original
\[
\lambda \alpha:* . \lambda f: \alpha \rightarrow \alpha . \lambda x: \alpha. \ f(\ f \ x)
\]
of type
\[
\Delta \alpha . (\alpha \rightarrow \alpha) \rightarrow \alpha\rightarrow \alpha
\]
or instead
\[
\lambda f: (\Delta \alpha:* . \alpha \rightarrow \alpha) . \lambda \beta:* . \lambda x: \beta . f \, [\beta] \, (f \, [\beta] \, x)
\]
which has type
\[
(\Delta \alpha:* . (\alpha \rightarrow \alpha)) \rightarrow \Delta \beta:* . \beta \rightarrow \beta
\]
The second version takes a polymorphic function as an argument, and so is not permitted in a Milner style. That the above types are incomparable means that the richness of Girard/Reynolds polymorphism does not admit principle types. Furthermore, the decidability of type inference mapping untyped $\lambda$-calculus into $F_2$—that is, full type inference for $F_2$—remains open [33,39].

The above ambiguity for the untyped double may be removed by including empty type applications in the appropriate positions, thereby steering partial type inference toward the desired typing. However, this still does not yield principle types. The expressiveness inherent in the higher-order nature of Girard/Reynolds polymorphism complicates type inference: that types may be functions requires higher-order unification to determine the intended types. In fact, partial type inference [3,41] may be more expensive than full type inference. Pfenning has shown that partial type inference for the $n^{th}$-order $\lambda$-calculus (in which type applications are omitted but placeholders are left to show where they must appear) is equivalent to $n^{th}$-order unification, which is undecidable for $n \geq 2$ [21]. However, Huet's [30] algorithm has proven tractable for realistic problems, and using this, Pfenning describes a partial type inference semi-algorithm for $F_\omega$ [46].

### 6.4 $F_\omega$ as the Basis for a Programming Language

We have shown in this tutorial that $F_\omega$ can be used to express a surprisingly broad range of computations. Still, it falls short in many ways as a practical programming language. Pfenning and Lee [49] have studied a number of extensions to $F_\omega$ that either make it more convenient without adding to its power or disrupting its desirable theoretical properties, or strictly increase its power:
\((\lambda n_2. (\text{succ } n_2) \; [\;] \; 1 \; \text{succ}) \; 3)\)

The task of expanding a partially-typed, or "type-elided", term into \(F_{\omega}\) is called \textit{partial} type inference. Enough type information must be included, however, so that "proper" typings are recreated in the process. (This is further developed below.)

\textbf{Curry vs. Church Types.} This leads us to a more fundamental difference between typed languages: What is the role of types? The first or "Curry" view of types is that terms are initially untyped and types serve merely to group (structurally or behaviorally) related terms [13]. Central to the Curry view is that untyped terms are meaningful and may be given a semantics without regard to any associated type. In fact, syntactic and semantic properties of the language (e.g., reduction) are formulated without types. Generally under such an approach, types semantically characterize terms, perhaps by denoting components of a mathematical model (e.g., a set). Types which \textit{do} appear in the syntax of the language are viewed as meta-syntactic; the ':' is interpreted semantically (e.g., a predicate establishing membership within a set). Type inference is viewed as the process of determining where a term lives, that is, what is the class (type) of its semantic denotation.

In contrast, the "Church" view treats untyped terms as meaningless [10]. Types, in the Church view, are an integral part of the language. Indeed, the reduction rules are formulated with types. Similarly, type inference is defined syntactically. Each of the typed \(\lambda\)-calculi we have described fall into this school. Of course like Curry, the Church approach admits denotational semantics, but whereas a typical "Curry semantics" consists of one large domain containing the denotations of all terms, in a "Church semantics" there will be a family of domains indexed by types.

In the literature, the reader may find the Curry view associated with implicit types and the Church view with explicit. This is because of the importance of untyped terms within the Curry approach and of typed terms under Church. However, we wish to emphasize that they are independent notions: we might imagine a concrete syntax containing types which are all viewed meta-syntactically (Curry), or a concrete syntax initially free of types, but for which type inference generates a syntactically complete, fully typed term (Church). Curry vs. Church is primarily a philosophical issue, while explicit vs. implicit is a question of language syntax and implementation. (For further insight on these issues, see Mitchell and Harper [40].)

\textbf{Principle Types and Milner vs. Girard/Reynolds Polymorphism.} We have suggested that a type inference algorithm assigns at most one fully-typed term for a given partially or untyped one. However, this is usually not possible. Existing implicitly polymorphic languages (notably ML) admit complete type inference algorithms [5,38] that yield "principle" rather than unique types. A \textit{principle} type is a type (or type scheme) from which all other valid typings may be derived by substitution.

The "Milner style" polymorphism of ML limits occurrences of universal quantification (our \(\Delta\)) to the top-level or outermost scope of the type. In contrast, the more expressive
6.3 Types and Type Inference

In order to fit $F_\omega$ into the "big picture," we consider its relation to other programming languages along several dimensions.

**Implicit vs. Explicit Types.** We have remarked that $F_\omega$ is *explicitly* rather than *implicitly* typed: polymorphism is represented by explicit type abstraction ($\lambda$), and polymorphic functions must be applied explicitly to a type argument ($\Pi$) yielding a monomorphic instance, which can in turn be applied to term arguments (or further type arguments). In contrast, types do not appear in the concrete syntax of a purely implicitly typed language (although they may eventually be derived). We use the terms *explicit polymorphism* (or explicit typing) and *implicit polymorphism* to denote polymorphism in an explicitly or implicitly typed language, respectively.

Explicit typing offers several advantages over implicit in that types may be given a greater role than just ensuring the well-formedness of terms. Although an implicitly typed program may be typable (i.e., well-formed), it may not be the one the author intended. Explicit typing adds an additional level of insurance that the writer's intentions are realized. Explicit types also serve as formal documentation and can increase the readability of programs.

In reality, explicitly and implicitly typed languages represent a continuum: at one end is a language with no types in its concrete syntax (purely implicit); at the other, a language in which every term has an associated syntactic type (purely explicit). Although we have classified $F_\omega$ as explicitly typed, not all terms have explicit types in the concrete syntax. In fact, it is only the variables bound by $\lambda$-abstractions that are given explicit types. Nevertheless, $F_\omega$ remains explicitly typed because the types of the remaining terms may easily be deduced from the type information provided (see Section 6.1). In general, *type inference* is the process of determining missing type information. Although this involves only minimal work in the case of $F_\omega$, type inference is also applicable to implicitly typed languages in which more or all of the type information is missing. Terms before type inference may be *partially-typed* (under explicit polymorphism) or *untyped* (under implicit polymorphism), while the results of successful type inference are *fully typed*.

A downside of explicit typing is that programs may become so verbose as to be unintelligible. In our development of $\text{ack}$ (Section 3.3) one of the steps appeared as follows:

$$((\lambda n_3 : \text{Nat.} \ (\text{succ } n_3) [\text{Nat}] \ 1 \ \text{succ})$$

$$((\lambda n_2 : \text{Nat.} \ (\text{succ } n_2) [\text{Nat}] \ 1 \ \text{succ})$$

$$3))$$

The problem with writing the above expression is that each instance of $\text{Nat}$ may be determined by context: the iterations produce $\text{Nats}$ because their result is a combination of $1$ and $\text{succ}$, while the iterations are over $\text{Nats}$ because $\text{succ}$ is applied first. A remedy to the redundancy of explicit types is to omit type information when it may be inferred. The equivalent term without the redundant type information is

$$((\lambda n_3 \ . \ (\text{succ } n_3) [\ ] \ 1 \ \text{succ})$$
Exercises 6.1.2:
Use the deduction rules for $F_\omega$ to prove that

$$\text{cons} \ [\text{Nat}] \ 1 \ (\text{nil} \ [\text{Nat}] \ 0) \in \text{List Nat}$$

(Warning: This will take quite a lot of work.)

The languages $F_1$, $F_2$, etc. can now be defined as restrictions of $F_\omega$.

Definition 6.1.3: The kind $*$ has order 1. If the greater of the orders of $K$ and $K'$ is $i$, then $K\rightarrow K'$ has order $i + 1$.

The $n^{th}$-order polymorphic $\lambda$-calculus ($F_n$) consists of those terms of $F_\omega$ for which types can be derived using the above rules without mentioning any kinds of order greater than or equal to $n$.

6.2 Properties of $F_\omega$

We have discussed various properties of the languages in our hierarchy of typed $\lambda$-calculi. To summarize, we can state the following theorems for $F_\omega$:\(^1\)

Theorem 6.2.1:

1. If $\Pi \vdash e : \alpha$ then $\Pi \vdash \alpha : *$.
2. If $\Pi \vdash T : K$ then $T$ has a unique $\beta\eta$-normal form.
3. If $\Pi \vdash e : T$ then $e$ has a unique $\beta\eta$-normal form.
4. $\Pi \vdash e : T$ is decidable.

Some other interesting properties of $F_\omega$ are:

- $F_\omega^{\text{id}} = F_\omega$. (Inductive data type definitions whose constructors have types in $F_\omega$ can be translated into closed type expressions in $F_\omega$ [45].)

- $F_2$ can express every function which is provable total under second-order peano arithmetic [18,16]. In general, $F_n$ can express all functions whose totality is provable in $n^{th}$-order arithmetic [19].

- Typed $\lambda$-terms can be extracted from proofs in higher-order logic [11,47,43]. A program extracted from a proof in $n^{th}$-order logic will be typable in $F_n$.

---

\(^1\)Strictly speaking, these are conjectures: they have not been proved for the formulation of $F_\omega$ that we are using. Our system is intended not as an object of study in itself but as a concrete way of talking about other systems. For Girard's system [20,18], the appropriate analogues of these statements are theorems.
\( t_5 \equiv \lambda x:\Theta \alpha.\ x \)
\( t_4 \equiv \Lambda \alpha:*.\ t_5 \)
\( t_3 \equiv \Lambda \Theta:*.->.\ t_4 \)
\( t_2 \equiv t_3 [\lambda \sigma:*.\sigma] \)
\( t_1 \equiv t_2 [\text{Nat}] \)

and hence:
\( t \equiv t_1 \ 5 \)

At each step of the proof we give the name of the inference rule used, together with the numbers of the lines of the proof used as premises (in order). When one line follows from another by application of the above abbreviations we say that it follows “by definition.” Additionally, for brevity we allow ourselves to assume that \( \text{Nat} \in * \) and that \( 5 \in \text{Nat} \), and we omit the sections of the proof dealing with the well-formedness of environments.

(1) \( \langle \Theta, *->*, (\alpha, *) \rangle \vdash \alpha \in * \) by (Tvar)
(2) \( \langle \Theta, *->*, (\alpha, *) \rangle \vdash \Theta \in *->* \) by (Tvar)
(3) \( \langle \Theta, *->*, (\alpha, *) \rangle \vdash \Theta \alpha \in * \) by (WF-app) from 2,1
(4) \( \langle \Theta, *->*, (\alpha, *), (x, \Theta \alpha) \rangle \vdash \alpha \in * \) by (Tvar)
(5) \( \langle \Theta, *->*, (\alpha, *), (x, \Theta \alpha) \rangle \vdash \Theta \in *->* \) by (Tvar)
(6) \( \langle \Theta, *->*, (\alpha, *), (x, \Theta \alpha) \rangle \vdash \Theta \alpha \in * \) by (WF-app) from 5,4
(7) \( \langle \Theta, *->*, (\alpha, *), (x, \Theta \alpha) \rangle \vdash x \in \Theta \alpha \) by (Var) from 6
(8) \( \langle \Theta, *->*, (\alpha, *), \lambda x:\Theta \alpha.x \Theta \alpha \in \Theta \alpha->\Theta \alpha \) by (\( \rightarrow \)I) from 3,7
(9) \( \langle \Theta, *->*, (\alpha, *) \rangle \vdash t_5 \in \Theta \alpha->\Theta \alpha \) by definition from 8
(10) \( \langle \Theta, *->*, (\alpha, *) \rangle \vdash \Delta \alpha:*.\ t_5 \in \Delta \alpha:*.\Theta \alpha->\Theta \alpha \) by (\( \Delta \)I) from 9
(11) \( \langle \Theta, *->* \rangle \vdash t_4 \in \Delta \alpha:*.\Theta \alpha->\Theta \alpha \) by definition from 10
(12) \( \vdash \Delta \Theta:*.->*.\ t_4 \in \Delta \Theta:*.->*.\Delta \alpha:*.\Theta \alpha->\Theta \alpha \) by (\( \Delta \)I) from 11
(13) \( \vdash t_3 \in \Delta \Theta:*.->*.\Delta \alpha:*.\Theta \alpha->\Theta \alpha \) by definition from 12
(14) \( \langle \sigma, * \rangle \vdash \sigma \in * \) by (Tvar)
(15) \( \vdash \lambda \sigma:*.\sigma \in *->* \) by (WF-\( \lambda \)) from 14
(16) \( \vdash t_3 [\lambda \sigma:*.\sigma] \in \Delta \alpha:*.\ (\lambda \sigma:*.\sigma)\alpha->(\lambda \sigma:*.\sigma)\alpha \) by (\( \Delta \)E) from 13,15
(17) \( \langle \alpha, * \rangle \vdash \alpha \in * \) by (Tvar)
(18) \( \langle \alpha, * \rangle \vdash \alpha->\alpha \in * \) by (WF-\( \rightarrow \)) from 17,17
(19) \( \vdash \Delta \alpha:*.\alpha->\alpha \approx \Delta \alpha:*.\ (\lambda \sigma:*.\sigma)\alpha->(\lambda \sigma:*.\sigma)\alpha \) by \( \beta\eta \)-conversion of types
(20) \( \vdash \Delta \alpha:*.\alpha->\alpha \in * \) by (WF-\( \Delta \)) from 18
(21) \( \vdash t_3 [\lambda \sigma:*.\sigma] \in \Delta \alpha:*.\alpha->\alpha \) by (\( \approx \)) from 16,19,20
(22) \( \vdash t_2 \in \Delta \alpha:*.\alpha->\alpha \) by definition from 21
(23) \( \vdash \text{Nat} \in * \) by assumption
(24) \( \vdash t_2 [\text{Nat}] \in \text{Nat} \rightarrow \text{Nat} \) by (\( \Delta \)E) from 22,23
(25) \( \vdash t_1 \in \text{Nat} \rightarrow \text{Nat} \) by definition from 24
(26) \( \vdash 5 \in \text{Nat} \) by assumption
(27) \( \vdash t_1 \ 5 \in \text{Nat} \) by (\( \Delta \)E) from 26,25
(28) \( \vdash t \in \text{Nat} \) by definition from 27
type variable \( \alpha \) in \( T' \) with \( T \), renaming bound variables in \( T' \) to avoid capture of free variables in \( T \). In both cases it is necessary to respect the binding constructs \( \lambda \) and \( \Delta \) in a type expression. Note that the rules are designed so that for any term \( e \), where \( e \in T \) with \( T \) some type expression, it is assured that \( T \) is actually of kind \( * \). This is usually left to the \((\text{Var})\) rule, but in the case of the \((\approx)\) rule it must be written explicitly because a new type expression \( T \) is being introduced, which might not be of the same kind as \( T' \) although the two are \( \beta\eta \)-convertible. Similarly, in the case of \((-I)\) a new type expression is being introduced, so we must check explicitly that it is of kind \( * \). We adopt the convention that any two \( \alpha \)-convertible terms or type expressions are considered identical. Furthermore, we take \( \alpha \) not free in \( \Pi \) to mean that \( \alpha \) is not free in any type expression assigned by \( \Pi \). (The problem is that \( \alpha \) may be free in \( \Pi(\beta) \) in which case rebinding \( \alpha \) might introduce an inconsistency. The same is not true of term variables since \( x \) can not appear in the range of \( \Pi \).)

Here are the typing rules:

\[
\begin{align*}
(\text{ENV}(\cdot)) & \quad \frac{}{w_f(\cdot)} \\
(\text{ENV-term}) & \quad \frac{\Pi \vdash T \in *}{w_f(\Pi[x/T])} \\
(\text{ENV-type}) & \quad \frac{w_f(\Pi)}{w_f(\Pi[\alpha / K])} \quad \text{when } \alpha \text{ is not free in } \Pi \\
(\text{Tvar}) & \quad \frac{w_f(\Pi)}{\Pi \vdash \alpha \in K} \quad \text{when } \Pi(\alpha) = K \\
(\text{WF-\(\Rightarrow\)}) & \quad \frac{\Pi \vdash T \in * \quad \Pi \vdash T' \in *}{\Pi \vdash T \rightarrow T' \in *} \\
(\text{WF-\(\Delta\)}) & \quad \frac{\Pi[\alpha / K] \vdash T \in * \quad \Pi \vdash \Delta \alpha K T \in *}{\Pi[\alpha / K] \vdash T \in K'} \quad \text{when } \alpha \text{ is not free in } \Pi \\
(\text{WF-\(\lambda\)}) & \quad \frac{\Pi[\alpha / K] \vdash T \in K'}{\Pi \vdash \lambda \alpha K T \in K'} \quad \text{when } \alpha \text{ is not free in } \Pi \\
(\text{WF-app}) & \quad \frac{\Pi \vdash T \in K \rightarrow K' \quad \Pi \vdash T' \in K}{\Pi \vdash TT' \in K'} \\
(\text{Var}) & \quad \frac{\Pi \vdash T \in * \quad \Pi \vdash x \in T}{\Pi \vdash x \in T} \quad \text{when } \Pi(\alpha) = T \\
(-I) & \quad \frac{\Pi \vdash T \in * \quad \Pi[\alpha / T] \vdash e \in T'}{\Pi \vdash \lambda x T e \in T \rightarrow T'} \\
(-E) & \quad \frac{\Pi \vdash e \in T \rightarrow T' \quad \Pi \vdash e' \in T}{\Pi \vdash e e' \in T'} \\
(\Delta I) & \quad \frac{\Pi[\alpha / K] \vdash e \in T}{\Pi \vdash e \in \Delta \alpha K T} \quad \text{when } \alpha \text{ is not free in } \Pi \\
(\Delta E) & \quad \frac{\Pi \vdash e \in \Delta \alpha K T' \quad \Pi \vdash e[\alpha / T] \in T'[\alpha / T]}{\Pi \vdash e \in T} \\
(\approx) & \quad \frac{\Pi \vdash e \in T'}{\Pi \vdash e \in T} \quad \Pi \vdash T \in *
\end{align*}
\]

We now show how the above inference rules are applied by checking the type judgment \( t \in \text{Nat} \), where:

\[
t \equiv (\Lambda \Theta : * \rightarrow *, \Lambda \alpha : *. \lambda x : \Theta \alpha . x) [\lambda \sigma : * . \sigma ] [\text{Nat}] 5
\]

For notational convenience we use the following abbreviations:
Chapter 6

The $\omega$-order Polymorphic $\lambda$-Calculus

We are finally ready to close off our hierarchy of languages by defining $F_\omega$, the $\omega$-order polymorphic $\lambda$-calculus, and discussing some of its properties.

6.1 Basic Definitions

The language $F_\omega$ differs from $F_3$, $F_4$, etc. only in that the set of legal kinds is larger:

**Definition 6.1.1:** The syntax of $F_\omega$ is given by the following inductively defined classes:

- (kinds) $K ::= \star \mid K \rightarrow K'$
- (types) $T ::= \alpha \mid T \rightarrow T' \mid \Delta \alpha : K.T \mid \lambda \alpha : K.T \mid TT'$
- (terms) $e ::= x \mid \lambda x : T.e \mid e e' \mid \lambda \alpha : K.e \mid e[T]

where $K$ ranges over kinds, $\alpha$ ranges over type variables, $T$ ranges over types, $e$ ranges over expressions, and $x$ ranges over variables.

Now we can finally list a full set of type and kind inference rules for $F_\omega$, in much the same way as we have in previous chapters. The first three rules deal with the well-formedness of environments, and are named in the same way as before. The next five rules deal with the well-kindness (well-formedness) of types. The base case (Tvar) deals with the kind of a type variable, while the other four, whose names begin with “WF,” deal with the four ways that types may be built up. The next five rules deal with the correct typing of terms. The base case (Var) ensures that the type of a variable is of kind $\star$. The other four deal with cases where $\rightarrow$ or $\Delta$ is introduced or eliminated at the type level, and are named accordingly. Finally, the ($\approx$) rule allows two types that are $\beta\eta$-convertible to be considered equal. There are two new notions in these rules: $T \approx T'$ indicates that the type expressions $T$ and $T'$ are $\beta\eta$-convertible, and $T'[\alpha/T]$ is the result of replacing all occurrences of the
Definability of substitution (and hence $\beta$-reduction). As far as we know, there is no way to define substitution for the higher-order representation of terms. The typed first-order representation requires an equality test for elements of $\text{Var}$, which we do not know how to provide. The first-order representation with explicitly encoded types clearly admits a substitution function.

Inability to represent ill-typed terms. One of the most important properties of both typed representations of terms is that it is impossible to construct a $\text{Term}$ data structure corresponding to a poorly typed term. This eliminates the need to write typechecking functions for represented terms.

Ability to represent the whole metalanguage. Conventional wisdom holds that it is not possible to write a metacircular interpreter in a typed language. For some more exotic type systems—systems with reflexive types [36] or where “Type” is a type [6,35,37]—no complete story has been told.

Uniqueness of representation. Because of the $\text{rep}$ constructor in the higher-order term representation, there are in general many $\text{Terms}$ representing a given term. For the typed first-order representation, it is easy to see that there is an exact correspondence between terms and $\text{Terms}$. For the first-order representation with the encoding of types in the data structure, the situation in general is turned around: when this representation scheme is extended to $\text{F}_\lambda$, there may again be many $\text{Terms}$ corresponding to a given term (since it is possible to represent non-normalized type expressions in the $\text{Term}$ data structure.)
The complete definition of Term would then be:

\[
\text{indtype} \quad \text{Term} : * \rightarrow * \quad \text{with}
\]
\[
\begin{align*}
\text{var:} & \Delta \alpha : *. \ \text{Var} \ \alpha \rightarrow \text{Term} \ \alpha \\
\text{app:} & \Delta \alpha : *. \ \Delta \beta : *. \ \text{Term} (\alpha \rightarrow \beta) \rightarrow \text{Term} \ \alpha \rightarrow \text{Term} \ \beta \\
\text{lam:} & \Delta \alpha : *. \ \Delta \beta : *. \ \text{Var} \ \alpha \rightarrow \text{Term} \ \beta \rightarrow \text{Term} (\alpha \rightarrow \beta)
\end{align*}
\]

This solution corresponds to a "first-order" view of variables: data structures representing variables appear explicitly in the data structure representing a term. For example, the \(F_1\) term

\[
\lambda x : \text{Nat}. \ x
\]

might be represented in this formulation by the Term:

\[
\text{lam } [\text{Nat}][\text{Nat}] (\forall [\text{Nat}] \ 5) (\text{var } [\text{Nat}] (\forall [\text{Nat}] \ 5))
\]

**Exercise 5.5.1:** How would \(\lambda f : \text{Nat} \rightarrow \text{Nat}. \ \lambda a : \text{Nat}. \ f \ a\) be represented in this formulation of Term?

Appendix A carries out the details of this kind of representation of *untyped* \(\lambda\)-calculus terms. Unfortunately, the approach used there apparently cannot be extended to the representation of typed terms. The most important operation on this representation is the substitution of one Term for a variable of the same type in another Term. But in order to decide which variables to replace, it is necessary to decide whether an arbitrary \text{Var} node is equal to the variable being substituted, which requires an equality test on elements of \text{Var}. This test seems not to be implementable in \(F_\omega\). It is easy to implement a test for equality between two elements of most inductive types. But a *polymorphic* equality test, or equivalently a test for "run time" equality between types, cannot be added to \(F_\omega\) without losing the strong normalization property.\(^3\)

In both of these representations of \(F_1\) terms, the \(F_1\) types are subsumed as \(F_3\) types—there is no explicit encoding of types as a data structure in their own right. Another formulation of Term would depend not only on a \text{Var} inductive type, but also on an inductive type called \text{Type}, perhaps with constructors \text{tvar} and \text{arrow}. This idea is pursued further at the end of Appendix A.

There are a number of properties that might be desirable in a representation of terms. The three representations above have different combinations of these properties:

**Definability of eval.** One of the interesting things to do with a data structure representing a term is to evaluate it to produce a value in the metalanguage. The higher-order representation of terms has this property—indeed, so little work is involved in defining \text{eval} that one almost feels a little cheated. We do not know how to define this kind of evaluation function for either of the first-order representations.

\(^3\)We are grateful to Thierry Coquand and Christine Paulin-Mohring for pointing this out. The result is due to Girard [20].
Exercises 5.4.1:
1. Generalize the definitions of [] and eval to correspond to this version of \texttt{Term}.
2. In \texttt{F}_5, functions at the type level may take arguments of any \texttt{F}_4 kind. Write down a \texttt{Term} representation for \texttt{F}_4 in \texttt{F}_4. How many cases are needed for \texttt{typlam} and \texttt{typapp}?

5.5 Alternative Formulations of Term

The higher-order-abstract-syntax-style formulation of \texttt{lam} in Section 5.2 may have seemed somewhat arbitrary and curious, but it has properties that are not shared by the other obvious formulations.

Starting from scratch, our first attempt at a constructor for \texttt{lam} might be:

\texttt{lam: }\Delta\alpha:* \cdot \Delta\beta:* \cdot (\texttt{Term} \alpha \rightarrow \texttt{Term} \beta) \rightarrow \texttt{Term}(\alpha \rightarrow \beta)

But this clearly won't work for us because it cannot be part of an inductive type definition (the variable \texttt{Term} appears negatively). We might then consider two possible patches.

First, we can erase the negative occurrence of \texttt{Term}, leaving

\texttt{lam: }\Delta\alpha:* \cdot \Delta\beta:* \cdot (\alpha \rightarrow \texttt{Term} \beta) \rightarrow \texttt{Term}(\alpha \rightarrow \beta)

and rely on our ability to use \texttt{rep} to convert values of type \alpha to values of type \texttt{Term} \alpha when we're building functions to pass to the constructor. This is the approach we adopted in Section 5.2.

Alternatively, we can erase the parentheses so that \texttt{Term} remains but is no longer in a negative position:

\texttt{lam: }\Delta\alpha:* \cdot \Delta\beta:* \cdot \texttt{Term} \alpha \rightarrow \texttt{Term} \beta \rightarrow \texttt{Term}(\alpha \rightarrow \beta)

This almost corresponds to the syntactic notion that "a \lambda-abstraction of type \alpha\rightarrow\beta is built up by taking a term of type \beta and abstracting it over a variable of type \alpha." Following this intuition, it seems that what we want as the type of the first argument is \texttt{Var} \alpha rather than \texttt{Term} \alpha.\footnote{We could also try to work with this formulation as-is, following the path taken, for example, by LCF \cite{lf}.}

\texttt{lam: }\Delta\alpha:* \cdot \Delta\beta:* \cdot \texttt{Var} \alpha \rightarrow \texttt{Term} \beta \rightarrow \texttt{Term}(\alpha \rightarrow \beta)

Of course, now we need to know what a \texttt{Var} \alpha is. One sensible choice might be that a variable is specified by its type and a numeric index (since we need a countable number of variables of each type):

indtype \texttt{Var}:*\rightarrow* with

\texttt{v: }\Delta\alpha:* \cdot \texttt{Nat} \rightarrow \texttt{Var} \alpha
The extended representation algorithm is:

\[
\begin{align*}
[\text{\[x\]}_\Pi & = \text{rep} \ [\alpha] \ x \quad \text{where } \Pi(x) = \alpha \\
[\lambda \alpha : \cdot e]_\Pi & = \text{lam} \ [\alpha] \ \beta \ \lambda \cdot \cdot \ [\alpha] \ e]_\Pi[\chi/\alpha] \quad \text{where } \Pi[\chi/\alpha](e) = \beta \\
[e \ e']_\Pi & = \text{app} \ [\alpha] \ \beta \ [\alpha] \ [\beta] \ e]_\Pi[\chi/\alpha] \quad \text{where } \Pi(e) = \alpha \rightarrow \beta \text{ and } \Pi(e') = \alpha \\
[\Delta \alpha : * \cdot e]_\Pi & = \text{typlam} \ [\Theta] \ \Delta \alpha : * \cdot e]_\Pi[\alpha/\cdot] \quad \text{where } \Pi(\Delta \alpha : * . e) = \Delta \alpha : * . \Theta \alpha \\
[e[T]]_\Pi & = \text{typapp} \ [\Theta] \ [\alpha]_\Pi \ [\cdot] \quad \text{where } \Pi(e) = \Delta \alpha : * . \Theta \alpha
\end{align*}
\]

The extended eval function is:

\[
\begin{align*}
\text{eval} & \equiv \ \Delta \alpha : * . \ \lambda \cdot : \text{Term } \alpha . \\
& \quad \text{iterTerm} \ [\alpha] \ [\lambda \cdot : * . \sigma] \ t \\
& \quad \quad (\Delta \cdot : * . \lambda u : \cdot : u) \\
& \quad \quad (\Delta \cdot : * . \lambda u : \cdot : u) \\
& \quad \quad (\Delta \cdot : * . \lambda v : \cdot : u) \\
& \quad \quad (\Delta \Theta : * \rightarrow \cdot . \lambda u : \Delta \cdot : * . \Theta \alpha . u) \\
& \quad \quad (\Delta \Theta : * \rightarrow \cdot . \lambda u : \Delta \cdot : * . \Theta \alpha . u[\alpha])
\end{align*}
\]

**Exercise 5.3.1:** Translate the F_2 term

\[
(\Delta \alpha : * . \lambda x : \cdot . x) \ [\text{Nat}] \ 5
\]

into its representation as a Term. Apply eval to the result.

5.4 Representing F_3 in F_4

The only difference between F_3 and F_4 is in the kinds that may appear in terms. Whereas F_3 uses only the kinds * and *\rightarrow*, F_4 also allows the kind *\rightarrow*\rightarrow*, the kind of functions (at the type level) that take type functions as arguments. When we allow type expressions involving this new kind, it becomes possible to represent F_3 programs using an inductive type definition in F_4:

\[
\begin{align*}
\text{indtype} \ \text{Term} : * \rightarrow* \text{ with} \\
\text{rep} : \Delta \alpha : * . \alpha \rightarrow \text{Term } \alpha \\
\text{lam} : \Delta \alpha : * . \Delta \beta : * . \ (\alpha \rightarrow \text{Term } \beta) \rightarrow \text{Term} (\alpha \rightarrow \beta) \\
\text{app} : \Delta \alpha : * . \Delta \beta : * . \ \text{Term} (\alpha \rightarrow \beta) \rightarrow \text{Term } \alpha \rightarrow \text{Term } \beta \\
\text{typlam1} : \Delta \Theta : * \rightarrow \cdot . \\
\quad (\Delta \cdot : * . \text{Term} (\Theta \alpha)) \rightarrow \text{Term} (\Delta \cdot : * . \Theta \alpha) \\
\text{typlam2} : \Delta \Theta : * \rightarrow \cdot \rightarrow \cdot \rightarrow \cdot . \\
\quad (\Delta \cdot : * \rightarrow \cdot . \text{Term} (\Theta \alpha)) \rightarrow \text{Term} (\Delta \cdot : * \rightarrow \cdot . \Theta \alpha) \\
\text{typapp1} : \Delta \Theta : * \rightarrow \cdot . \\
\quad \text{Term} (\Delta \cdot : * . \Theta \alpha) \rightarrow (\Delta \cdot : * . \text{Term} (\Theta \alpha)) \\
\text{typapp2} : \Delta \Theta : * \rightarrow \cdot \rightarrow \cdot . \\
\quad \text{Term} (\Delta \cdot : * \rightarrow \cdot . \Theta \alpha) \rightarrow (\Delta \cdot : * \rightarrow \cdot . \text{Term} (\Theta \alpha))
\end{align*}
\]

Note that since the kind of \Theta appears explicitly in the definition of Term, we need to have two separate cases for typlam and typapp.
Conjecture 5.2.2: There is a bijection between ($\alpha$-equivalence classes of) well-typed terms in $F_1$ and normal-form terms of type $\text{Term}$ in $F_3$.

The importance of this conjecture is that it does not appear to hold for the versions of $\text{Term}$ without $\lambda \text{am}$. (We can make a similar conjecture for the extensions of $\text{Term}$ in Sections 5.3 and 5.4.)

5.3 Representation of $F_2$

Using the technique that we just applied to $\lambda$, it is also possible to represent type abstraction and application, giving a representation of $F_2$ in $F_3$.

\[
\text{indtype} \quad \text{Term} : * \to * \quad \text{with}
\]

\[
\begin{align*}
\text{rep} : & \quad \Delta \alpha : \ast. \quad \alpha \to \text{Term} \alpha \\
\text{lam} : & \quad \Delta \alpha : \ast. \quad \Delta \beta : \ast. \quad (\alpha \to \text{Term} \beta) \to \text{Term}(\alpha \to \beta) \\
\text{app} : & \quad \Delta \alpha : \ast. \quad \Delta \beta : \ast. \quad \text{Term}(\alpha \to \beta) \to \text{Term} \alpha \to \text{Term} \beta \\
\text{typlam} : & \quad \Delta \Theta : * \to * . \quad (\Delta \alpha : \ast . \text{Term}(\Theta \alpha)) \to \text{Term}(\Delta \alpha : \ast . \Theta \alpha) \\
\text{typapp} : & \quad \Delta \Theta : * \to * . \quad \text{Term}(\Delta \alpha : \ast . \Theta \alpha) \to \Delta \alpha : \ast . \text{Term}(\Theta \alpha)
\end{align*}
\]

The intuition behind the constructors $\text{typlam}$ and $\text{typapp}$ is exactly the same as for $\text{lam}$. It is an important characteristic of this representation of $F_2$ terms that, like variables, types are represented implicitly: there is no explicit encoding of $F_2$ types as an $F_3$ data structure. Instead, a $\text{Term}$ representing an $F_2$ term of type $\alpha$ has type $\text{Term} \alpha$, where $\alpha$ is an actual $F_3$ type. Among the benefits of this approach is the fact that it is impossible to construct a poorly typed $\text{Term}$. Only well-typed $F_2$ terms can be represented as instances of $\text{Term}$. (It is this fact that makes the definition of $\text{eval}$ so simple, since it means that there is no need to deal explicitly with types, typechecking, or the possibility of failure at runtime.)

Like $\text{lam}$, the $\text{typlam}$ constructor takes a function as its main argument—this time a function from types to representations of terms rather than from values to representations of terms. The type of the $\text{Term}$ returned by this argument is some function $(\Theta \alpha)$ of the type ($\alpha$) it is passed. The first parameter to $\text{typlam}$ specifies this dependency. Thus, $\Delta$ can be thought of as almost exactly like $\to$, except that $\to$ does not require this extra $\Theta$ to specify how its right hand side depends on its left hand side. Otherwise the correspondence between $\Delta$ and $\to$ would be exact.

Again, $\text{typapp}$ is similar to application, except that it needs an extra parameter $\Theta$ to specify the relation between the argument and result types.
is represented by:

\[
\text{app} \ [\text{Nat} \to \text{Nat}] \ [\text{Nat} \to \text{Nat}]
\]

\[
(\text{lam} \ [\text{Nat} \to \text{Nat}] \ [\text{Nat} \to \text{Nat}] \ (\lambda f: \text{Nat} \to \text{Nat}. \ \text{rep} \ [\text{Nat} \to \text{Nat}] \ f))
\]

\[
(\text{lam} \ [\text{Nat}] \ [\text{Nat}] \ (\lambda x: \text{Nat}. \ \text{rep} \ [\text{Nat}] \ x))
\]

This representation may be a little puzzling at first. Rather than introducing a data type for \( F_1 \) variables and having the \( \text{lam} \) constructor take the representation of a variable (the bound variable) and the representation of a term (the body of the \( \lambda \)-abstraction) as arguments, we represent an \( F_1 \) \( \lambda \)-abstraction by a \textit{higher-order Term}—a function from \( \text{Terms} \) to \( \text{Terms} \). Intuitively, we have incorporated the notion of substitution (or \( \beta \)-reduction) as an integral part of the definition of terms: \( F_1 \) variables have been replaced by \( F_3 \) variables. This method was introduced by Church [10] and appears in the work of Martin-Löf [42], the “Second-Order Patterns” of Huet and Lang [32], the encodings of logics in Harper, Honsell, and Plotkin’s “Logical Framework” [25], and the idea of “Higher-Order Abstract Syntax” of Pfennig and Elliott [48].

The revised canonical representation of \( F_1 \) terms is:

\[
[\alpha]_\Pi = \text{rep} \ [\alpha] \ x \quad \text{where} \ \Pi(x) = \alpha
\]

\[
[\lambda x: \alpha. e]_\Pi = \text{lam} \ [\alpha] \ [\beta] \ \lambda x: \alpha. \ [e]_\Pi[\alpha/x] \quad \text{where} \ \Pi[x/\alpha](e) = \beta
\]

\[
[e \ e']_\Pi = \text{app} \ [\alpha] \ [\beta] \ [e]_\Pi \ [e']_\Pi \quad \text{where} \ \Pi(e) = \alpha \to \beta \text{ and } \Pi(e') = \alpha
\]

The definition of \( \text{eval} \) is extended to:

\[
\text{eval} \equiv \ \Lambda \alpha: \sigma. \ \lambda t: \text{Term} \ \alpha.
\]

\[
\text{iterTerm} \ \alpha [\lambda \sigma: \ast. \sigma] \ t
\]

\[
(\Lambda \sigma: \ast. \ \text{lam} \ \sigma. \ u)
\]

\[
(\Lambda \sigma: \ast. \ \Lambda r: \ast. \ \lambda u: \sigma \to r. \ u)
\]

\[
(\Lambda \sigma: \ast. \ \Lambda r: \ast. \ \lambda u: \sigma \to r. \ \lambda v: \sigma. \ u \ v)
\]

**Exercise 5.2.1:** Translate the \( F_1 \) term

\[(\lambda x: \text{Nat}. \ x) \ 5\]

into its representation as a \( \text{Term} \). Apply \( \text{eval} \) to the result.

Our original definition of \( \text{Term} \) in Section 5.1 allowed certain \( F_1 \) terms—those consisting only of applications and variables—to be represented canonically, in the sense that there was a bijection between \( F_1 \) terms not containing \( \lambda \) and instances of \( \text{Term} \). We would hope that our new definition of \( \text{Term} \) would create a bijection between all the \( F_1 \) terms and those \( F_3 \) terms of type \( \text{Term} \) where \( \text{rep} \) is applied only to variables. This is not quite the case, since two \( F_1 \) terms that differ only in the names of bound variables will have identical canonical representations. But if we consider such pairs of \( F_1 \) terms to be identical, we can make the following conjecture:
\[ \lambda x: \text{Nat.} \ \text{cons} \ [\text{Term(Nat)}] \ [x] \]
\[ \quad (\text{cons} \ [\text{Term(Nat)}] \ [\text{plus} \ x \ x]) \]
\[ \quad (\text{nil} \ [\text{Term(Nat)}]) \]

(letting the typechecker fill in the appropriate environments), instead of:

\[ \lambda x: \text{Nat.} \ \text{cons} \ [\text{Term(Nat)}] \]
\[ \quad (\text{rep} \ [\text{Nat}] \ x) \]
\[ \quad (\text{cons} \ [\text{Term(Nat)}]) \]
\[ \quad (\text{app} \ [\text{Nat}] \ [\text{Nat}] \]
\[ \quad \quad (\text{app} \ [\text{Nat}] \ [\text{Nat} \rightarrow \text{Nat}] \]
\[ \quad \quad \quad (\text{rep} \ [\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}] \ \text{plus}) \]
\[ \quad \quad \quad (\text{rep} \ [\text{Nat}] \ x) \]
\[ \quad \quad \quad (\text{rep} \ [\text{Nat}] \ x)) \]
\[ \quad (\text{nil} \ [\text{Term(Nat)}]) \]

Having defined a representation of $F_1$ terms in $F_3$, it is natural to ask what sorts of manipulations can be performed on this representation. This turns out to be a difficult question, and we defer a complete discussion until we have fully developed the representation itself (see Section 5.5). But it is worth noting here that it is easy to define an evaluation function on this representation:

\[
\text{eval} \in \Delta \alpha: \ast. \ \text{Term} \ \alpha \rightarrow \alpha
\]
\[
\text{eval} \equiv \Delta \alpha: \ast. \ \lambda t: \text{Term} \ \alpha.
\]
\[
\text{iterTerm} \ [\alpha] \ [\lambda \sigma: \ast. \sigma] t
\]
\[
\quad (\Delta \sigma: \ast. \ \lambda u: \sigma. \ u)
\]
\[
\quad (\Delta \sigma: \ast. \ \lambda r: \ast. \ \lambda u: \sigma \rightarrow r. \ \lambda v: \sigma. \ u v)
\]

**Exercises 5.1.2:**

1. Try applying `eval` to the two instances of `Term` given at the beginning of the section.

2. Translate `eval` into pure $F_3$.

**5.2 A Complete Representation of $F_1$ Terms**

Since `app` is used to represent applications and `rep` can be used to represent variables, the only construct left to consider is $\lambda$. There are several possible formulations for the `lam` constructor (see Section 5.5). This one is due to Pfenning and Lee [49]:

indtype `Term:*→*` with
\[
\text{rep:} \ \Delta \alpha: \ast. \ \alpha \rightarrow \text{Term} \ \alpha
\]
\[
\text{lam:} \ \Delta \alpha: \ast. \ \Delta \beta: \ast. \ \lambda (\alpha \rightarrow \text{Term} \ \beta) \rightarrow \text{Term}(\alpha \rightarrow \beta)
\]
\[
\text{app:} \ \Delta \alpha: \ast. \ \Delta \beta: \ast. \ \text{Term}(\alpha \rightarrow \beta) \rightarrow \text{Term} \ \alpha \rightarrow \text{Term} \ \beta
\]

For example, the $F_1$ term
is represented by the $F_2$ data structure

$$\text{app} \ [\text{Nat}] \ [\text{Nat}]
\quad (\text{app} \ [\text{Nat}] \ [\text{Nat} \rightarrow \text{Nat}]
\quad (\text{rep} \ [\text{Nat} \rightarrow \text{Nat}] \ (\lambda x: \text{Nat}. \ \lambda y: \text{Nat}. \ \text{plus} \ x \ y))
\quad (\text{rep} \ [\text{Nat}] \ 2))
\quad (\text{rep} \ [\text{Nat}] \ 3))$$

**Exercise 5.1.1:** Translate the indtype definition of Term into pure $F_3$.

It is already apparent why we seem to need to get out of $F_2$ in order to represent $F_1$ terms. All of the inductively defined types that we can translate into $F_2$ by the techniques described in Chapter 3 share the property of being homogeneous. For example, every sublist of a List Nat is also a List Nat. With terms, this is no longer the case. A $F_1$ term of type $\alpha$ may have subterms of types $\beta \rightarrow \alpha$ and $\beta$, for arbitrary $\beta$. To represent such nonhomogeneous types with inductive definitions, constructors with $\Delta$-types are required.

Another way of looking at this phenomenon is as follows: in the $F_3$ representation of the $F_2$ type constructor Term, the type operator $\Gamma$ is applied to different arguments. In the representation of List, it was possible to define an infinite set of $F_2$ types List Nat, List Bool, etc., each corresponding to an instance of the type constructor List where the variable $\Gamma$ (representing List itself) was applied throughout to exactly one argument $\alpha$, making it possible to replace ($\Gamma \alpha$) everywhere by a single type variable $\gamma$. We cannot think of Term in this way.

There may be many representations of a given $F_1$ term as a Term data structure. For example, the term above is also represented by:

$$\text{rep} \ [\text{Nat}] \ (((\lambda x: \text{Nat}. \ \lambda y: \text{Nat}. \ \text{plus} \ x \ y) \ 2 \ 3))$$

This is a common characteristic of all the term representations discussed in this chapter. However, we can distinguish a “canonical” representation of each $F_1$ term as an instance of Term by stipulating that any application in the original term that is not in the scope of any $\lambda$ must be represented by an app node. (So the first representation above is canonical, and the second is not.)

Formally, we can define the canonical representation $\left[ t \right]_\Pi$ of a term $t$ in an environment $\Pi$ where the types of $t$’s free variables are given by $\Pi$. $\Pi$ is a function taking each free variable of $t$ to an $F_1$ type. We write $\Pi[x/T]$ for the environment $\Pi'$ that agrees with $\Pi$ everywhere except $x$, and that assigns $x$ the type $T$. When $\Pi$ is an environment and $t$ is a well-typed $F_1$ term of type $\alpha$ (assuming its free variables have the types assigned to them by $\Pi$), we write $\Pi(e) = \alpha$.

$$\left[ x \right]_\Pi = \text{rep} \ [\alpha] \ x \quad \text{where} \ \Pi(x) = \alpha$$
$$\left[ \lambda x: \alpha. \ e \right]_\Pi = \text{rep} \ [\alpha \rightarrow \beta] \ (\lambda x: \alpha. \ e) \quad \text{where} \ \Pi(\lambda x: \alpha. \ e) = \alpha \rightarrow \beta$$
$$\left[ e \ e' \right]_\Pi = \text{app} \ [\alpha] \ [\beta] \ [e]_\Pi \ [e']_\Pi \quad \text{where} \ \Pi(e) = \alpha \rightarrow \beta \ \text{and} \ \Pi(e') = \alpha$$

This translation allows us to use the double square brackets as syntactic sugar in $F_3$. We can write $F_3$ programs like
Chapter 5

F₃ as a Metalanguage

Higher-order inductive types appear whenever typed data structures (programs, proofs, etc.) are to be represented and manipulated by other programs. In this chapter we show how F₁ and F₂ terms can be represented as F₂ data structures and discuss how this translation can be generalized to higher orders (representing F₃ in F₂, etc.)

5.1 A Simple Representation of F₁ Terms

To get some feeling for the technique, we begin with a very simple representation of F₁ terms as an inductive type in F₂, which, as usual, can be translated into a closed type in F₃. (We prefer to use inductive types in this section rather than programming directly in F₃ because it makes the definitions easier to read.) Term α is the type of data structures representing F₁ terms of type α. In this representation, only applications appear as explicit constructors in the data structure representing an F₁ term. Variables and λ-abstractions appear only inside of "rep" nodes at the leaves of a tree of applications:

indtype Term:*→* with
  rep: Δα:* . α → Term α
  app: Δα:* . Δβ:* . Term(α→β) → Term α → Term β

For example, the F₁ term

(λx:Nat. λy:Nat. plus x y) 2 3

---

¹Strictly speaking, this is an F₁⁺ term, not a term of pure F₁. For illustrative purposes in the early parts of this chapter, we blur the distinction between the two languages and assume that types like Nat are available in F₁. The inconsistency will disappear when we extend our representation in Section 5.3 to include all of F₂.
\texttt{car} \equiv \Lambda \gamma: *. \lambda l:\text{List} \; \gamma. \lambda d: \gamma.
\begin{align*}
(1 & \ [\lambda \theta: *. \ \theta \to \theta] \\
& \ (\Lambda \alpha: *. \ \lambda d: \alpha. \ d) \\
& \ (\Lambda \alpha: *. \ \lambda x: \alpha. \ \lambda tl': \alpha \to \alpha. \ \lambda d: \alpha. \ x)) \\
& d
\end{align*}

\textbf{Exercises 4.3.2:}

Hand evaluate
\[
\texttt{car} \ [\text{Nat}] \ (\text{nil} \ [\text{Nat}]) \ 0
\]
and
\[
\texttt{car} \ [\text{Nat}] \ (\text{cons} \ [\text{Nat}] \ 3 \ (\text{nil} \ [\text{Nat}]))) \ 0
\]

1. Define the \texttt{cdr} operation on polymorphic lists. (It is a simple generalization of the \texttt{F$_2$} version, since it requires no extra parameters.)
indtype List:*→* with
  nil: Δα:* . List α
  and cons: Δα:* . α → List α → List α

Reasoning as before, we begin its representation as a closed F₅ type with a type abstraction that supplies the type γ of list elements, followed by a higher-order type abstraction on a variable Γ representing List itself, followed by Γγ:

List ≡ λγ:* . ΔΓ:*→*. [?] → Γγ.

Now we fill in the constructors nil and cons, replacing List in their types by Γ:

List ≡ λγ:* . ΔΓ:*→*.
  (Δα:* . Γα)
  → (Δα:* . α → Γα → Γα)
  → Γγ

The constructors nil and cons are given by:

nil ≡ λγ:* . ΔΓ:*→*.
  λn: (Δα:* . Γα).
  λc: (Δα:* . α → Γα → Γα).
  n[γ]
  cons ≡ λγ:* . λx:γ . λl:List γ.
         ΔΓ:*→*.
         λn: (Δα:* . Γα).
         λc: (Δα:* . α → Γα → Γα).
         c[γ] x (l[γ] n c)

By this stage, the reader should be able to check fairly easily that these definitions make sense.

The definition of car in Section 2.3 involved a straightforward use of induction on lists. Unfortunately, the polymorphic case is more difficult. The induction scheme for car is

```
car nil[γ] d       = d
car (cons[γ] x l) d = x
```

where γ is the type of the list elements and d is a default element representing the car of an empty list. This extra parameter is what causes the trouble. By analogy with Section 2.3, we would like to write:

```
car ≡ λγ:* . λl:List γ . λd:γ.
   1 [?]   
  (Δα:* . d)
  (Δα:* . λx:* . λtl':α . x)
```

But there is no type expression to put in place of [?] that makes both arms of the iterator well-typed. Writing λθ:* . γ fails for the cons arm; writing λθ:* . θ fails for the nil arm.

Parameterized induction in F₅ always exhibits this difficulty with the typing of extra parameters. The solution is to reorder the applications so that types are instantiated differently:
pair ≡ Λσ:* . Λτ:* . λx:σ . λy:τ .
ΛΓ:*→*→* .
λp: (Δσ':* . Δτ':*: . σ'→τ'→(Γ σ' τ')). \[ \]

There is only one choice for \[ \] since it must be of type Γστ:

\[
\text{pair} ≡ Λσ:* . Λτ:* . λx:σ . λy:τ .
ΛΓ:*→*→* .
λp: Δσ':* . Δτ':*: . σ'→τ'→(Γ σ' τ') .
\]

\[
p[σ][τ] \times y
\]

It should be starting to become clear that this sort of definition is not as circular as it looks, but to emphasize the point once again let us define the destructor \texttt{fst} without using the \texttt{iterPair} operator.

From its type

\[
\text{fst} : Δα:* . Δβ:* . (\text{Pair } α β) → α
\]

we can see that the outermost part of the definition of \texttt{fst} must be

\[
\text{fst} ≡ Λα:* . Λβ:* . λw:Pair α β . \[ \]
\]

where \[ \] \in α. Clearly, to get something of type α we need to do something with w. Expanding the definition of \texttt{Pair},

\[
\text{fst} ≡ Λα:* . Λβ:* . λw:(ΔΓ:*→*→* . (Δσ:* . Δτ:* . σ→τ→Γστ) → Γαβ) . \[ \]
\]

we see that \[ \] will take the form \( w[F][f] \) for some type constructor F whose kind is \(*→*→*\), and some term \( f \) whose type is \( Δσ:* . Δτ:* . σ→τ→Fστ \). The F and f that will do the job are

\[
F ≡ λφ:* . λψ:* . φ
\]

\[
f ≡ Λφ:* . Λψ:* . λx:φ . λy:ψ . x
\]

so the final definition is:

\[
\text{fst} ≡ Λα:* . Λβ:* .
λw:(ΔΓ:*→*→* . (Δσ:* . Δτ:* . σ→τ→Γστ) → (Γ α β)) .
\[
\]

\[
w [λφ:* . λψ:* . φ] (Λφ:* . Λψ:* . λx:φ . λy:ψ . x)
\]

**Exercises 4.3.1:**

1. Let x:T and y:U. Show that \texttt{fst[T][U]} (\texttt{pair[T][U]} \times y) = x.

2. By analogy with \texttt{fst}, define a destructor \texttt{smd} that returns the second component of a pair.

3. What happens to the first two arguments of \texttt{iterPair} when the \( F_1 \) \texttt{version} of \texttt{fst} is translated into pure \( F_3 \)?

Here is the \texttt{indtype} definition of the type of polymorphic lists:
Whenever we use pair we will need to provide four arguments:

\[
\text{pair } [\text{Nat}] [\text{Bool}] \ 5 \ \text{true}
\]

A simple function manipulating polymorphic pairs is the destructor \(\text{fst} \): 

\[
\text{fst} \equiv \Lambda \alpha ::. \Lambda \beta ::. \lambda w: \text{Pair } \alpha \beta . \ \\
\text{iterPair } [\alpha] \ [\beta] \ [\lambda \phi ::. \lambda \psi ::. \phi] \\
\ w \\
(\Lambda \phi ::. \Lambda \psi ::. \lambda x: \phi . \lambda y: \psi . \ x)
\]

Note that the iterator takes three type parameters instead of just one. The first two specify the type of the argument to the iterator, which is not just Pair, but Pair \(\alpha \beta\) for some particular \(\alpha\) and \(\beta\). Moreover, the third argument is not a type, but a type function with two arguments (i.e., something of the same kind as Pair). It specifies how the type of the iteration's result depends on the kinds of the first and second projections of \(w\).

Let us translate our "higher-order" indtype definition of Pair into a \(F_3\) type definition. The structure of the translation is a bit different this time, but the intuition is the same. Pair itself is a type constructor, not a type; it should take two type parameters and return some type based on them. So our first approximation to the translation is

\[
\text{Pair} \equiv \lambda \alpha ::. \lambda \beta ::. \ [\ ]
\]

where \([\ ] \in \[*\].

Now we can proceed as before. The next argument is a type representing Pair—now of kind \(* \rightarrow* \rightarrow*\) rather than just *, but playing the same role as before:

\[
\text{Pair} \equiv \lambda \alpha ::. \lambda \beta ::. \ \Delta \Gamma ::* \rightarrow* \rightarrow* . \ [
\]

The indtype definition of Pair has one constructor, pair, which we represent here by an argument of the appropriate type (with Pair replaced by the bound variable \(\Gamma\)):

\[
\text{Pair} \equiv \lambda \alpha ::. \lambda \beta ::. \ \Delta \Gamma ::* \rightarrow* \rightarrow* . \ \\
(\Delta \sigma ::. \Delta \tau ::. \ \sigma \rightarrow \tau \rightarrow (\Gamma \ \sigma \ \tau) \\
\rightarrow [\ ]
\]

As usual, we finish by adding "\(\rightarrow [\ ]\)" at the end, where \([\ ]\) represents the type we are defining. In this case, we must apply the bound variable \(\Gamma\) to the arguments \(\alpha\) and \(\beta\) to get something of kind * (just \(\Gamma\) wouldn't make syntactic sense). Our complete definition is:

\[
\text{Pair} \equiv \lambda \alpha ::. \lambda \beta ::. \ \Delta \Gamma ::* \rightarrow* \rightarrow* . \ \\
(\Delta \sigma ::. \Delta \tau ::. \ \sigma \rightarrow \tau \rightarrow (\Gamma \ \sigma \ \tau) \\
\rightarrow (\Gamma \ \alpha \ \beta)
\]

Our next job is to define the constructor pair in \(F_3\). Its arguments are two types, \(\sigma\) and \(\tau\), and two terms of types \(\sigma\) and \(\tau\), respectively.

\[
\text{pair} \equiv \Lambda \sigma ::. \Lambda \tau ::. \lambda x: \sigma . \lambda y: \tau . \ [
\]

Because \([\ ]\) must be something of type Pair \(\sigma \ \tau\), we can proceed by examining the definition of Pair. We see that \([\ ]\) must begin with a type abstraction for the bound variable \(\Gamma\), followed by an abstraction for the bound constructor p:
Another small point that we need to deal with in this definition is that the types given in an \texttt{indtype} definition may now involve \( \lambda \) at the type level, so it only makes sense to talk about the rightmost component of the \textit{normal form} of a type expression.

\textbf{Definition 4.2.2:} The general form of an \texttt{indtype} definition with representation in \( F_3 \) is:

\[
\text{indtype } T:* \rightarrow \ldots \rightarrow * \text{ with } \quad x_1: U_1 \\
\vdots \\
\text{and } x_m: U_m
\]

where each \( U_i \) is an \( F_3 \) type expression, the rightmost component (of the normal form) of which is \( T \) applied to \( n - 1 \) types, \( n \geq 1 \) being the number of occurrences of \( * \) on the first line. The type variable \( T \) may not occur negatively in any of the arguments to the constructors.

Each such definition introduces the following global constants:

1. The type constructor \( T \).
2. Zero or more constructors \( x_i \). Each \( x_i \) takes zero or more type and/or term arguments (as specified by \( U_i \)) and returns a term whose type is \( T \) applied to \( n - 1 \) arguments.
3. An iteration operator \( \text{iter} T \), whose type is

\[
\text{iter} T \in \Delta \alpha_1:* \ldots \Delta \alpha_{n-1}:* \Delta \Gamma:* \rightarrow \ldots \rightarrow *.
\]

\[
T \Rightarrow \alpha_1 \ldots \alpha_n \\
\rightarrow \hat{U}_1 \\
\vdots \\
\rightarrow \hat{U}_m \\
\rightarrow \Gamma \Rightarrow \alpha_1 \ldots \alpha_{n-1}
\]

where \( \hat{U} \) denotes the result of substituting \( \Gamma \) for all occurrences of \( T \) in \( U \).

\textbf{4.3 Programming in \( F_3 \)}

Now we can complete our definition of \texttt{Pair} as an \texttt{indtype}. We know so far that it takes two types as arguments and returns a type, and therefore that its kind is \( * \rightarrow * \rightarrow * \):

\[
\text{indtype } \text{Pair}:* \rightarrow * \rightarrow * \text{ with } [\mathsf{\square}]
\]

Next, we need to define the constructor \texttt{pair}. In general, it takes two terms and returns a pair whose first component has the type of the first term and whose second component has the type of the second term. But we cannot just write

\[
\text{indtype } \text{Pair}:* \rightarrow * \rightarrow * \text{ with } \text{pair}: \alpha \rightarrow \beta \Rightarrow (\text{Pair} \alpha \beta)
\]

because the type variables \( \alpha \) and \( \beta \) are unbound: we do not know in advance what the types of the arguments to \texttt{pair} will be. So we take \( \alpha \) and \( \beta \) as extra parameters:

\[
\text{indtype } \text{Pair}:* \rightarrow * \rightarrow * \text{ with } \text{pair}: \Delta \alpha:* \Delta \beta:* \alpha \rightarrow \beta \Rightarrow (\text{Pair} \alpha \beta)
\]
\[ \Pi \vdash T \in K \]
to mean that in the context \( \Pi \) the type expression \( T \) has kind \( K \). We may abbreviate this as
\[ T \in K \]
when the context \( \Pi \) is clear. By analogy with terms, we also write
\[ \Pi(\alpha) = K \]
or:
\[ \Pi(T) = K \]
and extend \( \Pi \) with the kind of a type variable by writing \( \Pi[\alpha/K] \).

### 4.2 Polymorphic Inductive Datatypes

As before, we think of \textit{indtype} definitions as adding new constants for types, constructors, and iterators to a global environment. But now that we know we will be translating them into simple "macro definitions" of names for closed types and terms in \( F_3 \), we need not bother to be so formal.

One specific point of informality appears when we need to talk about the "result type" of the constructors, which is hard because of an asymmetry in the language. In \( F_1 \) it was easy because the type of any constructor was just a sequence of zero or more arrows, one for each argument, with the argument types on the left of the arrows and the result type on the right of the last arrow. In \( F_2 \), the type of a constructor may be built up from any combination of \( \to \)s and \( \Delta \)s. Intuitively, this presents no problem: we can think of \( \to \) and \( \Delta \) as being the same sort of constructs (both are type operators) and explain their different appearances by observing that \( \Delta \) binds a variable while \( \to \) does not. (In the calculus of constructions [12], for example, this difference disappears and the two operators correspond exactly in form.) So we can still think of an innermost or "rightmost" component of a type. But the notational difficulties involved in giving a completely formal definition of \( F_2 \) are formidable.

We need to know how the notion of positivity extends to types containing \( \Delta \):

**Definition 4.2.1:** The set \( Pos(\mathcal{U}) \) of positively occurring variables in an \( F_2 \) type expression \( \mathcal{U} \) is defined by:

\[
\begin{align*}
Pos(\alpha) & = \{\alpha\} \quad \text{(where } \alpha \text{ is a type variable)} \\
Pos(\alpha \to \beta) & = \text{Neg}(\alpha) \cup \text{Pos}(\beta) \\
Pos(\Delta \alpha:K.\beta) & = \text{Pos}(\beta)
\end{align*}
\]

The set \( Neg(\mathcal{U}) \) of negatively occurring variables in a type expression \( \mathcal{U} \) is defined by:

\[
\begin{align*}
Neg(\alpha) & = \{\} \quad \text{(where } \alpha \text{ is a type variable)} \\
Neg(\alpha \to \beta) & = \text{Pos}(\beta) \cup \text{Neg}(\alpha) \\
Neg(\Delta \alpha:K.\beta) & = \text{Neg}(\beta)
\end{align*}
\]
simpler to begin with \( F_3 \) as a whole and then informally characterize the inductive type declarations that can be represented by closed types in \( F_3 \).)

The main new feature of \( F_3 \) is the ability to express functions from types to types. We will use the same notation as at the term level: the abstraction operator on types is written \( \lambda \), and the juxtaposition of two type expressions denotes the application of the first to the second. (As with ordinary terms, application at the type level is left-associative.)

Finally, just as we needed types to make sure that terms involving abstraction and application were well formed, we now need some notion of the "types" of type expressions to keep them under control. We call the types of types *kinds*. There is a constant kind named \( * \), which is the kind of types of terms; that is, each well-typed term \( e \) has a type \( \alpha \) and the kind of \( \alpha \) is \( * \). (Actually, we introduced \( * \) in \( F_2 \), where it was the only kind and therefore could be treated as pure syntax.) If \( K \) is a kind, then so is \( * \to K \) (the kind of a type function from \( * \) to \( K \)).

**Definition 4.1.1:** The syntax of \( F_3 \) is given by the following inductively defined classes:

\[
K := * | * \to K \\
T := \alpha | T \to T' | \Delta \alpha : K.T | \lambda \alpha : K.T | T T' \\
e := x | \lambda x : T.e | e e' | \Lambda \alpha : K.e | e[T]
\]

where \( K \) ranges over kinds, \( T \) ranges over types, \( \alpha \) ranges over type variables, \( e \) ranges over expressions, and \( x \) ranges over variables.

There can be no confusion between \( \lambda \) at the term level and \( \lambda \) at the type level. The former takes a term argument and returns a term; the latter takes a type argument and returns a type. However, the difference between \( \lambda \) (on types) and \( \Lambda \) can be confusing. Type expressions beginning with \( \Delta \) are the types of terms beginning with \( \Lambda \) (and, of course, of variables of this type)—that is, the type of functions that take a type argument and return a term. Type expressions beginning with \( \lambda \), on the other hand, do not correspond to any terms at all: before they may be the type of a term, they must be applied to enough arguments to produce a type whose outermost operator is either \( \Delta \) or \( \to \), that is, something of kind \( * \). (It does not make sense for \( \lambda \) to be nested within \( \Delta \) or \( \to \) as the latter are of base kind \( * \), that is, the types of terms.)

As an example of \( F_3 \), here is an application of an "even more polymorphic" identity function to some arguments:

\[
(\Lambda \Theta:* \to *. \ \Lambda \sigma:* \ \lambda x : \Theta \sigma . \ x) \\
[\lambda \sigma . \ \text{List} \ \sigma] \\
[\text{Nat}] \\
(\text{cons} \ [\text{Nat}] \ 5 \ (\text{nil} \ [\text{Nat}]))
\]

(More useful examples appear below!)

An important notion we need to introduce here, corresponding to that of type judgment, is the notion of kind judgment. We write
Chapter 4

The Third-order Polymorphic λ-Calculus

We have used the polymorphism of $F_2$ several times now to translate \texttt{indtype} definitions of data types into their representations as closed polymorphic types. But so far, all of the data types we have translated have themselves been monomorphic. In order to go beyond these to "parametric" types like \texttt{List} and \texttt{Pair}, we have to extend our language again, bringing us to $F_3$.

4.1 Definition and Properties of $F_3$

In a sense, lists and cartesian pairs \textit{can} be defined in $F_2$: for each type $\alpha$, there is a type \texttt{List}_$\alpha$. Similarly, for each pair of types $\sigma$ and $\tau$, the type of cartesian pairs of elements of $\sigma$ with elements of $\tau$ can be expressed by the inductive type definition

\begin{quote}
\texttt{indtype Pair}_$\sigma$,$\tau$ with \texttt{pair}_$\sigma$,$\tau$: $\sigma \rightarrow \tau \rightarrow \texttt{Pair}_$\sigma$,$\tau$
\end{quote}

which translates, by the usual method, to the following representation in $F_2$:

\begin{quote}
$\texttt{Pair}_$\sigma$,$\tau$ = $\Delta \gamma:: \ast_\ast: (\sigma \rightarrow \tau \rightarrow \gamma) \rightarrow \gamma$
\end{quote}

But this formulation of pairs is awkward: each $\texttt{Pair}_$\sigma$,$\tau$ instance is different from all others. Whenever we want to use a new kind of pair, we have to write down a new type definition that is structurally identical to the others. We need to abstract away from the types of the elements of the pair and write something like:

\begin{quote}
\texttt{indtype Pair} ...
\end{quote}

Of course, \texttt{Pair} by itself is not a type. (There are no terms whose type is just \texttt{Pair}: each pair of terms has type $\texttt{Pair}_\sigma$, for some particular $\sigma$.) \texttt{Pair} itself is better thought of as a "type constructor," a function from types to types. This sort of \texttt{indtype} definition cannot be expressed in $F^i_2$, nor to our knowledge can it be represented as a closed type in $F_2$. This motivates our next language extension, which takes us to $F_3$. (We will not bother defining a separate language $F^i_2$ this time. It would be very similar to $F_3$, so it is
Similarly:

\[
\text{ack } 1 \ 1 \ \Rightarrow \ (\lambda n:\text{Nat.} \ (\text{succ } n) [\text{Nat}] \ 1 \ \text{succ}) \ 1 \\
\Rightarrow \ 2 \ [\text{Nat}] \ 1 \ \text{succ} \\
\Rightarrow \ 3
\]

The situation gets more complex for \(m, n = 2\):

\[
\text{ack } 2 \ 2 \ \Rightarrow \ (\lambda n:\text{Nat.} \ (\text{succ } n) [\text{Nat}] \ 1 \\
\hspace{1cm} \lambda n':\text{Nat.} \ (\text{succ } n') [\text{Nat}] \ 1 \ \text{succ}) \ 2 \\
\Rightarrow \ 3 \ [\text{Nat}] \ 1 \ (\lambda n':\text{Nat.} \ (\text{succ } n') [\text{Nat}] \ 1 \ \text{succ}) \\
\Rightarrow \ ((\lambda n_3:\text{Nat.} \ (\text{succ } n_3) [\text{Nat}] \ 1 \ \text{succ}) \\
\hspace{1cm} ((\lambda n_2:\text{Nat.} \ (\text{succ } n_2) [\text{Nat}] \ 1 \ \text{succ}) \\
\hspace{2cm} ((\lambda n_1:\text{Nat.} \ (\text{succ } n_1) [\text{Nat}] \ 1 \ \text{succ}) \\
\hspace{3cm} 1))) \\
\Rightarrow \ ((\lambda n_3:\text{Nat.} \ (\text{succ } n_3) [\text{Nat}] \ 1 \ \text{succ}) \\
\hspace{1cm} ((\lambda n_2:\text{Nat.} \ (\text{succ } n_2) [\text{Nat}] \ 1 \ \text{succ}) \\
\hspace{2cm} 3)) \\
\Rightarrow \ ((\lambda n_3:\text{Nat.} \ (\text{succ } n_3) [\text{Nat}] \ 1 \ \text{succ}) \\
\hspace{1cm} 5) \\
\Rightarrow \ 7
\]

Ackermann's function serves to illustrate more of \(F_2\)'s power than our previous examples.
After all these examples, the general form of the translation from \( F_1 \) \text{indtype} definitions to \( F_2 \) types should come as no surprise. The \text{indtype} definition

\[
\text{indtype } T:* \text{ with} \\
c_1:U_{11}\rightarrow U_{12}\rightarrow\cdots\rightarrow U_{1n_1} \rightarrow T \\
\text{and } c_2:U_{21}\rightarrow U_{22}\rightarrow\cdots\rightarrow U_{2n_2} \rightarrow T \\
\vdots \\
\text{and } c_m:U_{m1}\rightarrow U_{m2}\rightarrow\cdots\rightarrow U_{mn_m} \rightarrow T
\]

becomes the \( F_2 \) type

\[
T \equiv \Delta \gamma:*.
\]

\[
(c_1:U_{11}\rightarrow U_{12}\rightarrow\cdots\rightarrow U_{1n_1} \rightarrow \gamma) \\
\vdots \\
\rightarrow (c_m:U_{m1}\rightarrow U_{m2}\rightarrow\cdots\rightarrow U_{mn_m} \rightarrow \gamma) \\
\rightarrow \gamma
\]

where \( \hat{U}_{ij} \) denotes the result of substituting \( \gamma \) for \( T \) in \( U_{ij} \).

### 3.3 The Power of \( F_2 \)

We have seen that \( F_2 \) is sufficient to define the primitive recursive functions, but we have not discussed how powerful the language is. In fact, the class of functions definable in \( F_2 \) is much larger than the primitive recursive functions. In particular, consider Ackermann's function:

\[
\text{ack } 0 \ n \quad = \quad \text{succ } n \\
\text{ack } (\text{succ } m) \ 0 \quad = \quad \text{ack } m \ 1 \\
\text{ack } (\text{succ } m) \ (\text{succ } n) \quad = \quad \text{ack } m \ (\text{ack } (\text{succ } m) \ n)
\]

Ackermann's function exhibits surprisingly explosive growth and is not primitive recursive (A proof is beyond the scope of this work, but may be found in [44].) It is not even immediately obvious that the function is indeed total—that is, that it always terminates. However, we may argue that it is total simply by the fact that it may be encoded within \( F_2 \) [52]:

\[
\text{ack } \equiv \lambda m: \text{Nat. } m \ [\text{Nat}\rightarrow\text{Nat}] \\
\quad \text{succ} \\
\quad \lambda f: \text{Nat}\rightarrow\text{Nat. } \lambda n: \text{Nat.} \\
\quad \ (\text{succ } n) \ [\text{Nat}] \ 1 \ f
\]

(In the last line, the unnecessary parentheses indicate that it is the successor of \( n \) that is iterated.) As this is a complex term, we shall attempt to give the reader some insight into its behavior. For \( m, n = 0 \):

\[
\text{ack } 0 \ 0 \ \rightarrow \ \text{succ } 0 \ \rightarrow \ 1
\]
\[ f \equiv \lambda n : \text{Nat}. \, \text{snd} \cdot \text{Nat} \cdot (n \ [\text{Pair} \cdot \text{Nat} \cdot \alpha] \\
\text{(pair} \cdot \text{Nat} \cdot \alpha \ 0 \ z) \\
\text{(\lambda w : \text{Pair} \cdot \text{Nat} \cdot \alpha, \ \text{pair} \cdot \text{Nat} \cdot \alpha} \\
\text{(succ} \ (\text{fst} \cdot \text{Nat} \cdot \alpha \ w)) \\
\text{(s} \ (\text{fst} \cdot \text{Nat} \cdot \alpha \ w) \ (\text{snd} \cdot \text{Nat} \cdot \alpha \ w))) \]

The function \( f \) iterates over pairs of the form \( \text{pair} \cdot \text{Nat} \cdot \alpha \ n \) (\( f \ n \)):

\[
\begin{align*}
\text{pair} \cdot \text{Nat} \cdot \alpha & \ 0 \ z \\
\text{pair} \cdot \text{Nat} \cdot \alpha & \ 1 \ s \ 0 \ z \\
\text{pair} \cdot \text{Nat} \cdot \alpha & \ 2 \ s \ 1 \ (s \ 0 \ z) \\
\text{pair} \cdot \text{Nat} \cdot \alpha & \ 3 \ s \ 2 \ (s \ 1 \ (s \ 0 \ z)) \\
\end{align*}
\]

Abstracting over \( z \) and \( s \), we arrive at a general formulation of primitive recursion over numbers (Although we would like to abstract over \( \alpha \) as well, this is not possible with the current formulation of pairs.):

\[
\begin{align*}
\text{primrec} \cdot \alpha & \in \alpha \rightarrow (\text{Nat} \rightarrow \alpha \rightarrow \alpha) \rightarrow \text{Nat} \rightarrow \alpha \\
\text{primrec} \cdot \alpha & \equiv \lambda z : \alpha. \ \lambda s : \text{Nat} \rightarrow \alpha \rightarrow \alpha. \\
& \quad \lambda n : \text{Nat}. \ \text{snd} \cdot \text{Nat} \cdot \alpha \\
& \quad \ (n \ [\text{Pair} \cdot \text{Nat} \cdot \alpha] \\
& \quad \text{(pair} \cdot \text{Nat} \cdot \alpha \ 0 \ z) \\
& \quad \text{(\lambda w : \text{Pair} \cdot \text{Nat} \cdot \alpha, \ \text{pair} \cdot \text{Nat} \cdot \alpha} \\
& \quad \text{(succ} \ (\text{fst} \cdot \text{Nat} \cdot \alpha \ w)) \\
& \quad \text{(s} \ (\text{fst} \cdot \text{Nat} \cdot \alpha \ w) \ (\text{snd} \cdot \text{Nat} \cdot \alpha \ w))) \\
\end{align*}
\]

The predecessor function on \( \text{Nat} \) may now be defined as

\[ \text{pred} \equiv \text{primrec} \cdot \text{Nat} \ 0 \ (\lambda m : \text{Nat}. \ \lambda n : \text{Nat}. \ m) \]

The pairs within the resulting iteration look like

\[
\begin{align*}
\text{pair} \cdot \text{Nat} \cdot \text{Nat} & \ 0 \ 0 \\
\text{pair} \cdot \text{Nat} \cdot \text{Nat} & \ 1 \ ((\lambda m : \text{Nat}_. \ \lambda n : \text{Nat}_. \ m) \ 0 \ 0) \\
\text{pair} \cdot \text{Nat} \cdot \text{Nat} & \ 2 \ ((\lambda m : \text{Nat}_. \ \lambda n : \text{Nat}_. \ m) \ 1 \ 0) \\
\text{pair} \cdot \text{Nat} \cdot \text{Nat} & \ 3 \ ((\lambda m : \text{Nat}_. \ \lambda n : \text{Nat}_. \ m) \ 2 \ 1) \\
\end{align*}
\]

Notice that our predecessor function is surprisingly inefficient, requiring order \( n \) time to compute \( \text{pred} \ n \). Unfortunately, there is some theoretical evidence that this inefficiency is inherent.

**Exercises 3.2.7:**

1. Define the factorial function using \( \text{primrec} \).
2. Define a function in \( F_2 \) that returns \text{true} if its two numeric arguments are equal and \text{false} if they are unequal.
Like iteration, primitive recursion performs structural induction—that is, computation is guided by the traversal of existing data structure. As these structures are finite, the computation must terminate. We only consider primitive recursion over numbers, but other domains may be formulated analogously.

The primitive recursion scheme on Nats is

\[
\begin{align*}
    f \, 0 &= z \\
    f \, (\text{succ} \, n) &= s \, n \, (f \, n)
\end{align*}
\]

for \( f \in \text{Nat} \rightarrow \alpha \), \( z \in \alpha \), \( s \in \text{Nat} \rightarrow \alpha \rightarrow \alpha \). Primitive recursion is somewhat more complex than iteration in that successive values may depend on the value of the predecessor \( n \) as well as the function result \( (f \, n) \) computed from \( n \). It should be obvious that iteration is a special case of primitive recursion. However, it turns out that primitive recursion may similarly be derived from iteration.

To implement the primitive recursive functions within \( F_2 \), we require a representation of pairs. This comes from the need to iterate over \( s \)—a function of \( n \) and the value of \( s \) for \( n \). The pair data structure makes both these values available to the incremental computation at each step of the iteration (as in the \( \text{cdr} \) of Section 2.3).

As with iterators and lists, we may form the family of cartesian pairs. (As is the case with \( \text{iter} \), the need for a type family may be replaced with an abstraction, but the reader must wait for \( F_3 \).)

\[
\begin{align*}
\text{indtype} \ & \ \text{Pair}_{\sigma, \tau} \ 
\text{with} \\
\ 
\text{pair}_{\sigma, \tau} : \sigma \rightarrow \tau \rightarrow \text{Pair}_{\sigma, \tau}
\end{align*}
\]

This may be translated into \( F_2 \):

\[
\begin{align*}
\text{Pair}_{\sigma, \tau} & \equiv \Delta \gamma : \ast. \, (\sigma \rightarrow \tau \rightarrow \gamma) \\
\text{pair}_{\sigma, \tau} & \equiv \lambda x : \sigma. \ \lambda y : \tau. \ \Delta \gamma : \ast. \ \lambda p : \sigma \rightarrow \tau \rightarrow \gamma. \ p \ x \ y
\end{align*}
\]

So, for example,

\[
\text{pair}_{\text{Bool}, \text{Nat}} \, \text{true} \, 1 = \Delta \gamma : \ast. \ \lambda p : \text{Bool} \rightarrow \text{Nat} \rightarrow \gamma. \ p \ \text{true} \ 1
\]

In order for our pairs to be of any use, we also need families of destructuring operators:

\[
\begin{align*}
\text{fst}_{\sigma, \tau} & \in \text{Pair}_{\sigma, \tau} \rightarrow \sigma \\
\text{fst}_{\sigma, \tau} & \equiv \lambda w : \text{Pair}_{\sigma, \tau}. \ w \ [\sigma] \ (\lambda x : \sigma. \ \lambda y : \tau. \ x) \\
\text{smd}_{\sigma, \tau} & \in \text{Pair}_{\sigma, \tau} \rightarrow \tau \\
\text{smd}_{\sigma, \tau} & \equiv \lambda w : \text{Pair}_{\sigma, \tau}. \ w \ [\tau] \ (\lambda x : \sigma. \ \lambda y : \tau. \ y)
\end{align*}
\]

For example:

\[
\begin{align*}
\text{fst}_{\text{Bool}, \text{Nat}} \ (\text{pair}_{\text{Bool}, \text{Nat}} \, \text{true} \, 1) \\
&= (\Delta \gamma : \ast. \ \lambda p : \text{Bool} \rightarrow \text{Nat} \rightarrow \gamma. \ p \ \text{true} \ 1) \ [\text{Bool}] \ (\lambda x : \text{Bool}. \ \lambda y : \text{Nat}. \ x) \\
&= \text{true}
\end{align*}
\]

We may now implement the primitive recursive function \( f \) from above as:
\[\text{iterNat} \in \Delta \alpha : *. \text{Nat} \to \alpha \to (\alpha \to \alpha) \to \alpha \]
\[\text{iterNat} \equiv \lambda \alpha : *. \lambda n : \text{Nat}. \lambda z : \alpha. \lambda s : \alpha \to \alpha. n [\alpha] z s\]

As is the case with Bools, we have
\[\text{iterNat} [\alpha] n = n [\alpha]\]

For the remainder of this text, we will generally avoid explicit iterators since the self-iterating approach is both simpler and more elegant.

We previously defined the addition of \( m \) and \( n \) by taking the \( m \)th successor of \( n \). Using the iteration implicit within our representation of \( \text{Nat} \), we have
\[\text{plus} \in \text{Nat} \to \text{Nat} \to \text{Nat}\]
which may be defined as
\[\text{plus} \equiv \lambda m : \text{Nat}. \lambda n : \text{Nat}. m [\text{Nat}] n \\text{succ}\]
(The final arguments \( n \) and \( \text{succ} \) are substituted for the \( z \) and \( s \) arguments of \( m \), respectively.)

**Exercise 3.2.5:** Try adding two small numbers.

Similarly, multiplication may be defined by iterating \( \text{plus} \):
\[\text{mult} \equiv \lambda m : \text{Nat}. \lambda n : \text{Nat}. n [\text{Nat}] 0 (\text{plus} m)\]
It might help to think of \((\text{plus} m)\) as \((\lambda n : \text{Nat}. \text{plus} m n)\).

The following predicate tests for zero:
\[\text{zero?} \equiv \lambda m : \text{Nat}. m [\text{Bool}] \text{true} (\lambda b : \text{Bool}. \text{false})\]

Now let us return to the formulation of lists we gave in \( F_1^i \):
\[\text{indtype List}_\text{Nat} : * \text{ with }\]
\[
\begin{align*}
\text{nil}_\text{Nat} & : \text{List}_\text{Nat} \\
\text{cons}_\text{Nat} & : \text{Nat} \to \text{List}_\text{Nat} \to \text{List}_\text{Nat}
\end{align*}
\]

The above is rendered in \( F_2 \) as:
\[\begin{align*}
\text{List}_\text{Nat} & \equiv \Delta \gamma : *. \gamma \to (\text{Nat} \to \gamma \to \gamma) \to \gamma \\
\text{nil}_\text{Nat} & \equiv \Lambda \gamma : *. \Lambda n : \gamma. \lambda c : \text{Nat} \to \gamma \to \gamma. n \\
\text{cons}_\text{Nat} & \equiv \lambda x : \text{Nat}. \lambda l : \text{List}_\text{Nat}. \Lambda \gamma : *. \lambda n : \gamma. \lambda c : \text{Nat} \to \gamma \to \gamma. c \times (1 [\gamma] n \ c)
\end{align*}\]

We may use the representation to iterate over a \( \text{List}_\text{Nat} \) producing its sum:
\[\begin{align*}
\text{sumlist} & \in \text{List}_\text{Nat} \to \text{Nat} \\
\text{sumlist} & \equiv \lambda l : \text{List}_\text{Nat}. l [\text{Nat}] 0 \ \text{plus}
\end{align*}\]

**Exercise 3.2.6:** Render the Tree \( \text{indtype} \) of Section 3 in \( F_2 \).

General or unrestricted recursion is not available in \( F_2 \), since recursion can be used to express nonterminating computations and we know that \( F_2 \) is strongly normalizing. However, the more restricted mechanism of primitive recursion may be formulated in our calculi.
For self-iteration, we must be able to specify a type for the result. Hence instances of type \texttt{Nat} begin with a type abstraction so the terms may be specialized to produce the result type of the iteration. The first \( \gamma \) argument corresponds to the \texttt{zero} constructor (of type \texttt{Nat}), while the \( \gamma \rightarrow \gamma \) parameter corresponds to \( \texttt{succ} \in \texttt{Nat} \rightarrow \texttt{Nat} \).

Instances of \texttt{Nat} take the following form:

\[
\begin{align*}
0 & \equiv \Lambda \gamma :: \texttt{. } \lambda z :: \gamma :: \lambda s :: \gamma \rightarrow \gamma :: z \\
1 & \equiv \Lambda \gamma :: \texttt{. } \lambda z :: \gamma :: \lambda s :: \gamma \rightarrow \gamma :: s \ z \\
2 & \equiv \Lambda \gamma :: \texttt{. } \lambda z :: \gamma :: \lambda s :: \gamma \rightarrow \gamma :: (s \ z) \\
3 & \equiv \Lambda \gamma :: \texttt{. } \lambda z :: \gamma :: \lambda s :: \gamma \rightarrow \gamma :: (s \ (s \ z)) \\
\vdots
\end{align*}
\]

\textbf{Exercise 3.2.3:} Verify that all closed, normal-form terms of type \texttt{Nat} have this shape.

The zero function, \( \texttt{zero} \in \texttt{Nat} \), should just produce \( 0 \):

\[
\texttt{zero} \equiv \Lambda \gamma :: \texttt{. } \lambda z :: \gamma :: \lambda s :: \gamma \rightarrow \gamma :: z
\]

The successor function, \( \texttt{succ} \in \texttt{Nat} \rightarrow \texttt{Nat} \), may then be represented by

\[
\texttt{succ} \equiv \lambda n :: \texttt{Nat} :: \Lambda \gamma :: \texttt{. } \lambda z :: \gamma :: \lambda s :: \gamma \rightarrow \gamma :: s(n[\gamma] z s)
\]

or alternatively as

\[
\texttt{succ}' \equiv \lambda n :: \texttt{Nat} :: \Lambda \gamma :: \texttt{. } \lambda z :: \gamma :: \lambda s :: \gamma \rightarrow \gamma :: n[\gamma](s z)s
\]

Let us make sure again that these definitions are well typed.

\[
\begin{align*}
\texttt{succ}, \texttt{succ}' & \in \texttt{Nat} \rightarrow \Delta \gamma :: \texttt{. } \gamma \rightarrow (\gamma \rightarrow \gamma) \rightarrow \gamma \\
& \in \texttt{Nat} \rightarrow \Delta \gamma :: \texttt{. } \gamma \rightarrow (\gamma \rightarrow \gamma) \rightarrow \gamma \\
& \text{since } n[\gamma](s z)s, \ s(n[\gamma]zs) \in \gamma \\
& \in \texttt{Nat} \rightarrow \texttt{Nat}
\end{align*}
\]

As with the above definition of \( \texttt{not} \), the arguments \( z \) and \( s \) are not actually instantiated when \( \texttt{succ} \) is applied to a number.

\textbf{Exercise 3.2.4:} Check that \( (\texttt{succ} \ 0) = 1 \).

The above definitions of \( \texttt{succ} \) and \( \texttt{succ}' \) demonstrate how an operation may have two definitions that denote the same function but are not \( \beta \)-equivalent. However, in this case we should not be surprised that the two definitions are not convertible: they are clearly achieving the same result in two different ways. In the first definition, we replace the \( z \) part of a number by \( s(z) \). Since the arguments to a number are its "zero" and "successor" elements respectively, we have merely replaced the "zero" element with what is really "one." In the second definition, we apply an extra \( s \) to the "outside" of the number.

We have defined our representation so that the resulting terms serve as their own iterators, but let us define the explicit iteration schemes for \texttt{Nat} to reinforce this idea. The iterator over natural numbers is defined polymorphically:
terms directly as iterators by taking advantage of this similar structure; rather than explicitly abstracting \( t' \) and \( f' \) as above, we make use of the corresponding \( t \) and \( f \) parameters of \( b \). So for \( b \in \text{Bool} \), we have the following equivalence

\[
\text{iterBool} [\gamma] \ b \ = \ b \ [\gamma]
\]
because for \( t', f' \in \gamma \):

\[
\text{iterBool} [\gamma] \ b \ t' \ f' \ = \ b \ [\gamma] \ t' \ f'
\]

In fact, we shall see that this “self-iterating” property of \text{Bools} holds in general for the representation of all inductively defined types. Explicit iterators are, then, no longer necessary!

As before, we can use iteration to define various functions on booleans. For example, we may alternatively define \( \text{not} \) as

\[
\text{not} \ \equiv \ \lambda b: \text{Bool}. \ \text{iterBool} [\text{Bool}] \ b \ \text{false} \ \text{true}
\]
or now directly as

\[
\text{not} \ \equiv \ \lambda b: \text{Bool}. \ b \ [\text{Bool}] \ \text{false} \ \text{true}
\]

Although the above is similar to the original definition of \( \text{not} \), the two are not \( \beta\eta \)-equivalent, as the latter does not explicitly set up the \( \gamma \), \( t \), and \( f \) arguments. Rather the booleans \( \text{true} \) and \( \text{false} \) are returned directly.

We may also define binary boolean functions:

\[
\begin{align*}
\text{or} & \in \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\
\text{or} & \equiv \lambda b_1: \text{Bool}. \ \lambda b_2: \text{Bool}. \ b_1 \ [\text{Bool}] \ \text{true} \ b_2 \\
\text{and} & \in \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\
\text{and} & \equiv \lambda b_1: \text{Bool}. \ \lambda b_2: \text{Bool}. \ b_1 \ [\text{Bool}] \ b_2 \ \text{false}
\end{align*}
\]

The operation \( \text{or} \) works by returning \( \text{true} \) if \( b_1 \) is \( \text{true} \) and \( b_2 \) otherwise, while \( \text{and} \) yields \( b_2 \) if \( b_1 \) is \( \text{true} \) and \( \text{false} \) otherwise.

**Exercise 3.2.2:** Apply \( \text{and} \) or \( \text{or} \) to combinations of \( \text{true} \) and \( \text{false} \) to convince yourself of their validity. Consider the definition of terms performing other boolean operations (e.g., \( \text{xor} \), \( \text{implies} \)).

We have asked the reader to accept the translation of inductive definitions on faith, but now consider it more carefully. In rendering \( F_1 \) types into \( F_2 \), the primary goal is to achieve the self-iteration property we have observed in booleans. To accomplish this, **term instances of the defined type must capture the structure of the induction that defined them**. Returning to the natural numbers, we have

\[
\text{indtype} \ \text{Nat}: * \ \text{with} \ \text{zero}: \text{Nat} \ \text{and} \ \text{succ}: \text{Nat} \rightarrow \text{Nat}
\]
ranged as

\[
\text{Nat} \ \equiv \ \Delta \gamma:* \ . \ \gamma \rightarrow (\gamma \rightarrow \gamma) \rightarrow \gamma
\]
\[ \text{not} \in \text{Bool} \to \text{Bool} \]
\[ \text{not} \equiv \lambda b:\text{Bool}. \, \lambda \gamma'. \, \lambda t':\gamma'. \, \lambda f':\gamma'. \, b[\gamma'] t' f' \]

The reader may feel somewhat overwhelmed by the syntax of the above expression, so let us attempt to piece together an understanding in stages. The function \text{not} takes a boolean argument and produces a boolean result, so it must clearly have the form
\[ \lambda b:\text{Bool}. \, (\lambda t':\gamma'. \, \lambda f':\gamma'. \, \gamma') \]
where \([\gamma']\) has type \(\gamma '\). The bound variables \(t'\) and \(f'\) are both of type \(\gamma '\), but we need to select between them on the basis of \(b\)'s truth or falsity. The boolean \(b\) can be used (as an iterator) to do precisely this; that is, the boolean term \(b\) is represented by a function, and can therefore be applied to other terms. Since \(b\) is either the function \text{true} or the function \text{false}, it will select either its first or second argument, respectively. The above definition swaps the arguments to \(b\), producing the negation of \(b\).

It is often less than obvious that such a function behaves as expected. For example, \text{not} has a large number of arguments and seems to return an expression of type \(\gamma '\), rather than type \text{Bool}. The key is to remember that all functions are curried, and that when \text{not} is applied to an argument, only the first \(\lambda\) will be \(\beta\)-reduced. The arguments \(\gamma '\), \(t'\) and \(f'\) will not be instantiated. Rather, it is the \(t\) and \(f\) arguments of \(b\) that will be replaced by the outer \(t'\) and \(f'\). Let us go through and check that \text{not} is in fact correctly typed:
\[
\begin{align*}
\text{not} & \in (\Delta \gamma'. \, \gamma \to \gamma \to \gamma) \to \Delta \gamma'. \, \gamma' \to \gamma' \to [\gamma'] \\
& \in (\Delta \gamma. \, \gamma \to \gamma \to \gamma) \to \Delta \gamma'. \, \gamma' \to \gamma' \to \gamma' \\
& \quad \text{since } b[\gamma'] t f \in [\gamma'] \\
& \in (\Delta \gamma. \, \gamma \to \gamma \to \gamma) \to (\Delta \gamma'. \, \gamma' \to \gamma' \to \gamma') \\
& \in \text{Bool} \to \text{Bool}
\end{align*}
\]

**Exercise 3.2.1:** Check that the above is valid by performing the \(\beta\)-reductions in \(\text{not} \text{true}\).

We have shown the representation of some \(F^i_1\) inductive types as \(F_2\) types and terms, but we have not said anything about the associated \text{iter} operators. In fact, the iterators for \(F^i_1\) \text{ind} types may also be specified within \(F_2\). In the case of booleans, \text{iterBool} takes a type \(\gamma\) (specifying the result type of the iteration), and a \text{Bool}, followed by one argument (of type \(\gamma\)) for each of the constructors \text{true} and \text{false}:
\[ \text{iterBool} \in \Delta \gamma. \, \text{Bool} \to \gamma \to \gamma \to \gamma \]

Depending on which constructor was used to produce the \text{Bool}, \text{iterBool} returns either the first argument (when \(b = \text{true}\)) or the second (when \(b = \text{false}\)):
\[ \text{iterBool} \equiv \lambda \gamma:*. \, \lambda b:\text{Bool}. \, \lambda t':\gamma. \, \lambda f':\gamma. \, b[\gamma] t' f' \]

(Again you might think of \text{iterBool} as an "if...then...else" expression: if \(b\) then \(t'\) else \(f'\).)

Consider that \text{true} and \text{false} are each type-parameterized, two-argument functions, similar to \text{iterBool} (except that the latter has the boolean argument \(b\)). We may use boolean
where the omitted subterm $\overline{A}$ has type $\gamma$. Again, the normal-form theorem tells us that if there is any term that can take the place of $\overline{A}$, then there is one in normal-form. Hence, there are three possibilities to consider:

1. $\overline{A}$ is a $\lambda$-abstraction. But then the type of $\overline{A}$ would be an arrow type $(\alpha \to \beta$, for some $\alpha$ and $\beta$), whereas $\gamma$ is a type variable.

2. $\overline{A}$ is a $\Lambda$-abstraction. But the type of $\overline{A}$ would then begin with a $\Delta$, while $\gamma$ is a type variable.

3. $\overline{A}$ is a variable. This variable would clearly have to be of type $\gamma$. But the whole term must be closed, so we cannot use a free variable of type $\gamma$, nor does the $\overline{A}$ appear in the scope of any bound variable of type $\gamma$.

Thus there are no normal-form terms—and hence no terms at all—of type $\text{Void}$.

Again on faith, we render the $\text{F}_1^\prime$ definition

\textit{indtype \text{Unit}::* with \text{unit:Unit}}

as the $\text{F}_2$ type

$$\text{Unit} \equiv \Delta \gamma:* \cdot \gamma \to \gamma$$

which has exactly one normal-form instance (corresponding to the single constant constructor $\text{unit} \in \text{Unit}$). As before, any closed normal-form term of type $\text{Unit}$ must begin with a $\Lambda$-abstraction. Then, since the $\Delta$ in the type is followed by an $\to$ (and there are no variables with arrow type available), the $\Lambda$-abstraction in the term must be followed by a $\lambda$-abstraction:

$$\Lambda \gamma:* \cdot \lambda u: \gamma. \overline{A}$$

Again, $\overline{A}$ must be of type $\gamma$. Reasoning as before, we find that this time there is a normal-form term of type $\gamma$ available to fill the place of $\overline{A}$:

$$\text{unit} \equiv \Lambda \gamma:* \cdot \lambda u: \gamma. \ u$$

Turning our attention to booleans, we render

\textit{indtype \text{Bool}::* with \text{true:Bool} and \text{false:Bool}}

as

$$\text{Bool} \equiv \Delta \gamma:* \cdot \gamma \to \gamma \to \gamma$$

with the two possible normal-form terms of this type being these:

$$\text{true} \equiv \Lambda \gamma:* \cdot \lambda t: \gamma. \lambda f: \gamma. \ t$$

$$\text{false} \equiv \Lambda \gamma:* \cdot \lambda t: \gamma. \lambda f: \gamma. \ f$$

Let us consider the definition of the boolean negation function directly in $\text{F}_2$:
Exercise 3.1.2: Use the type deduction rules for F₂ to prove that:
\[
\text{double \space [Bool] \not \space true} \in \text{Bool}
\]

3.2 Representing \( F^i_1 \) Inductive Type Definitions in \( F_2 \)

At this point, the reader may be expecting us to reintroduce the indtype mechanism for defining primitive types and constructors, producing \( F^i_2 \). But we need not take this step yet. It is a surprising fact that we can translate any inductive type definition in \( F^i_1 \) into a representation in \( F_2 \). More specifically, each type \( T \) introduced by an indtype definition in \( F^i_1 \) can be expressed as a closed type expression of \( F_2 \), and each of the constructors of \( T \) can be expressed as a closed term in \( F_2 \).

The formal details of the translation are quite technical (see \([3,45,49]\)), but it is relatively easy to understand operationally.

Let us begin with the simplest example from \( F^i_1 \). The type Void, with no constructors, was defined by:

\[
\text{indtype \space Void:*}
\]

Without explaining, for the moment, how we arrive at it, let the \( F_2 \) type corresponding to this definition be:

\[
\text{Void} \equiv \Delta \gamma : * . \gamma
\]

To justify the assertion that this type "represents" the indtype Void, we show that there are no closed \( F_2 \) terms of type Void. If there is any term of type Void, then there is one in normal-form. Since Void begins with a \( \Delta \), any closed normal-form term of this type must be a \( \lambda \)-abstraction (this follows from the typing rules):

\[
\lambda \gamma : * . \gamma
\]
The remaining four rules (¬I), (¬E), (ΔI) and (ΔE) also deal with the correct typing of terms, and are named according to whether the symbols → and Δ are introduced or eliminated at the type level by that rule. The notation \( T'[\alpha/T] \) is the result of replacing all occurrences of the type variable \( \alpha \) in \( T' \) with \( T \), while renaming bound variables in \( T' \) to avoid capture. We take "\( \alpha \) is not free in \( \Pi \)" to mean that \( \alpha \) is not free in any type expression assigned by \( \Pi \).

\[
\begin{align*}
\text{(ENV-\{\})} & \quad \overline{\text{wf}(\{\})} \\
\text{(ENV-term)} & \quad \Pi \vdash T \in * \quad \text{wf}(\Pi[\alpha/T]) \\
\text{(ENV-type)} & \quad \text{wf}(\Pi) \quad \Pi \vdash \alpha \in \pi \quad \text{when } \alpha \text{ is not free in } \Pi \\
\text{(Tvar)} & \quad \Pi \vdash T \in * \quad \Pi \vdash \alpha \in \pi \\
\text{(WF→→)} & \quad \Pi \vdash T \in * \quad \Pi \vdash T' \in * \\
\text{(WF-Δ)} & \quad \Pi \vdash \Delta T \in * \\
\text{(Var)} & \quad \Pi \vdash x \in T \\
\text{(¬I)} & \quad \Pi \vdash \alpha \in * \\
\text{(¬E)} & \quad \Pi \vdash e \in T \quad \Pi \vdash e' \in T' \\
\text{(ΔI)} & \quad \Pi \vdash \alpha \in T \\
\text{(ΔE)} & \quad \Pi \vdash e \in \Delta T \\
\end{align*}
\]

We give an example of how to use these rules by proving that

\[
\text{double \ [Nat] succ zero } \in \text{ Nat}
\]

where

\[
\text{double } \equiv \Delta \alpha:*. \lambda f: \alpha \rightarrow \alpha. \lambda x: \alpha. f(f\ x)
\]

Note that in this example and the following exercise, we need to make some assumptions about the types Nat and Bool, since these are not included in \( F_2 \). In particular, we assume that in all environments we have \( \text{Nat} \in *, \text{zero } \in \text{Nat}, \text{succ } \in \text{Nat} \rightarrow \text{Nat}, \text{Bool} \in *, \text{true } \in \text{Bool} \) and \( \text{not } \in \text{Bool} \rightarrow \text{Bool} \). Later we will see how they can be defined, but for now we will treat them as if they were built in.

1. \( \text{wf}(\{\}) \) by (ENV-\{\})
2. \( \text{wf}(\langle(\alpha,*)\rangle) \) by (ENV-type) from 1
3. \( \langle(\alpha,*)\rangle \vdash \alpha \in * \) by (Tvar) from 2
4. \( \langle(\alpha,*)\rangle \vdash \alpha \rightarrow \alpha \in * \) by (WF→→) from 3, 3
5. \( \text{wf}(\langle(\alpha,*)\rangle(,f,\alpha \rightarrow \alpha)) \) by (ENV-term) from 4
6. \( \langle(\alpha,*)\rangle(,f,\alpha \rightarrow \alpha) \vdash \alpha \in * \) by (Tvar) from 5
7. \( \text{wf}(\langle(\alpha,*)\rangle(,f,\alpha \rightarrow \alpha)(,x,\alpha)) \) by (ENV-term) from 6
8. \( \langle(\alpha,*)\rangle(,f,\alpha \rightarrow \alpha)(,x,\alpha) \vdash \alpha \in * \) by (Tvar) from 7
3.1 Definitions

We now proceed to a formal definition of $F_2$.

**Definition 3.1.1:** The syntax of $F_2$ (the second-order polymorphic $\lambda$-calculus) is given by the following inductively defined classes:

$$
T ::= \alpha \mid T \to T' \mid \Delta \alpha : \tau\ .\ T \\
\text{e} ::= x \mid \lambda x : T . \text{e} \mid \text{e} e' \mid \Delta \alpha : \tau \ .\ \text{e} \mid \text{e}[T]
$$

where $T$ ranges over types, $\alpha$ ranges over type variables, $\text{e}$ ranges over expressions, and $x$ ranges over variables.

As before, $\lambda$ is used to construct functions that can be applied to a term, yielding a term—i.e., *term abstractions*—whereas $\Delta$ constructs functions that can be applied to a type, yielding a term—i.e., *type abstractions*. The $\to$ is used to represent the type of a term abstraction ($\lambda$), while $\Delta$ forms the type of a type abstraction ($\Delta$). In addition to applying terms to other terms (*term application*), $F_2$ allows the application of terms to types with the syntax $\text{e}[\alpha]$ (*type application*). As before $\gamma:*$ simply indicates that $\gamma$ is a type.

The concrete syntax of $F_2$ follows the same conventions as $F_1$. The $\to$ symbol associates to the right, and application (of both terms and types) associates to the left. The $\Delta$ and $\Lambda$ operators behave like $\lambda$ in that their bodies extend as far as possible to the right—to the end of the whole expression, or up to an unmatched right parenthesis.

Type variables are *free* or *bound* in the same sense as the term variables of $F_1$. A *closed* $F_2$ term contains no free term- or type variables. A closed type expression contains no free type variables.

As in the original definition of $F_1$, there are no constant types or terms in $F_2$. However, unlike pure $F_1$, the sets of types and terms of pure $F_2$ are inhabited (i.e., non-empty).

The $F_1$ notions of $\alpha$-conversion (renaming of bound variables) and $\beta\eta$-reduction and conversion are extended to include type abstraction over terms and term application:

$$(\Delta \alpha : \tau .\ \text{e})[T] \Rightarrow_\beta \text{e}[\alpha/T]$$

$$(\Delta \alpha : \tau .\ \text{e}[\alpha]) \Rightarrow_\eta \text{e} \quad \text{(provided $\alpha$ is not free in $\text{e}$)}$$

The appropriate analogues of Theorems 2.1.5 to 2.1.6 also hold for $F_2$ (though some of the proofs are significantly more difficult). In particular, every $F_2$ term is strongly normalizing.

We may define the typing of $F_2$ formally by extending the type inference rules of $F_1$. The first three rules ($\text{ENV}(-\cdot)$, (ENV-term) and (ENV-type) deal with the well-formedness of environments, and $\text{wf}(\Pi)$ is used to say that the environment $\Pi$ is well-formed. The base case (Tvar) deals with type variables. The two rules (WF-→) and (WF-Δ) also deal with the well-formedness of types, and this is how they get their names. The rule (Var) looks up the type of a variable in the current environment and checks that it is well formed.
double [Bool] not true ⇒ true

We say that functions taking type arguments in this manner are *polymorphic* in that they may be applied to terms of differing type.

We have extended the language of terms to include polymorphic functions, but have not considered the corresponding extension of the type language so that such terms may be given types. What, then, is the *type* of a polymorphic function? It is something like an → type, since it is a kind of function. But again, since it takes a type and returns a term, we want a different notation from →. Also, the type of the result returned by such a function can vary based upon the argument given to it; thus we need an explicit way of indicating this dependence. We introduce a new symbol ∆ that, like λ and Λ, binds a variable in the scope of another expression (but is used for describing types instead of terms). Now id, which takes a type α and returns a function from α to α, is said to have type ∆α.α→α:

\[
\begin{align*}
\text{id} & \in \Delta \alpha. \alpha \to \alpha \\
\text{id [Nat]} & \in \text{Nat} \to \text{Nat} \\
\text{double} & \in \Delta \alpha. (\alpha \to \alpha) \to \alpha \to \alpha \\
\text{double [Bool]} & \in (\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool}
\end{align*}
\]

The language we are introducing, generally called the second-order polymorphic λ-calculus (or often just the polymorphic λ-calculus, since many authors do not consider related languages of order higher than two), is *explicitly* rather than *implicitly* polymorphic. In languages with explicit polymorphism, type quantifiers like ∆ actually appear in type expressions and correspond to actual type abstractions with Λ. A polymorphic function must be applied explicitly to a type argument to give a monomorphic instance, which can then be applied to term arguments. On the other hand, implicitly polymorphic languages (notably ML [23]) generally omit types from the concrete syntax. Implicitly polymorphic functions may be applied directly to terms of different types; the task of determining the intended monomorphic instance is left to the interpreter/compiler. This topic is explored more fully in Section 6.3.

The polymorphic λ-calculus was invented by Girard in 1971 [20,18] and independently reinvented by Reynolds in 1974 [53]. Girard, a logician, was trying to extend the well-known Curry-Howard isomorphism between propositions and types [29,14, Section 9E], by regarding the binding operator ∆α as a universal quantifier ranging over propositions. Reynolds, a computer scientist, developed essentially the same system by formalizing the idea of “passing types as parameters” in a programming language. Second-order type systems have been the object of much recent research. Reynolds [52,51] and Cardelli and Wegner [8] have written excellent introductions to the area. (Further readings are cited in the bibliographies of these papers.)
Chapter 3

The Second-order Polymorphic λ-Calculus

Consider the following simple $F^+_1$ functions:

\[
\begin{align*}
\text{id}_\text{Nat} & \equiv \lambda x : \text{Nat}. \ x \\
\text{double}_\text{Nat} & \equiv \lambda f : \text{Nat} \to \text{Nat}. \ \lambda x : \text{Nat}. \ f(f \ x)
\end{align*}
\]

The first of these denotes the identity function on numbers, while the second takes a function on numbers and applies it twice. These functions would make perfect sense with \text{Bool}, \text{Pair}_\text{Nat}, or indeed any type whatsoever in place of \text{Nat}. Unfortunately, to express the same operations on \text{Bools}, another pair of \textit{essentially identical} functions must be written. Suppose instead we wanted to define the identity and doubling functions once for all types. We could start by replacing \text{Nat} by a variable, say $\alpha$,

\[
\begin{align*}
\text{id}_\alpha & \equiv \lambda x : \alpha. \ x \\
\text{double}_\alpha & \equiv \lambda f : \alpha \to \alpha. \ \lambda x : \alpha. \ f(f \ x)
\end{align*}
\]

giving two term schemas, each with an infinite number of instances. Now, in each situation where $\text{id}_\alpha$ or $\text{double}_\alpha$ is used, $\alpha$ is replaced by some actual type $T$. Instead we may extend the language so that $\alpha$ is explicitly abstracted; for example, we think of $\text{id}$ as a \textit{function} from types $T$ to terms $\text{id}_T$. To remind ourselves that the argument to this function is a type rather than a term, we write the abstraction operator with a capital $\Lambda$ instead of the usual $\lambda$:

\[
\begin{align*}
\text{id} & \equiv \Lambda \alpha : *. \ \lambda x : \alpha. \ x \\
\text{double} & \equiv \Lambda \alpha : *. \ \lambda f : \alpha \to \alpha. \ \lambda x : \alpha. \ f(f \ x)
\end{align*}
\]

When we want to apply the identity function to an actual (term) argument, we must first \textit{instantiate} it to one of its instances by supplying a type argument. Again, to remind ourselves that this is a different sort of application than before, we enclose type arguments in square brackets:

\[
\begin{align*}
\text{id} [\text{Nat}] & \Rightarrow \text{id}_\text{Nat} \\
\text{id} [\text{Nat}] \ 5 & \Rightarrow 5 \\
\text{double} [\text{Bool}] & \Rightarrow \text{double}_\text{Bool}
\end{align*}
\]
A tree may be totally empty, or may consist of a node and \( n \) subtrees. In the latter case, the tree is constructed by specifying \( n \) along with a function mapping each natural number \( 0 \leq i \leq n \) to the \( i \)th subtree.

To provide an easier way of constructing trees, we can define a function \( \text{build} \) that takes a list of trees and constructs a new tree with these trees as children:

\[
\text{build} \equiv \lambda l:\text{List.Tree.}
\begin{align*}
\text{node} \\
\text{(length } 1) \\
\text{(} \lambda n:\text{Nat. nth } 1 n \text{ empty)}
\end{align*}
\]

where \( \text{nth} \) is the function that takes a \( \text{List.Tree } 1 \), a number \( n \), and a default \( \text{Tree} \) to be returned in case \( 1 \) has less than \( n \) elements.

**Exercises 2.3.6:**

1. Define \( \text{nth} \).
2. Write a function that counts the number of nodes in a \( \text{Tree} \).
3. Extend the definition of \( \text{Tree} \) to include a numerical value at each leaf. Write a function that flattens a tree into a list of the values encountered during a depth-first left-to-right traversal.

The reader may wonder why we have used the rather complicated notion of iterators rather than just adding a \text{case} construct (a la ML or Pascal) to the language. The reason is that \text{case} is only usable for expressing computations over inductive data types when the language also has a construct for defining recursive functions (sometimes called \text{lerec} or \text{labels}). But such a construct would destroy the important property that all \( F_1 \) programs are strongly normalizing. Iteration, on the other hand, preserves this property. Primitive recursion, which also preserves strong normalization, could have been built into the language instead of iteration. But since primitive recursion can be defined in terms of iteration, we prefer to avoid the extra complication.
every primitive recursive function on an inductively defined type in terms of iteration and pairing.

First, we define a type of pairs of lists of numbers:

\[
\text{indtype } \text{Pair(List(Nat)):} \ast \text{ with }
\]
\[
\text{pair(List(Nat): List(Nat) \rightarrow List(Nat) \rightarrow Pair(List(Nat))}
\]

(with projection functions \text{fst(List(Nat))} and \text{snd(List(Nat))} as above).

The method of iterative definition forces us to start at the end of the list and build our result backwards. At each successive \text{cons}, it is not enough to know the \text{cdr} of the second argument to the \text{cons}: we need the second argument \textit{itself}. But when we finish and return the final result, it is not enough to have built up a new copy of the list itself: this time we want the \text{cdr}. The trick is to maintain both pieces of information in parallel. We shall iterate over a \( l \in \text{List(Nat)} \) producing successive pairs of \text{List(Nats)}. The first element of each pair is the \text{cdr(Nat)} of \( l \), while the second is \( l \) itself. A new pair \( p' \) is computed from the old \( p \) by pairing \((\text{snd(List(Nat)} p)\) and \((\text{cons(Nat)} c (\text{snd(List(Nat)} p))\). We will see in Section 3.2 that this corresponds in a fairly natural way to defining a function by primitive recursion.

\[
\text{cdr(Nat)} = \lambda l : \text{List(Nat).}
\]
\[
\text{fst(List(Nat)} (\text{iterList(Nat)[Pair(List(Nat)}]) l
\]
\[
(\text{pair(List(Nat)} \text{nil(Nat)} \text{nil(Nat)})
\]
\[
(\lambda c : \text{Nat.} \lambda r : \text{Pair(List(Nat).}
\]
\[
\text{pair(List(Nat)}
\]
\[
\text{snd(List(Nat)} r)
\]
\[
(\text{cons(Nat)} c (\text{snd(List(Nat)} r)))
\]

\textbf{Exercises 2.3.5:}

1. What is \((\text{cdr(Nat)} \text{nil(Nat)})\)?
2. Define a predecessor function on natural numbers.
3. Define an inductive type of binary trees with natural numbers as leaves. Write a function that sums the leaves of a tree. Write a function that extracts the right subtree of a tree.
4. Define a function that, given \( n \in \text{Nat} \), computes the \( n \)th Fibonacci number.

By considering constructors that take functions as arguments, we can expand the space of definable data types still further. For example, here is the type of arbitrarily branching finite trees (that is, trees in which each node may have any finite number of children):

\[
\text{indtype Tree:* with}
\]
\[
\text{empty: Tree}
\]
\[
\text{node: Nat} \rightarrow (\text{Nat} \rightarrow \text{Tree}) \rightarrow \text{Tree}
\]
For example, here is the type of pairs of numbers:

\[ \text{indtype } \text{Pair}_\text{Nat} : * \text{ with } \text{pair}_\text{Nat} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Pair}_\text{Nat} \]

The projection functions for \text{Pair}_\text{Nat} are easy to define by iteration:

\[ \text{fst}_\text{Nat} \equiv \lambda p: \text{Pair}_\text{Nat}. \ \text{iterPair}_\text{Nat}[\text{Nat}] \ p \ (\lambda f: \text{Nat}. \ \lambda s: \text{Nat}. \ f) \]
\[ \text{smd}_\text{Nat} \equiv \lambda p: \text{Pair}_\text{Nat}. \ \text{iterPair}_\text{Nat}[\text{Nat}] \ p \ (\lambda f: \text{Nat}. \ \lambda s: \text{Nat}. \ s) \]

Similarly, the type of finite lists of numbers is defined by:

\[ \text{indtype } \text{List}_\text{Nat} : * \text{ with } \]
\[ \text{nil}_\text{Nat} : \text{List}_\text{Nat} \]
\[ \text{cons}_\text{Nat} : \text{Nat} \rightarrow \text{List}_\text{Nat} \rightarrow \text{List}_\text{Nat} \]

All the usual list manipulation functions can be defined on this representation. For example:

\[ \text{car}_\text{Nat} \equiv \lambda l: \text{List}_\text{Nat}. \ \lambda d: \text{Nat}. \]
\[ \text{iterList}_\text{Nat}[\text{Nat}] \ l \ d \ (\lambda c: \text{Nat}. \ \lambda r: \text{Nat}. \ c) \]

The "default" parameter \( d \) is needed so that \text{car} will have something to return if it happens to be passed an empty list.

**Exercises 2.3.3:**

1. Check carefully that the types of the arguments to the iterations in \text{fst}_\text{Nat} and \text{car}_\text{Nat} correspond to Definition 2.2.1.

2. What is the purpose of the parameter \( r \) in the inner \( \lambda \)-abstraction above? Why does it have type \text{Nat}?

3. Define an \text{append}_\text{Nat} function that takes two lists of numbers and returns their concatenation.

4. Define a function \text{map}_\text{Nat} \in \text{List}_\text{Nat} \rightarrow (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{List}_\text{Nat} that takes a list \( l \) of numbers and a numeric function \( f \), and returns the list formed by applying \( f \) to each element of \( l \).

Somewhat unexpectedly, the definition of \text{cdr}_\text{Nat} turns out to be quite a bit more complicated than \text{car}_\text{Nat}.

**Exercise 2.3.4:** Readers are encouraged to pause here before reading further and try to see why this is so. What goes wrong with a simple definition of \text{cdr}_\text{Nat} in terms of \text{iterList}_\text{Nat}[\text{List}_\text{Nat}]? Is there a way to fix it?

The solution introduces a very important trick (important enough to be promoted to a "technique"), which we will use again and again in the rest of the tutorial. It was first used by Kleene [9] to define a predecessor function on the Church Numerals, which are essentially the same as our inductive type \text{Nat}. In general terms, it allows us to express
2.3 Programming with Iterators

Of course, primitive data types other than Nat can also be defined inductively. The type Bool has a particularly simple form, where all of the constructors are constants:

\[ \text{indtype} \text{ Bool} :: * \text{ with } \text{ true}: \text{ Bool} \text{ and } \text{ false}: \text{ Bool} \]

Using \texttt{iterator[Nat]}, we can define an "if...then...else" construct for choosing between two numbers on the basis of some test:

\[ \text{if}. \text{ Nat} \equiv \lambda b: \text{ Bool}. \lambda t: \text{ Nat}. \lambda e: \text{ Nat}. \]
\[ \text{iterator[Nat]} \ b \ t \ e \]

(Of course, it is equally easy to define if...then...else constructs for selecting between values of other types. However, a separate definition is required for each one because at this stage we don’t have any way to parameterize functions with respect to types. Consequently, we use this underscore notation to include the type being returned is part of the name of the iterator.)

Exercise 2.3.1: Use iterator[Bool] to define the binary and function.

An even simpler data type is Unit, which has just one constructor:

\[ \text{indtype} \text{ Unit} :: * \text{ with } \text{ unit}: \text{ Unit} \]

Unit is often used in statically-typed languages with imperative constructs (e.g., Standard ML [23]) as the result type of functions that are executed purely for their side effects.

Continuing in the same vein, there is one even simpler inductive type, called Void, which has no constructors at all:

\[ \text{indtype} \text{ Void} :: * \]

(Obviously, no term in F1 can ever have type Void, but that does not prevent us from defining it.)

Exercise 2.3.2: Define a type Day with constant constructors sunday, monday, ..., saturday. Write a function weekday \( \in \text{ Day} \rightarrow \text{ Bool} \) that returns true if its argument is in the range monday...friday.

A variety of other useful types can be defined if we expand our horizons to include more complicated inductive definitions. Additionally, it is often important to define destructors or projection functions—functions that take a term built up using constructors, and pull it apart.
a "negative" position—that is, inside of an odd number of negations—if and only if the whole implication appears inside an even number of negations.

With the notion of inductivity in hand, we can now complete the definition of $F^i_1$.

**Definition 2.2.1, continued:** The general form of an indtype definition is:

```
indtype T::* with
  c_1:U_{i_1}\rightarrow U_{i_2}\rightarrow \cdots \rightarrow U_{i_{n_1}}\rightarrow T
  and c_2:U_{i_2}\rightarrow U_{i_3}\rightarrow \cdots \rightarrow U_{i_{n_2}}\rightarrow T
  \vdots
  and c_m:U_{i_1}\rightarrow U_{i_2}\rightarrow \cdots \rightarrow U_{i_{n_m}}\rightarrow T
```

Note that the type being defined must appear as the rightmost component of the type of each constructor, and may appear positively (but not negatively) in the $U_{i_j}$'s.

Each such definition introduces the following globally-bound identifiers:

1. The type $T$.
2. Zero or more constructors $c_i$. Each $c_i$ takes zero or more arguments (of types $U_{i_1} \ldots U_{i_{n_i}}$, respectively), and produces a result of type $T$.
3. An iteration scheme $\text{iter}_T[\alpha]$ with infinitely many instances $\text{iter}_T[V]$, called iteration operators—one for each type $V$. An $\text{iter}_T[V]$ takes one argument for each constructor of $T$ (in order), and returns a result of type $V$. More formally, the type of $\text{iter}_T[V]$ is

```
\text{iter}_T[V] \in T \rightarrow (\hat{U}_{i_1} \rightarrow \hat{U}_{i_2} \rightarrow \cdots \rightarrow \hat{U}_{i_{n_1}} \rightarrow V)
\rightarrow (\hat{U}_{i_2} \rightarrow \hat{U}_{i_3} \rightarrow \cdots \rightarrow \hat{U}_{i_{n_2}} \rightarrow V)
\vdots
\rightarrow (\hat{U}_{i_{n_1}} \rightarrow \hat{U}_{i_{n_2}} \rightarrow \cdots \rightarrow \hat{U}_{i_{n_m}} \rightarrow V)
\rightarrow V
```

where $\hat{U}$ denotes the result of substituting $V$ for all occurrences of $T$ in $U$.

An iterator $\text{iter}_T[V]$ associates a function building up values of type $V$ with each constructor for type $T$. It pulls apart a term of type $T$, constructor by constructor, and applies the function associated with that constructor to the arguments, but only after recursively applying $\text{iter}_T[V]$ within them on all subterms of type $T$.

We can express this as one reduction rule:

```
\text{iter}_T[V] (c_i a_1 \ldots a_{n_i}) e_1 \ldots e_m \Rightarrow e_i \hat{a}_1 \ldots \hat{a}_{n_i}
```

where $c_i$ is the $i^{th}$ constructor for type $T$, $a_1$ to $a_{n_i}$ are its arguments, and $e_i \in \hat{U}_{i_1} \rightarrow \cdots \rightarrow \hat{U}_{i_{n_i}} \rightarrow V$ is the function corresponding to this constructor, and where $\hat{a}$ is the result of replacing, in $a$, each subterm $t$ of type $T$ with $\text{iter}_T[V] t e_1 \ldots e_m$. 

The definitions of `plus` and 2 are simply abbreviations, as before. We have allowed ourselves to write these macro definitions at any convenient point in the program text. However, the `indtype` definitions at the beginning of a program actually define a *global environment* in which the body of the program is evaluated. This points out a major difference between symbols defined as global abbreviations and the types and constructors introduced by `indtype` definitions: the latter cannot be expanded away; indeed, they must appear in fully-normalized programs since there are no primitive datatypes.

Having looked at `Nat`, we are ready to define the general form of inductive type definitions and iterators.\(^2\)

It is important to distinguish between *inductive* types and the more general class of *reflexive* types. It is not the case that every instance of the `indtype` syntax actually defines an inductive type. For example,

```
indtype T:* with
c: (T→T)→T
```

is reflexive, but not inductive.

Formally, an inductive type definition is one where the type being defined appears only *positively* in the types of the arguments to the constructors. The notions of positive and negative occurrences may be formulated within a pair of mutually-recursive functions.\(^3\)

**Definition 2.2.2:** The set `Pos(U)` of positively occurring variables in an F\(\dagger\) type expression \(U\) is defined by:

\[
\begin{align*}
\text{Pos}(\alpha) &= \{ \alpha \} \quad \text{(where } \alpha \text{ is a type variable)} \\
\text{Pos}(V→W) &= \text{Neg}(V) \cup \text{Pos}(W)
\end{align*}
\]

The set `Neg(U)` of negatively occurring variables in a type expression \(U\) is defined by:

\[
\begin{align*}
\text{Neg}(\alpha) &= \{ \} \quad \text{(where } \alpha \text{ is a type variable)} \\
\text{Neg}(V→W) &= \text{Pos}(V) \cup \text{Neg}(W)
\end{align*}
\]

A type variable \(\alpha\) is said to *appear positively* in \(U\) if \(\alpha \in \text{Pos}(U)\) and to *appear negatively* in \(U\) if \(\alpha \in \text{Neg}(U)\).

The words “positive” and “negative” come from logic. According to the well-known “Curry-Howard isomorphism” [29,14, Section 9E] between propositions and types, the type \(A \rightarrow B\) corresponds to the logical proposition \(A \supset B\), which, by the definition of logical implication, is equivalent to \(\neg A \vee B\). The subproposition \(A\) here is obviously in

\(^2\)Any many-sorted first-order algebraic signature without laws (or “heterogeneous free algebra”) can be considered as an inductively defined type [4].

\(^3\)The technical intuition behind the definition is roughly as follows. A data type definition of the form we have described can be translated into a function on the lattice of types. If the definition has the form of an *inductive* type definition, then this function will be covariant, and hence (by Tarski’s fixed point theorem) will be guaranteed to have both a least and a greatest fixed point.
Definition 2.2.1: The syntax of $F_1$ is given by the following inductively defined classes:

\[
\begin{align*}
P & ::= \ I \ e \ | \ I \ P \\
I & ::= \ \text{indtype} \ \alpha : \ * \ | \ \text{indtype} \ \alpha : \ * \ \text{with} \ C \\
C & ::= \ x : T \ | \ x : T \ \text{and} \ C \\
T & ::= \ \alpha \ | \ T \rightarrow T' \\
e & ::= \ x \ | \ \lambda x : T. \ e \ | \ e \ e'
\end{align*}
\]

where $P$ ranges over programs, $I$ ranges over (lists of) inductive type definitions, $C$ ranges over (lists of) constructors, $\alpha$ ranges over type variables, $T$ ranges over types, $e$ ranges over expressions, and $x$ ranges over variables.

An $F_1$ program consists of a sequence of global $\text{indtype}$ definitions, followed by an expression.

The type of natural numbers can be defined in $F_1$ as follows:\(^1\)

\[
\text{indtype} \ \text{Nat} : \ * \ \text{with} \ \text{zero} : \text{Nat} \ \text{and} \ \text{succ} : \text{Nat} \rightarrow \text{Nat}
\]

This is precisely what we had in $F_1^+$, except that the $\text{iter}$ of $F_1^+$ becomes $\text{iterNat}[\text{Nat}]:$

\[
\text{plus} \equiv \ \lambda x : \text{Nat}. \ \lambda y : \text{Nat}. \ \text{iterNat}[\text{Nat}] \times y \ \text{succ}
\]

Instead of a single $\text{iter}$ function, $F_1$ has, for each inductively defined type $T$, an infinite number of iteration functions—one called $\text{iterT}[V]$ for every type $V$. (For now, the square brackets and the type $V$ should be thought of as part of the name of the iterator.) Each $\text{iterT}[V]$ performs structural induction on elements of type $T$, returning a value of type $V$ as the result of the induction.

To illustrate how a different instance of the iteration scheme for $\text{Nat}$ might be used, here is an alternative definition of addition:

\[
\begin{align*}
\text{plus} & \equiv \ \lambda x : \text{Nat}. \\
& \ \text{iterNat}[\text{Nat} \rightarrow \text{Nat}] \\
& \ x \\
& \ (\lambda y : \text{Nat}. \ y) \\
& \ (\lambda x : \text{Nat}. \ \lambda y : \text{Nat}. \ \text{succ} \ (x \ y))
\end{align*}
\]

This version uses $\text{iterNat}[\text{Nat} \rightarrow \text{Nat}]$ to construct a function that applies $\text{succ} \times x$ times to its argument $y$.

An example of a complete program in $F_1$ is:

\[
\begin{align*}
\text{indtype} & \ \text{Nat} : \ * \ \text{with} \ \text{zero} : \text{Nat} \ \text{and} \ \text{succ} : \text{Nat} \rightarrow \text{Nat} \\
\text{plus} & \equiv \ \lambda x : \text{Nat}. \ \lambda y : \text{Nat}. \ \text{iterNat}[\text{Nat}] \times y \ \text{succ} \\
2 & \equiv \ \text{succ} \ (\text{succ} \ \text{zero}) \\
\text{plus} \ 2 \ 2
\end{align*}
\]

\(^1\)The $*$ can be ignored for now. It just indicates that we are defining a type. Later on we use $\text{indtype}$ to define more complicated things as well.
\[ \lambda x: \text{Nat}. \lambda y: \text{Nat}. \text{iter } x \ y \ \text{succ} \]

Given \( x \) and \( y \), the \text{iter} expression has the effect of taking \( x \) successors of \( y \).

This example also illustrates the trick of \textit{currying} multi-argument functions into single-argument functions. For example, instead of taking both of its arguments at once, the curried \text{plus} function accepts \( x \) and returns a function from \( y \) to \( x+y \). In general, a function of \( n \) arguments of types \( T_1, \ldots, T_n \) returning an answer of type \( T_0 \) has type:

\[ T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_0 \]

\textbf{Exercises 2.1.8:}

1. Define a function that sums three numbers.
2. Define a function that multiplies two numbers.

To make our programs more manageable, we allow both terms and types to be abbreviated by individual symbols. For example:

\[
\begin{align*}
\text{BinaryFun} & \equiv \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
\text{plus} & \in \text{BinaryFun} \\
\text{plus} & \equiv \lambda x: \text{Nat}. \lambda y: \text{Nat}. \text{iter } x \ y \ \text{succ} \\
\text{times} & \in \text{BinaryFun} \\
\text{times} & \equiv \lambda x: \text{Nat}. \lambda y: \text{Nat}. \text{iter } x \ 0 \ (\lambda r: \text{Nat}. \text{plus } r \ y)
\end{align*}
\]

These abbreviations should be thought of as global macro definitions that can be completely expanded away without affecting the meaning of any term that mentions them. (They should not be thought of as global definitions in the sense of ML or Scheme. In particular, they may not be recursive.)

\section*{2.2 Inductive Type Definitions}

We have seen that \( F_1^+ \) can express some useful programs. But it leaves something to be desired in the way of available data types. We chose the primitive types and terms somewhat arbitrarily and then enshrined this choice in the very definition of the language, with no provision for extending the available types short of defining an entirely new language.

In this section we take a more general approach, adding to \( F_1 \) a general type definition facility instead of a particular set of predefined types. We use the keyword \texttt{indtype} to introduce an "inductively defined" type. The rest of the language remains as before.
Together, Theorems 2.1.5 and 2.1.6 guarantee that every well-typed term reduces in a finite number of steps to a unique normal form. This stands in sharp contrast to the untyped \(\lambda\)-calculus, where non-normalizable terms like

\[
\Omega = (\lambda x. \ x) \ (\lambda x. \ x)
\]

are easy to construct [2].

The constant \(\text{iter}\) provides iteration over natural numbers, which can be used to implement all the ordinary primitive recursive functions on numbers. It takes three arguments: a number \(x\) to "iterate over," a number \(z\) to be returned in case \(x\) is zero, and a function \(f\) to be "iterated" in case \(x\) is non-zero. For example:

\[
\text{iter} \ x \ (\text{succ} \ z) \ (\lambda r:\text{Nat}. \ \text{succ} \ (\text{succ} \ r))
\]

The reduction rules for \(\text{iter}\) are as follows.

\[
\begin{align*}
\text{iter zero} \ z \ e \ &\Rightarrow z \\
\text{iter} \ (\text{succ} \ x) \ z \ e \ &\Rightarrow (e \ (\text{iter} \ x \ z \ e))
\end{align*}
\]

Strictly speaking, these should have been included in our discussion above of reduction, conversion, strong normalization, and so on. But since our main task in Chapter 3 is to show how inductive types (like \(\text{Nat}\)) and iteration over them can be eliminated from the core language, we prefer to discuss them separately. The reduction of an \(\text{iter}\) expression is guaranteed to terminate (i.e. strong normalization is maintained), since when the second case applies, the first argument to \(\text{iter}\) is reduced by one. Hence the first case must eventually apply. It is easy to see that the result will be \(x\) applications of \(e\) to \(z\). More graphically, if \(x\) is

\[
\text{succ} \ (\text{succ} \ (\text{succ} \ \ldots \ (\text{succ} \ \text{zero} \ \ldots)))
\]

then \(\text{iter} \ x \ z \ e\) has exactly the same structure as \(x\), with the \(\text{zero}\) replaced by \(z\) and each \(\text{succ}\) replaced by \(e\):

\[
e \ (e \ (e \ \ldots \ (e \ z) \ \ldots))
\]

Returning to the example,

\[
\lambda x:\text{Nat}. \ \text{iter} \ x \ (\text{succ} \ z) \ (\lambda r:\text{Nat}. \ \text{succ} \ (\text{succ} \ r))
\]

denotes a function that returns \(2n+1\) when applied to a number \(n\).

**Exercises 2.1.7:**

1. Define a function that returns \(3n\) when applied to \(n\).

2. Define a function that returns one when its argument is zero, and zero otherwise.

Similarly, a function that adds two numbers can be defined using \(\text{iter}\):

\[
\lambda x:\text{Nat}. \ \lambda y:\text{Nat}. \\
\text{iter} \ x \ y \ (\lambda r:\text{Nat}. \ \text{succ} \ r)
\]

or more simply (by \(\eta\)-conversion):
A term e is one-step $\beta$-reducible to e' if e' can be obtained from e by a single application of the rule for $\beta$-reduction to a subterm of e. A term e_0 is $\beta$-reducible to e_n (written e_0 $\Rightarrow_{\beta}$ e_n) if there is a reduction sequence e_0 $\Rightarrow_{\beta}$ e_1 $\Rightarrow_{\beta}$ e_2 $\Rightarrow_{\beta}$ ... $\Rightarrow_{\beta}$ e_n (with n $\geq$ 0) where each element one-step $\beta$-reduces to the next. One-step $\beta$-conversion and $\beta$-conversion are defined similarly, but allow $\beta$-reduction to be applied in either direction. Two terms are $\beta$-equivalent (written $\equiv_{\beta}$) if one can be $\beta$-converted to the other. The definitions of $\eta$-reduction, $\eta$-conversion, and $\eta$-equivalence are similar; the definitions of $\beta\eta$-reduction, $\beta\eta$-conversion, and $\beta\eta$-equivalence allow the two rules to be intermixed. We often write just = instead of $\equiv_{\beta\eta}$. A term is in $\beta\eta$-normal form if it contains no $\beta$- or $\eta$-redexes.

Throughout this document we frequently blur the distinction between terms and their denotations. For example, we will speak of an expression like

$$\lambda x : \text{Nat.} \; \text{succ} \; (\text{succ} \; x)$$

as being the function that adds two to its argument, when, more properly, we should say that the expression denotes this function in some mathematical model we have in mind, or else that when applied to a term representing a number n, it reduces to a term representing n + 2.

Terms differing only in the names of bound variables are said to be $\alpha$-equivalent. (The renaming of bound variables within a term is often called $\alpha$-conversion.) Following standard practice [2,28], we consider $\alpha$-equivalent terms to be identical.

$F_1$ and $F_1^+$ share a number of interesting theoretical properties with the other languages we consider in this tutorial. Perhaps the most important is the fact that only terminating computations can be expressed:

**Definition 2.1.4:** A term is strongly normalizable (under a given set of reduction rules) if every sequence of reductions beginning with that term reaches a normal form after a finite number of steps. A set of reduction rules is strongly normalizing if there is no infinite reduction sequence on any term.

**Theorem 2.1.5:** The rules given above are strongly normalizing (i.e., there is no infinite $\beta\eta$-reduction on any term).

The proof of this theorem for $F_1$ and $F_1^+$ is fairly straightforward (see [28], for example). Intuitively, $\eta$-reduction always decreases the size of a term, while $\beta$-reduction may increase its size but always decreases the nesting of arrows in the types of bound variables. The general proof for $F_n$ is much more delicate [17,18,19,31].

The next theorem assures us that it does not matter which redexes in a term are reduced first:

**Theorem 2.1.6:** (Church-Rosser) For any well-typed term e, if e $\Rightarrow_{\beta\eta}$ e_1 and e $\Rightarrow_{\beta\eta}$ e_2 then there exists a term e' such that e_1 $\Rightarrow_{\beta\eta}$ e' and e_2 $\Rightarrow_{\beta\eta}$ e'.
is a valid judgment. Within rule names the I stands for ‘introduction’ and the E for ‘elimination.’

\[
\begin{align*}
(\text{Var}) & & \Pi \vdash x \in T & \text{when } \Pi(x) = T \\
(-I) & & \Pi \vdash x \in T & \Pi[x/T] \vdash e \in T' \\
(+E) & & \Pi \vdash e \in T \rightarrow T' & \Pi \vdash \lambda x.T.e \in T \rightarrow T' \\
(\text{Zero}) & & \Pi \vdash \text{zero} \in \text{Nat} \\
(\text{Suc}) & & \Pi \vdash \text{succ} \in \text{Nat} \rightarrow \text{Nat} \\
(\text{Iter}) & & \Pi \vdash n \in \text{Nat} & \Pi \vdash z \in \text{Nat} & \Pi \vdash s \in \text{Nat} \rightarrow \text{Nat} \\
& & \Pi \vdash \text{iter} \ n \ z \ s \in \text{Nat} \\
\end{align*}
\]

To show how these deduction rules are used, we use them to prove that 

\[\lambda n: \text{Nat. succ (succ n)} \in \text{Nat} \rightarrow \text{Nat}.\]

1. \(\langle n, \text{Nat} \rangle \vdash \text{succ} \in \text{Nat} \rightarrow \text{Nat}\) by \((\text{Suc})\)
2. \(\langle n, \text{Nat} \rangle \vdash n \in \text{Nat}\) by \((\text{Var})\)
3. \(\langle n, \text{Nat} \rangle \vdash \text{succ} \ n \in \text{Nat}\) by \((\rightarrow\text{E})\) from 1,2
4. \(\langle n, \text{Nat} \rangle \vdash \text{succ (succ n)} \in \text{Nat}\) by \((\rightarrow\text{E})\) from 1,3
5. \(\vdash \lambda n: \text{Nat. succ (succ n)} \in \text{Nat} \rightarrow \text{Nat}\) by \((\rightarrow\text{I})\) from 4

**Exercises 2.1.3:**

1. Prove that:
\[\lambda f: \text{Nat} \rightarrow \text{Nat.} \ \lambda a: \text{Nat.} \ f \ a \in (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat} \rightarrow \text{Nat}\]
2. Prove that:
\[\lambda x: \text{Nat. iter x (succ zero) (\lambda n: \text{Nat. succ (succ n)})} \in \text{Nat} \rightarrow \text{Nat}\]

We define the operational meaning of programs via reduction rules. A \(\beta\)-\textit{redex} is a term \(t\) of the form:

\[(\lambda x: T. e) \ a\]

It is \(\beta\)-\textit{reduced} (or just \textit{reduced}) according to the rule

\[(\lambda x: T. e) \ a \Rightarrow_\beta e[x/a]\]

where \(e[x/a]\) is the term obtained by replacing each free occurrence of \(x\) in \(e\) by \(a\), and renaming any bound variables in \(e\) as necessary to prevent capture of free variables in \(a\).

An \(\eta\)-\textit{redex} is a term of the form

\[\lambda x: T. e \ x\]

(where \(x\) is not free in \(e\)). It is \(\eta\)-\textit{reduced} (or \textit{reduced}) according to the rule

\[\lambda x: T. e \ x \Rightarrow_\eta e\]

(when \(x\) is not free in \(e\)).
where $T$ ranges over types, $e$ ranges over expressions, and $x$ ranges over variables.

A typical term of $\text{F}_1^+$ is
\[\lambda n: \text{Nat}. \, \text{succ} \ (\text{succ} \ n)\]
which denotes the function that, given a number, increases it by two. The whole term is a $\lambda$-abstraction with bound variable $n$ whose body (or scope) consists of the two nested applications ($\text{succ} \ (\text{succ} \ n)$).

The rest of this section uses $\text{F}_1^+$ to review some notations and conventions of the $\lambda$-calculus. (Hindley and Seldin [28] provide a more thorough introduction to the basics of both typed and untyped $\lambda$-calculi. Barendregt [2] is an excellent reference on the untyped $\lambda$-calculus.) Definitions 2.1.1 and 2.1.2 specify the abstract syntax of $\text{F}_1$ and $\text{F}_1^+$; we deal with questions of concrete syntax and parsing (e.g., precedence and associativity) informally. Following standard practice, the $\to$ symbol associates to the right and application associates to the left. Parentheses are used when necessary to override these conventions. The body of a $\lambda$-abstraction extends as far to the right as possible—to the end of the whole expression, or up to an unmatched right parenthesis.

An occurrence of a variable $x$ is bound if it appears in the scope of a $\lambda$-abstraction with bound variable $x$, and free otherwise. A closed $\lambda$-term is one with no free variables.

The abstract syntax given in Definitions 2.1.1 and 2.1.2 allows us to write meaningless programs like $(\text{succ} \ \text{succ})$. We focus our attention only on the well-typed terms of each of the languages we define. We write the type judgment
\[\Pi \vdash e \in T\]
to indicate that an expression $e$ has type $T$ in the context of the type environment $\Pi$ (which maps variables to types). To denote an explicit environment $\Pi$ we write a list of ordered pairs enclosed in brackets '{' and '}'; separated by commas. An empty environment is written as '(' or is simply omitted. Although $\Pi$ is potentially multiple-valued, we will think of it as single-valued, and we agree that it is searched from right to left to find the appropriate pair. We can think of $\Pi$ as being extended to terms if we agree that
\[\Pi(e) = \alpha \quad \text{where} \quad \Pi \vdash e \in \alpha.\]
$\Pi[x/T]$ denotes the extension of the environment $\Pi$ to $x$ such that $\Pi(x) = T$. By convention pairs are always added to the right end of $\Pi$. In cases where $e$ is a closed term or $\Pi$ is obvious from context we write
\[e \in T.\]

The symbols '{' and '}' have intuitively similar meanings, since both declare something to have a particular type. The difference between them is that '{' is part of the object language—it is used within terms to declare the types of bound variables—whereas '}' is a notation of the metalanguage used to make statements (e.g., type judgments) about terms.

We formally define the typing of $\text{F}_1^+$ via type inference rules. Each of the following rules has the property that, if the premises are all valid type judgments, then the conclusion
Chapter 2

The Simply-typed $\lambda$-Calculus

We begin by defining a simple programming language and studying some of its properties. We then extend this language with a powerful facility for defining data types inductively.

2.1 Definitions and Properties

The purest form of Church's simply-typed $\lambda$-calculus [10], which we call $F_1$, may be defined as follows:

**Definition 2.1.1:** The syntax of $F_1$ is given by the following inductively defined classes:

\[
T ::= T \rightarrow T' \\
e ::= x | \lambda x : T. e | e e'
\]

where $T$ ranges over *types*, $e$ ranges over *expressions* (also called *terms*), and $x$ ranges over *variables*. An expression of the form $\lambda x : T. e$ is called a *$\lambda$-abstraction*; $e e'$ is an *application*.

The language of Definition 2.1.1 is theoretically important because it forms the base of an infinite sequence of more and more powerful languages culminating in $F_\omega$. But from a practical standpoint, there is a major problem. The equation defining $T$ is an inductive definition with no base case (no constant types), so the set of types is empty. Furthermore, since $\lambda$-abstractions are typed, the set of $\lambda$-abstractions is empty. In order to do anything useful with $F_1$, we need to add some primitive types and terms. We do this first in an *ad hoc* way, adding a single constant type and some constant terms to $F_1$ to form a language we call $F_1^+$:

**Definition 2.1.2:** The syntax of $F_1^+$ is given by the following inductively defined classes:

\[
T ::= \text{Nat} | T \rightarrow T' \\
e ::= \text{zero} | \text{succ} | \text{iter} | x | \lambda x : T. e | e e'
\]
credit for most of the constructions we describe. We are also indebted to Luca Cardelli, Bob Harper, John Mitchell, and John Reynolds for helpful discussions and technical guidance, and to Tim Freeman, Bob Harper, Nevin Heintze, Peter Lee, David Long, and Frank Pfenning for suggesting improvements to the text.
a number of practical programming languages [7,24], but they are normally embellished with a number of built-in types and type constructors. In view of the recent work on representing data structures, it now makes sense to ask whether the pure higher-order typed λ-calculi might form a suitable basis of a practical language for program and proof manipulation [49].

The pure polymorphic λ-calculi all share the property that every reduction sequence terminates after a finite number of steps. This implies that only total functions are definable, and that the familiar control construct of general recursive definition is not available. Instead, functions must be expressed in terms of primitive recursion (or iteration) over inductively defined data structures. Our central purpose in this tutorial is to explore the unusual programming style that arises from these constraints.

Chapter 2 of the tutorial introduces the simply typed λ-calculus (called F₁ here), reviews some of its properties, and establishes basic notational conventions. This chapter also introduces the notion of an "inductively defined type" and shows how values of such types can be manipulated using a basic iteration construct. Chapter 3 introduces the polymorphic λ-calculus (F₂), and shows how inductive type definitions over F₁ can be translated into pure F₂. Chapter 4 introduces the third-order polymorphic λ-calculus (F₃) and shows how the techniques of the previous chapter can be generalized to allow inductive type definitions over F₂ to be translated into pure F₃. Chapter 5 uses the principles developed so far to experiment with metaprogramming in F₃—building data structures that can be used to represent and manipulate terms in F₁ and F₂. Chapter 6 completes the hierarchy of languages by discussing the definition and properties of the ω-order polymorphic λ-calculus (Girard's Fₗ without existential quantifiers). Appendix A presents an extended metaprogramming example—a representation of untyped λ-terms in F₂. Appendix B summarizes our notational conventions.

Most sections are supplemented with exercises. We strongly recommend that readers try working most of these, since it has been our experience that the only way to understand programs written in this style is to generate a fair number of them. The tutorial is intended to be self-contained, but the early sections will be easier for readers who are familiar with the basic concepts of the untyped λ-calculus [2,28], or have done some programming in a λ-calculus-based language like Scheme [1,50] or ML [23,26]. Some acquaintance with polymorphic type systems [8,27,52] will also be helpful. Technical details that may not be accessible to all of our readers are placed in footnotes.

This document grew out of discussions in the LEAP Working Group at CMU. It reflects the authors' state of understanding after only a few months of experience in the area, and hence falls lamentably short of a full treatment of any of the subjects it introduces. Furthermore, many the technical results that it presents are subjects of intense current research, which raises the possibility that our present perspective may turn out to be incorrect or misguided in any number of ways. Still, the document represents a significant expansion of our own knowledge, and we hope it will be a useful guide for other newcomers. We welcome corrections and suggestions for clarification.

The other members of the LEAP group—Ken Cline, Peter Lee, and Frank Pfenning—share
Chapter 1

Introduction

Typed $\lambda$-calculi have been objects of theoretical study for many years [10,20,18,53,52,51, etc.]. One of the earliest results in this area was the demonstration that a wide class of number-theoretic functions could be defined in the simply typed $\lambda$-calculus. The basic trick behind these results was a representation of natural numbers as typed terms (the so-called "Church numerals"):

\[
0 \equiv \lambda f : \text{Nat} \to \text{Nat}. \lambda x : \text{Nat}. \ x \\
1 \equiv \lambda f : \text{Nat} \to \text{Nat}. \lambda x : \text{Nat}. \ f(x) \\
2 \equiv \lambda f : \text{Nat} \to \text{Nat}. \lambda x : \text{Nat}. \ f(f(x))
\]

More recently, Böhm and Berarducci [4], and independently Leivant [34], have shown that any set of inductively defined types\(^1\) can be translated into a set of types in the polymorphic $\lambda$-calculus of Reynolds [53] and Girard [18,20]. For example, the standard inductive definition of the natural numbers

\[
\text{indtype Nat:* with} \\
\text{zero: Nat} \\
\text{succ: Nat \to Nat}
\]

can be translated mechanically into the representation above. This technique makes it possible to define a host of commonly used data types—booleans, pairs, lists, trees, and so on—and to express functions over them, even though the pure polymorphic $\lambda$-calculus provides no built-in types whatsoever. (Steensgaard-Madsen [55] presents a similar idea.) Generalizing the technique to higher orders, Pfenning [45] has shown that inductively defined types with polymorphic constructors in the $n^{th}$-order $\lambda$-calculus can be translated into the pure $(n + 1)^{th}$-order $\lambda$-calculus. This further expands the class of definable data structures to include, for example, representations of typed $\lambda$-terms and proofs in higher-order logic.

Variants of the second-order polymorphic $\lambda$-calculus have been used as the foundations of

---

\(^1\)Or more technically, any heterogeneous term algebra.
6.3 Types and Type Inference ........................................ 54
6.4 $F_\omega$ as the Basis for a Programming Language ........ 56

A Representing the Untyped $\lambda$-Calculus .................. 59

B Symbols and Terminology ...................................... 63
# Contents

1 Introduction 3

2 The Simply-typed $\lambda$-Calculus 6
2.1 Definitions and Properties 6
2.2 Inductive Type Definitions 11
2.3 Programming with Iterators 15

3 The Second-order Polymorphic $\lambda$-Calculus 19
3.1 Definitions 21
3.2 Representing $F_1$ Inductive Type Definitions in $F_2$ 23
3.3 The Power of $F_2$ 31

4 The Third-order Polymorphic $\lambda$-Calculus 33
4.1 Definition and Properties of $F_3$ 33
4.2 Polymorphic Inductive Datatypes 35
4.3 Programming in $F_3$ 36

5 $F_3$ as a Metalanguage 41
5.1 A Simple Representation of $F_1$ Terms 41
5.2 A Complete Representation of $F_1$ Terms 43
5.3 Representation of $F_2$ 45
5.4 Representing $F_3$ in $F_4$ 46
5.5 Alternative Formulations of $\text{Term}$ 47

6 The $\omega$-order Polymorphic $\lambda$-Calculus 50
6.1 Basic Definitions 50
6.2 Properties of $F_\omega$ 53