6.2 Linear Type Checking

The typing rules for the linear λ-calculus are *syntax-directed* in that the principal term constructor determines the typing rule which must be used. Nonetheless, the typing rules are not immediately suitable for an efficient type-checking algorithm since we would have to guess how the linear hypotheses are to be split between the hypothesis in a number of rules.

The occurrence constraints introduced in Section **??** would be sufficient to avoid this choice, but they are rather complex, jeopardizing our goal of designing a simple procedure which is easy to trust. Fortunately, we have significantly more information here, since the proof term is given to us. This determines the amount of work we have to do in each branch of a derivation, and we can resolve the don't-care non-determinism directly.

Instead of guessing a split of the linear hypotheses between two premisses of a rule, we pass all linear variables to the first premiss. Checking the corresponding subterm will consume some of these variables, and we pass the remaining ones one to check the second subterms. This idea requires a judgment

$$\Gamma; \Delta_I \setminus \Delta_O \vdash M : A$$

where Δ_I represents the available linear hypotheses and $\Delta_O \subseteq \Delta_I$ the linear hypotheses not used in M. For example, the rules for the simultaneous conjunction and unit would be

$$\frac{\Gamma; \Delta_I \setminus \Delta' \vdash M : A \qquad \Gamma; \Delta' \setminus \Delta_O \vdash N : B}{\Gamma; \Delta_I \setminus \Delta_O \vdash M \otimes N : A \otimes B} \otimes\mathrm{I}$$

$$\frac{}{\Gamma; \Delta_I \setminus \Delta_I \vdash \star : A} \mathbf{1}\mathrm{I}.$$

Unfortunately, this idea breaks down when we encounter the additive unit (and only then!). Since we do not know which of the linear hypotheses might be used in a different branch of the derivation, it would have to read

$$\frac{\Delta_I \supseteq \Delta_O}{\Gamma; \Delta_I \setminus \Delta_O \vdash \langle\,\rangle : \top} \top\mathrm{I}$$

which introduces undesirable non-determinism if we were to guess which subset of Δ_I to return. In order to cirumvent this problem we return all of Δ_I, but flag it to indicate that it may not be exact, but that some of these linear hypotheses may be absorbed if necessary. In other words, in the judgment

$$\Gamma; \Delta_I \setminus \Delta_O \vdash_1 M : A$$

any of the remaining hypotheses in Δ_O need not be consumed in the other branches of the typing derivation. On the other hand, the judgment

$$\Gamma; \Delta_I \setminus \Delta_O \vdash_0 M : A$$

indicates the M uses exactly the variables in $\Delta_I - \Delta_O$.

When we think of the judgment $\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : A$ as describing an algorithm, we think of Γ, Δ_I and M as given, and Δ_O and the slack indicator i as part of the result of the computation. The type A may or may not be given—in one case it is synthesized, in the other case checked. This refines our view as computation being described as the bottom-up construction of a derivation to include parts of the judgment in different roles (as input, output, or bidirectional components). In logic programming, which is based on the notion of computation-as-proof-search, these roles of the syntactic constituents of a judgment are called *modes*. When writing a deductive system to describe an algorithm, we have to be careful to respect the modes. We discuss this further when we come to the individual rules.

Hypotheses. The two variable rules leave no slack, since besides the hypothesis itself, no assumptions are consumed.

$$\frac{}{\Gamma; (\Delta_I, w{:}A) \setminus \Delta_I \vdash_0 w : A}\, w \qquad \frac{}{(\Gamma, u{:}A); \Delta_I \setminus \Delta_I \vdash_0 u : A}\, u$$

Multiplicative Connectives. For linear implication, we must make sure that the hypothesis introduced by $\multimap$I actually was used and is not part of the residual hypothesis Δ_O. If there is slack, we can simply erase it.

$$\frac{\Gamma; (\Delta_I, w{:}A) \setminus \Delta_O \vdash_i M : B \quad \text{where } i = 1 \text{ or } w \text{ not in } \Delta_O}{\Gamma; \Delta_I \setminus (\Delta_O - w{:}A) \vdash_i \hat{\lambda} w{:}A.\, M : A \multimap B}\, \multimap \mathrm{I}^w$$

$$\frac{\Gamma; \Delta_I \setminus \Delta' \vdash_i M : A \multimap B \qquad \Gamma; \Delta' \setminus \Delta_O \vdash_k N : A}{\Gamma; (\Delta_I \setminus \Delta_O) \vdash_{i \vee k} M\,\hat{}\,N : B}\, \multimap \mathrm{E}$$

Here $i \vee k = 1$ if $i = 1$ or $k = 1$, and $i \vee k = 0$ otherwise. This means we have slack in the result, if either of the two premisses permits slack.

$$\frac{\Gamma; \Delta_I \setminus \Delta' \vdash_i M : A \qquad \Gamma; \Delta' \setminus \Delta_O \vdash_k N : B}{\Gamma; \Delta_I \setminus \Delta_O \vdash_{i \vee k} M \otimes N : A \otimes B}\, \otimes\mathrm{I}$$

$$\frac{\begin{array}{l}\Gamma; \Delta_I \setminus \Delta' \vdash_i M : A \otimes B \\ \Gamma; (\Delta', w_1{:}A, w_2{:}B) \setminus \Delta_O \vdash_k N : C \\ \text{where } k = 1 \text{ or } w_1 \text{ and } w_2 \text{ not in } \Delta_O\end{array}}{\Gamma; \Delta_I \setminus (\Delta_O - w_1{:}A - w_2{:}B) \vdash_{i \vee k} \mathsf{let}\, w_1 \otimes w_2 = M \,\mathsf{in}\, N : C}\, \otimes\mathrm{E}^{w_1, w_2}$$

In the ⊗E rule we stack the premisses on top of each other since they are too long to fit on one line. The unit type permits no slack.

$$\frac{}{\Gamma; \Delta_I \setminus \Delta_I \vdash_0 \star : \mathbf{1}}\ \mathbf{1}\mathrm{I}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta' \vdash_i M : \mathbf{1} \qquad \Gamma; \Delta' \setminus \Delta_O \vdash_k N : C}{\Gamma; \Delta_I \setminus \Delta_O \vdash_{i \vee k} \mathsf{let}\,\star = M \,\mathsf{in}\, N : C}\ \mathbf{1}\mathrm{E}$$

Additive Connectives. The mechanism of passing and consuming resources was designed to eliminate unwanted non-determinism in the multiplicative connectives. This introduces complications in the additives, since we have to force premisses to consume exactly the same resources. We write out four version of the &I rule.

$$\frac{\Gamma; \Delta_I \setminus \Delta'_O \vdash_0 M : A \quad \Gamma; \Delta_I \setminus \Delta''_O \vdash_0 N : B \quad \Delta'_O = \Delta''_O}{\Gamma; \Delta_I \setminus (\Delta'_O \cap \Delta''_O) \vdash_0 \langle M, N \rangle : A \& B}\ \&\mathrm{I}_{00}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta'_O \vdash_0 M : A \quad \Gamma; \Delta_I \setminus \Delta''_O \vdash_1 N : B \quad \Delta'_O \subseteq \Delta''_O}{\Gamma; \Delta_I \setminus (\Delta'_O \cap \Delta''_O) \vdash_0 \langle M, N \rangle : A \& B}\ \&\mathrm{I}_{10}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta'_O \vdash_1 M : A \quad \Gamma; \Delta_I \setminus \Delta''_O \vdash_0 N : B \quad \Delta'_O \supseteq \Delta''_O}{\Gamma; \Delta_I \setminus (\Delta'_O \cap \Delta''_O) \vdash_0 \langle M, N \rangle : A \& B}\ \&\mathrm{I}_{01}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta'_O \vdash_1 M : A \quad \Gamma; \Delta_I \setminus \Delta''_O \vdash_1 N : B}{\Gamma; \Delta_I \setminus (\Delta'_O \cap \Delta''_O) \vdash_1 \langle M, N \rangle : A \& B}\ \&\mathrm{I}_{11}$$

Note that in $\&\mathrm{I}_{00}$, $\Delta'_O \cap \Delta''_O = \Delta'_O = \Delta''_O$ by the condition in the premiss. Similarly for the other rules. We chose to present the rules in a uniform way despite this redundancy to highlight the similarities. Only if both premisses permit slack do we have slack overall.

$$\frac{\Gamma; \Delta \setminus \Delta_O \vdash_i M : A \& B}{\Gamma; \Delta_I \setminus \Delta_O \vdash_i \mathsf{fst}\, M : A}\ \&\mathrm{E_L} \qquad \frac{\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : A \& B}{\Gamma; \Delta_I \setminus \Delta_O \vdash_i \mathsf{snd}\, M : B}\ \&\mathrm{E_R}$$

Finally, we come to the reason for the slack indicator.

$$\frac{}{\Gamma; \Delta_I \setminus \Delta_I \vdash_1 \langle\,\rangle : \top}\ \top\mathrm{I} \qquad \textit{No } \top \textit{ elimination}$$

The introduction rules for disjunction are direct.

$$\frac{\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : A}{\Gamma; \Delta_I \setminus \Delta_O \vdash_i \mathsf{inl}^B : A \oplus B}\ \oplus\mathrm{I_L} \qquad \frac{\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : B}{\Gamma; \Delta_I \setminus \Delta_O \vdash_i \mathsf{inr}^A : A \oplus B}\ \oplus\mathrm{I_R}$$

The elimination rule for disjunction combines resource propagation (as for multiplicatives) introduction of hypothesis, and resource coordination (as for additives) and is therefore somewhat tedious. It is left to Exercise 6.6. The **0**E rule permits slack, no matter whether the derivation of the premiss permits slack.

$$\textit{No } \mathbf{0} \textit{ introduction} \qquad \frac{\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : \mathbf{0}}{\Gamma; \Delta_I \setminus \Delta_O \vdash_1 \mathsf{abort}^C\, M : C}\,\mathbf{0}\mathrm{E}$$

Exponentials. Here we can enforce the emptiness of the linear context directly.

$$\frac{(\Gamma, u{:}A); \Delta_I \setminus \Delta_O \vdash_i M : B}{\Gamma; \Delta_I \setminus \Delta_O \vdash_i \lambda u{:}A.\, M : A \supset B}\,{\supset}\mathrm{I}^u$$

$$\frac{\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : A \supset B \qquad \Gamma; \cdot \setminus \Delta^* \vdash_k N : A}{\Gamma; \Delta_I \setminus \Delta_O \vdash_i M\, N : B}\,{\supset}\mathrm{E}$$

Here Δ^* will always have to be $\cdot$ (since it must be a subset of $\cdot$) and k is irrelevant. The same is true in the next rule.

$$\frac{\Gamma; \cdot \setminus \Delta^* \vdash_i M : A}{\Gamma; \Delta_I \setminus \Delta_I \vdash_0 {!M} : {!A}}\,!\mathrm{I}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta' \vdash_i M : {!A} \qquad (\Gamma, u{:}A); \Delta' \setminus \Delta_O \vdash_k N : C}{\Gamma; \Delta_I \setminus \Delta_O \vdash_{i \vee j} \mathsf{let}\, {!u} = M\, \mathsf{in}\, N : C}\,!\mathrm{E}^u$$

The desired soundness and completeness theorem for the algorithmic typing judgment must first be generalized before it can be proved by induction. For this generalization, the mode (input and output) of the constituents of the judgment is a useful guide. For example, in the completness direction (3), we can expect to distinguish cases based on the slack indicator which might be returned when we ask the question if there are Δ_O and i such that $\Gamma; \Delta \setminus \Delta_O \vdash_i M : A$ for the given Γ, Δ, M and A.

Lemma 6.4 (Properties of Algorithmic Type Checking)

1. *If* $\Gamma; \Delta_I \setminus \Delta_O \vdash_0 M : A$ *then* $\Delta_I \supseteq \Delta_O$ *and* $\Gamma; \Delta_I - \Delta_O \vdash M : A$.
2. *If* $\Gamma; \Delta_I \setminus \Delta_O \vdash_1 M : A$ *then* $\Delta_I \supseteq \Delta_O$ *and for any* Δ *such that* $\Delta_I \supseteq \Delta \supseteq \Delta_I - \Delta_O$ *we have* $\Gamma; \Delta \vdash M : A$.
3. *If* $\Gamma; \Delta \vdash M : A$ *then either*
   (a) $\Gamma; (\Delta', \Delta) \setminus \Delta' \vdash_0 M : A$ *for any* Δ', *or*
   (b) $\Gamma; (\Delta', \Delta) \setminus (\Delta', \Delta_O) \vdash_1 M : A$ *for all* Δ' *and some* $\Delta_O \subseteq \Delta$.

Proof: By inductions on the structure of the given derivations.[1] Items (1) and (2) must be proven simultaneously. □

From this lemma, the soundness and completeness of algorithmic type checking follow directly.

Theorem 6.5 (Algorithmic Type Checking)
$\Gamma; \Delta \vdash M : A$ *if and only if either*

1. $\Gamma; \Delta \setminus \cdot \vdash_0 M : A$, *or*
2. $\Gamma; \Delta \setminus \Delta' \vdash_1 M : A$ *for some* Δ'.

Proof: Directly from Lemma 6.4 □

6.3 Pure Linear Functional Programming

The linear λ-calculus developed in the preceding sections can serve as the basis for a programming language. The step from λ-calculus to programming language can be rather complex, depending on how realistic one wants to make the resulting language. The first step is to decide on *observable types* and a language of *values* and then define an *evaluation judgment*. This is the subject of this section. Given the purely logical view we have taken, this language still lacks datatypes and recursion. In order to remedy this, we introduce *recursive types* and *recursive terms* in the next section.

Our operational semantics follows the intuition that we should not evaluate expressions whose value may not be needed for the result. Expressions whose value will definitely be used, can be evaluated eagerly. There is a slight mismatch in that the linear λ-calculus can identifies expressions whose value will be needed *exactly once*. However, we can derive other potential benefits from the stronger restriction at the lower levels of an implementation such as improved garbage collection or update-in-place. These benefits also have their price, and at this time the trade-offs are not clear. For the *strict λ-calculus* which captures the idea of definite use of the value of an expression, see Exercise 6.2.

We organize the functional language strictly along the types, discussing observability, values, and evaluation rules for each. We have two main judgments, M *Value* (M is a value), and $M \hookrightarrow v$ (M evaluates to v). In general we use v for terms which are legal values. For both of these we assume that M is *closed* and *well-typed*, that is, $\cdot; \cdot \vdash M : A$.

Linear Implication. An important difference between a general λ-calculus and a functional language is that the structure of functions in a programming language is *not* observable. Instead, functions are compiled to code. Their behavior can be observed by applying functions to arguments, but their definition cannot be seen. Thus, strictly speaking, it is incorrect to say that functions

[1] [*check*]

are first-class. This holds equally for so-called lazy functional languages such as Haskell and eager functional languages such as ML. Thus, any expression of the form $\hat{\lambda} w{:}A.\ M$ is a possible value.

$$\frac{}{\hat{\lambda} w{:}A.\ M\ \mathit{Value}} \multimap \mathrm{val}$$

Evaluation of a $\hat{\lambda}$-abstraction returns itself immediately.

$$\frac{}{\hat{\lambda} w{:}A.\ M \hookrightarrow \hat{\lambda} w{:}A.\ M} \multimap \mathrm{Iv}$$

Since a linear parameter to a function is definitely used (in fact, used exactly once), we can evaluate the argument without doing unnecessary work and substitute it for the bound variable during the evaluation of an application.

$$\frac{M_1 \hookrightarrow \hat{\lambda} w{:}A_2.\ M_1' \qquad M_2 \hookrightarrow v_2 \qquad [v_2/w]M_1' \hookrightarrow v}{M_1 \,\hat{}\, M_2 \hookrightarrow v} \multimap \mathrm{Ev}$$

Note that after we substitute the value of argument v_2 for the formal parameter w in the function, we have to evaluate the body of the function.

Simultaneous Pairs. The multiplicative conjunction $A \otimes B$ corresponds to the type of pairs where both elements must be used exactly once. Thus we can evaluate the components (they will be used!) and the pairs are observable. The elimination form is evaluated by creating the pair and then deconstructing it.

$$\frac{M_1\ \mathit{Value} \qquad M_2\ \mathit{Value}}{M_1 \otimes M_2\ \mathit{Value}} \otimes\mathrm{val}$$

$$\frac{M_1 \hookrightarrow v_1 \qquad M_2 \hookrightarrow v_2}{M_1 \otimes M_2 \hookrightarrow v_1 \otimes v_2} \otimes\mathrm{Iv} \qquad \frac{M \hookrightarrow v_1 \otimes v_2 \qquad [v_1/w_1, v_2/w_2]N \hookrightarrow v}{\mathsf{let}\, w_1 \otimes w_2 = M \,\mathsf{in}\, N \hookrightarrow v} \otimes\mathrm{Ev}$$

Multiplicative Unit. The multiplicative unit $\mathbf{1}$ is observable and contains exactly one value $\star$. Its elimination rule explicitly evaluates a term and ignores its result (which must be $\star$).

$$\frac{}{\star\ \mathit{Value}} \mathbf{1}\mathrm{val}$$

$$\frac{}{\star \hookrightarrow \star} \mathbf{1}\mathrm{Iv} \qquad \frac{M \hookrightarrow \star \qquad N \hookrightarrow v}{\mathsf{let}\, \star = M \,\mathsf{in}\, N \hookrightarrow v} \mathbf{1}\mathrm{Ev}$$

Alternative Pairs. Alternative pairs of type $A \& B$ are such that we can only use one of the two components. Since we may not be able to predict which one, we should not evaluate the components. Thus pairs $\langle M_1, M_2 \rangle$ are *lazy*, not observable and any pair of this form is a value. When we extract a component, we then have to evaluate the corresponding term to obtain a value.

$$\frac{}{\langle M_1, M_2 \rangle \ \mathit{Value}}\ \&\mathrm{val}$$

$$\frac{}{\langle M_1, M_2 \rangle \hookrightarrow \langle M_1, M_2 \rangle}\ \&\mathrm{Iv}$$

$$\frac{M \hookrightarrow \langle M_1, M_2 \rangle \qquad M_1 \hookrightarrow v_1}{\mathsf{fst}\, M \hookrightarrow v_1}\ \&\mathrm{Ev}_1 \qquad \frac{M \hookrightarrow \langle M_1, M_2 \rangle \qquad M_2 \hookrightarrow v_2}{\mathsf{snd}\, M \hookrightarrow v_2}\ \&\mathrm{Ev}_2$$

Additive Unit. By analogy, the additive unit $\top$ is not observable. Since there is no elimination rule, we can never do anything interesting with a value of this type, except embed it in larger values.

$$\frac{}{\langle\, \rangle \ \mathit{Value}}\ \top\mathrm{val}$$

$$\frac{}{\langle\, \rangle \hookrightarrow \langle\, \rangle}\ \top\mathrm{Iv}$$

This rule does not express the full operational intuition behind $\top$ which "garbage collects" all linear resources. However, we can only fully appreciate this when we define evaluation under environments (see Section **??**).

Disjoint Sum. The values of a disjoint sum type are guaranteed to be used (no matter whether it is of the form $\mathsf{inl}^B\, M$ or $\mathsf{inr}^A\, M$). Thus we can require values to be built up from injections of values, and the structure of sum values is observable. There are two rules for evaluation, depending on whether the subject of a case-expression is a left injection or right injection into the sum type.

$$\frac{M \ \mathit{Value}}{\mathsf{inl}^B\, M \ \mathit{Value}}\ \oplus\mathrm{val}_1 \qquad \frac{M \ \mathit{Value}}{\mathsf{inr}^A\, M \ \mathit{Value}}\ \oplus\mathrm{val}_2$$

$$\frac{M \hookrightarrow v}{\mathsf{inl}^B\, M \hookrightarrow \mathsf{inl}^B\, v}\ \oplus\mathrm{Iv}_1 \qquad \frac{M \hookrightarrow v}{\mathsf{inr}^A\, M \hookrightarrow \mathsf{inr}^A\, v}\ \oplus\mathrm{Iv}_2$$

$$\frac{M \hookrightarrow \mathsf{inl}^B\, v_1 \qquad [v_1/w_1]N_1 \hookrightarrow v}{\mathsf{case}\, M\, \mathsf{of}\, \mathsf{inl}\, w_1 \Rightarrow N_1 \mid \mathsf{inr}\, w_2 \Rightarrow N_2 \hookrightarrow v}\ \oplus\mathrm{Ev}_1$$

$$\frac{M \hookrightarrow \mathsf{inr}^A\, v_2 \qquad [v_2/w_2]N_2 \hookrightarrow v}{\mathsf{case}\, M\, \mathsf{of}\, \mathsf{inl}\, w_1 \Rightarrow N_1 \mid \mathsf{inr}\, w_2 \Rightarrow N_2 \hookrightarrow v}\ \oplus\mathrm{Ev}_2$$

Void Type. The void type $\mathbf{0}$ contains no value. In analogy with the disjoint sum type it is observable, although this is not helpful in practice. There are no evaluation rules for this type: since there are no introduction rules there are no constructor rules, and the elimination rule distinguishes between zero possible cases (in other words, is impossible). We called this $\mathsf{abort}^A\, M$, since it may be viewed as a global program abort.

Unrestricted Function Type. The unrestricted function type $A \supset B$ (also written as $A \to B$ in accordance with the usual practice in functional programming) may or may not use its argument. Therefore, the argument is not evaluated, but simply substituted for the bound variable. This is referred to as a *call-by-name* semantics. It is usually implemented by *lazy evaluation*, which means that first time the argument is evaluated, this value is memoized to avoid re-evaluation. This is not represented at this level of semantic description. Values of functional type are not observable, as in the linear case.

$$\frac{}{\lambda u{:}A.\ M\ \ \mathit{Value}} \to \mathrm{Ival}$$

$$\frac{}{\lambda u{:}A.\ M \hookrightarrow \lambda u{:}A.\ M} \to \mathrm{Iv}$$

$$\frac{M_1 \hookrightarrow \lambda u{:}A_2.\ M_1' \qquad [M_2/u]M_1' \hookrightarrow v}{M_1\ M_2 \hookrightarrow v} \to \mathrm{Ev}$$

Modal Type. A linear variable of type $!A$ must be used, but the embedded expression of type A may *not* be used since it is unrestricted. Therefore, terms $!M$ are values and "!" is like a quotation of its argument M, protecting it from evaluation.

$$\frac{}{!M\ \ \mathit{Value}}\ !\mathrm{val}$$

$$\frac{}{!M \hookrightarrow\ !M}\ !\mathrm{Iv} \qquad \frac{M \hookrightarrow\ !M' \qquad [M'/u]N \hookrightarrow v}{\mathsf{let}\ !u = M\ \mathsf{in}\ N \hookrightarrow v}\ !\mathrm{Ev}$$

We abbreviate the value judgment from above in the form of a grammar.

Values	v	$::=$	$\hat{\lambda} w{:}A.\ M$	$A \multimap B$	not observable
		$\mid$	$v_1 \otimes v_2$	$A_1 \otimes A_2$	observable
		$\mid$	$\star$	$\mathbf{1}$	observable
		$\mid$	$\langle M_1, M_2 \rangle$	$A_1 \& A_2$	not observable
		$\mid$	$\langle\,\rangle$	$\top$	not observable
		$\mid$	$\mathsf{inl}^B\, v \mid \mathsf{inr}^A\, v$	$A \oplus B$	observable
			No values	$\mathbf{0}$	observable
		$\mid$	$\lambda u{:}A.\ M$	$A \to B$	not observable
		$\mid$	$!M$	$!A$	not observable

In the absence of datatypes, we cannot write many interesting programs. As a first example we consider the representation of the Booleans with two values, true and false, and a conditional as an elimination construct.

$$
\begin{array}{rcl}
\mathrm{bool} & = & \mathbf{1} \oplus \mathbf{1} \\
\mathrm{true} & = & \mathsf{inl}^{\mathbf{1}}\, \star \\
\mathrm{false} & = & \mathsf{inr}^{\mathbf{1}}\, \star \\
\textbf{if}\, M\, \textbf{then}\, N_1\, \textbf{else}\, N_2 & = & \mathsf{case}\; M \\
& & \quad \mathsf{of}\ \mathsf{inl}^{\mathbf{1}}\, w_1 \Rightarrow \mathsf{let}\, \star = w_1\, \mathsf{in}\, N_1 \\
& & \quad \mid\ \mathsf{inr}^{\mathbf{1}}\, w_2 \Rightarrow \mathsf{let}\, \star = w_2\, \mathsf{in}\, N_2
\end{array}
$$

The elimination of $\star$ in the definition of the conditional is necessary, because a branch $\mathsf{inl}^{\mathbf{1}}\, w_1 \Rightarrow N_1$ would not be well-typed: w_1 is a linear variable not used in its scope. Destructuring a value in several stages is a common idiom and it is helpful for the examples to introduce some syntactic sugar. We allow patterns which nest the elimination forms which appear in a let or case. Not all combination of these are legal, but it is not difficult to describe the legal pattern and match expressions (see Exercise 6.7).

$$
\begin{array}{llcl}
\textit{Patterns} & p & ::= & w \mid p_1 \otimes p_2 \mid \star \mid \mathsf{inl}\, p \mid \mathsf{inr}\, p \mid u \mid {!p} \\
\textit{Matches} & m & ::= & p \Rightarrow M \mid (m_1 \mid m_2) \mid \cdot
\end{array}
$$

An *extended case expression* has the form $\mathsf{case}\, M\, \mathsf{of}\, m$.

In the example of Booleans above, we gave a uniform definition for conditionals in terms of case. But can we define a function cond with arguments M, N_1 and N_2 which behaves like $\textbf{if}\, M\, \textbf{then}\, N_1\, \textbf{else}\, N_2$? The first difficulty is that the type of branches is generic. In order to avoid the complications of polymorphism, we uniformly define a whole family of functions cond_C types C. We go through some candidate types for cond_C and discuss why they may or may not be possible.

$\mathrm{cond}_C : \mathbf{1} \oplus \mathbf{1} \multimap C \multimap C \multimap C$**.** This type means that both branches of the conditional (second and third argument) would be evaluated before being substituted in the definition of cond_C. Moreover, both must be used during the evaluation of the body, while intuitively only one branch should be used.

$\mathrm{cond}_C : \mathbf{1} \oplus \mathbf{1} \multimap (!C) \multimap (!C) \multimap C$**.** This avoids evaluation of the branches, since they now can have the form $!N_1$ and $!N_2$, which are values. However, N_1 and N_2 can now no longer use linear variables.

$\mathrm{cond}_C : \mathbf{1} \oplus \mathbf{1} \multimap C \rightarrow C \rightarrow C$**.** This is equivalent to the previous type and undesirable for the same reason.

$\mathrm{cond}_C : \mathbf{1} \oplus \mathbf{1} \multimap (C \& C) \multimap C$**.** This type expresses that the second argument of type $C \& C$ is a pair $\langle N_1, N_2 \rangle$ such that exactly one component of this pair will be used. This expresses precisely the expected behavior and we define

$$
\begin{array}{lcl}
\mathrm{cond}_C & : & \mathbf{1} \oplus \mathbf{1} \multimap (C \& C) \multimap C \\
 & = & \hat{\lambda} b{:}\mathbf{1} \oplus \mathbf{1}.\ \hat{\lambda} n{:}C \& C. \\
 & & \quad \mathsf{case}\ b \\
 & & \qquad \mathsf{of}\ \mathsf{inl}\,\star \Rightarrow \mathsf{fst}\,n \\
 & & \qquad\ \mid\ \mathsf{inr}\,\star \Rightarrow \mathsf{snd}\,n
\end{array}
$$

which is linearly well-typed: b is used as the subject of the case and n is used in both branches of the case expression (which is additive).

As a first property of evaluation, we show that it is a strategy for β-reductions. That is, if $M \hookrightarrow v$ then M reduces to v in some number of β-reduction steps (possibly none), but not *vice versa*. For this we need a new judgment $M \longrightarrow_\beta^* M'$ is the congruent, reflexive, and transitive closure of the $M \longrightarrow_\beta M'$ relation. In other words, we extend β-reduction so it can be applied to an arbitrary subterm of M and then allow arbitrary sequences of reductions. The subject reduction property holds for this judgment as well.

Theorem 6.6 (Generalized Subject Reduction) *If* $\Gamma; \Delta \vdash M : A$ *and* $M \longrightarrow_\beta^* M'$ *then* $\Gamma; \Delta \vdash M' : A$.

Proof: See Exercise 6.8 □

Evaluation is related to β-reduction in that an expression reduces to its value.

Theorem 6.7 *If* $M \hookrightarrow v$ *then* $M \longrightarrow_\beta^* v$.

Proof: By induction on the structure of the derivation of $M \hookrightarrow v$. In each case we directly combine results obtained by appealing to the induction hypothesis using transitivity and congruence. □

The opposite is clearly false. For example,

$$\langle (\hat{\lambda} w{:}\mathbf{1}.\ w)\,\hat{}\,\star, \star \rangle \longrightarrow_\beta^* \langle \star, \star \rangle,$$

but

$$\langle (\hat{\lambda} w{:}\mathbf{1}.\ w)\,\hat{}\,\star, \star \rangle \hookrightarrow \langle (\hat{\lambda} w{:}\mathbf{1}.\ w)\,\hat{}\,\star, \star \rangle$$

and this is the only evaluation for the pair. However, if we limit the congruence rules to the components of $\otimes$, inl, inr, and all elimination constructs, the correspondence is exact (see Exercise 6.9). Type preservation is a simple consequence of the previous two theorems. See Exercise 6.10 for a direct proof.

Theorem 6.8 (Type Preservation) *If* $\cdot; \cdot \vdash M : A$ *and* $M \hookrightarrow v$ *then* $\cdot; \cdot \vdash v : A$.

Proof: By Theorem 6.7, $M \longrightarrow_\beta^* v$. Then the result follows by generalized subject reduction (Theorem 6.6). □

The final theorem of this section establishes the uniqueness of values.

Theorem 6.9 (Determinacy) *If $M \hookrightarrow v$ and $M \hookrightarrow v'$ then $v = v'$.*

Proof: By straightforward simultaneous induction on the structure of the two given derivations. For each for of M except case expressions there is exactly one inference rule which could be applied. For `case` we use the uniqueness of the value of the case subject to determine that the same rule must have been used in both derivations. □

We can also prove that evaluation of any closed, well-typed term M terminates in this fragment. We postpone the proof of this (Theorem **??**) until we have seen further, more realistic, examples.

6.4 Recursive Types

The language so far lacks basic data types, such as natural numbers, integers, lists, trees, etc. Moreover, except for finitary ones such as booleans, they are not definable with the mechanism at our disposal so far. At this point we can follow two paths: one is to define each new data type in the same way we defined the logical connectives, that is, by introduction and elimination rules, carefully checking their local soundness and completeness. The other is to enrich the language with a general mechanism for defining such new types. Again, this can be done in different ways, using either *inductive types* which allow us to maintain a clean connection between propositions and types, or *recursive types* which are more general, but break the correspondence to logic. Since we are mostly interested in programming here, we chose the latter path.

Recall that we defined the booleans as $\mathbf{1} \oplus \mathbf{1}$. It is easy to show by the definition of values, that there are exactly two values of this type, to which we can arbitrarily assign true and false. A finite type with n values can be defined as the disjoint sum of n observable singleton types, $\mathbf{1} \oplus \cdots \oplus \mathbf{1}$. The natural numbers would be $\mathbf{1} \oplus \mathbf{1} \oplus \cdots$, except that this type is infinite. We can express it finitely as a recursive type $\mu\alpha.\ \mathbf{1} \oplus \alpha$. Intuitively, the meaning of this type should be invariant under unrolling of the recursion. That is,

$$
\begin{array}{rcl}
\mathrm{nat} & = & \mu\alpha.\, \mathbf{1} \oplus \alpha \\
 & \cong & [(\mu\alpha.\, \mathbf{1} \oplus \alpha)/\alpha]\mathbf{1} \oplus \alpha \\
 & = & \mathbf{1} \oplus \mu\alpha.\, \mathbf{1} \oplus \alpha \\
 & = & \mathbf{1} \oplus \mathrm{nat}
\end{array}
$$

which is the expected recursive definition for the type of natural numbers.

In functional languages such as ML or Haskell, recursive type definitions are not directly available, but the results of elaborating syntactically more pleaseant definitions. In addition, recursive type definitions are *generative*, that is, they generate new constructors and types every time they are invoked. This is of great practical value, but the underlying type theory can be seen as simple recursive types combined with a mechanism for generativity. Here, we will only treat the issue of recursive types.

Even though recursive types do not admit a logical interpretation as propositions, we can still define a term calculus using introduction and elimination rules, including local reduction and expansions. In order maintain the property that a term has a unique type, we annotate the introduction constant fold with the recursive type itself.

$$\frac{\Gamma; \Delta \vdash M : [\mu\alpha.\ A/\alpha]A}{\Gamma; \Delta \vdash \mathrm{fold}^{\mu\alpha.\ A} M : \mu\alpha.\ A}\,\mu\mathrm{I} \qquad \frac{\Gamma; \Delta \vdash M : \mu\alpha.\ A}{\Gamma; \Delta \vdash \mathrm{unfold}\, M : [\mu\alpha.\ A/\alpha]A}\,\mu\mathrm{E}$$

The local reduction and expansions, expressed on the terms.

$$\begin{array}{rcl} \mathrm{unfold}\,\mathrm{fold}^{\mu\alpha.\ A}\, M & \longrightarrow_\beta & M \\ M : \mu\alpha.\ A & \longrightarrow_\eta & \mathrm{fold}^{\mu\alpha.\ A}\,(\mathrm{unfold}\, M) \end{array}$$

It is easy to see that uniquess of types and subject reduction remain valid properties (see Exercise 6.11). There are also formulation of recursive types where the term M in the premiss and conclusion is the same, that is, there are no explicit constructor and destructors for recursive types. This leads to more concise programs, but significantly more complicated type-checking (see Exercise 6.12).

We would like recursive types to represent data types. Therefore the values of recursive type must be of the form $\mathrm{fold}^{\mu\alpha.\ A}\, v$ for values v—otherwise data values would not be observable.

$$\frac{M\ \mathit{Value}}{\mathrm{fold}^{\mu\alpha.\ A}\, M\ \mathit{Value}}\,\mu\mathrm{val}$$

$$\frac{M \hookrightarrow v}{\mathrm{fold}^{\mu\alpha.\ A}\, M \hookrightarrow \mathrm{fold}^{\mu\alpha.\ A}\, v}\,\mu\mathrm{Iv} \qquad \frac{M \hookrightarrow \mathrm{fold}^{\mu\alpha.\ A}\, v}{\mathrm{unfold}\, M \hookrightarrow v}\,\mu\mathrm{Ev}$$

In order to write interesting programs simply, it is useful to have a general recursion operator $\mathbf{fix}\, u{:}A.\ M$ at the level of terms. It is not associated with an type constructor and simply unrolls its definition once when executed. In the typing rule we have to be careful: since the number on unrollings generally unpredictable, no linear variables are permitted to occur free in the body of a recursive definition. Moreover, the recursive function itself may be called arbitrarily many times—one of the characteristics of recursion. Therefore its uses are unrestricted.

$$\frac{(\Gamma, u{:}A); \cdot \vdash M : A}{\Gamma; \cdot \vdash \mathbf{fix}\, u{:}A.\ M : A}\,\mathbf{fix}$$

The operator does not introduce any new values, and one new evaluation rules which unrolls the recursion.

$$\frac{[\mathbf{fix}\, u{:}A.\ M/u]M \hookrightarrow v}{\mathbf{fix}\, u{:}A.\ M \hookrightarrow v}\,\mathbf{fix}\mathrm{v}$$

In order to guarantee subject reduction, the type of whole expression, the body M of the fixpoint expression, and the bound variable u must all have the same type A. This is enforced in the typing rules.

We now consider a few examples of recursive types and some example programs.

Natural Numbers.

$$
\begin{array}{rcl}
\mathrm{nat} & = & \mu\alpha.\, \mathbf{1} \oplus \alpha \\
\mathrm{zero} & : & \mathrm{nat} \\
 & = & \mathrm{fold}^{\mathrm{nat}}\, (\mathsf{inl}^{\mathrm{nat}}\, \star) \\
\mathrm{succ} & : & \mathrm{nat} \multimap \mathrm{nat} \\
 & = & \hat{\lambda} x{:}\mathrm{nat}.\, \mathrm{fold}^{\mathrm{nat}}\, (\mathsf{inr}^{\mathbf{1}}\, x)
\end{array}
$$

With this definition, the addition function for natural numbers is linear in both argument.

$$
\begin{array}{rcl}
\mathrm{plus} & : & \mathrm{nat} \multimap \mathrm{nat} \multimap \mathrm{nat} \\
 & = & \mathbf{fix}\, p{:}\mathrm{nat} \multimap \mathrm{nat} \multimap \mathrm{nat}. \\
 & & \quad \hat{\lambda} x{:}\mathrm{nat}.\, \hat{\lambda} y{:}\mathrm{nat}.\, \mathsf{case}\ \mathrm{unfold}\, x \\
 & & \qquad\qquad\qquad\qquad \mathsf{of}\ \mathsf{inl}\, \star \Rightarrow y \\
 & & \qquad\qquad\qquad\qquad \mid \mathsf{inr}\, x' \Rightarrow \mathrm{succ}\,\hat{}\,(p\,\hat{}\,x'\,\hat{}\,y)
\end{array}
$$

It is easy to ascertain that this definition is well-typed: x occurs as the case subject, y in both branches, and x' in the recursive call to p. On the other hand, the natural definition for multiplication is *not* linear, since the second argument is used twice in one branch of the case statement and not at all in the other.

$$
\begin{array}{rcl}
\mathrm{mult} & : & \mathrm{nat} \multimap \mathrm{nat} \to \mathrm{nat} \\
 & = & \mathbf{fix}\, m{:}\mathrm{nat} \multimap \mathrm{nat} \to \mathrm{nat} \\
 & & \quad \hat{\lambda} x{:}\mathrm{nat}.\, \lambda y{:}\mathrm{nat}.\, \mathsf{case}\ \mathrm{unfold}\, x \\
 & & \qquad\qquad\qquad\qquad \mathsf{of}\ \mathsf{inl}\, \star \Rightarrow \mathrm{zero} \\
 & & \qquad\qquad\qquad\qquad \mid \mathsf{inr}\, x' \Rightarrow \mathrm{plus}\,\hat{}\,(m\,\hat{}\,x'\, y)\,\hat{}\,y
\end{array}
$$

Interestingly, there is also a linear definition of mult (see Exercise 6.13), but its operational behavior is quite different. This is because we can explicitly copy and delete natural numbers, and even make them available in an unrestricted way.

$$
\begin{array}{rcl}
\mathrm{copy} & : & \mathrm{nat} \multimap \mathrm{nat} \otimes \mathrm{nat} \\
 & = & \mathbf{fix}\, c{:}\mathrm{nat} \multimap \mathrm{nat} \otimes \mathrm{nat} \\
 & & \quad \hat{\lambda} x{:}\mathrm{nat}.\ \mathsf{case}\ \mathrm{unfold}\, x \\
 & & \qquad\qquad \mathsf{of}\ \mathsf{inl}\, \star \Rightarrow \mathrm{zero} \otimes \mathrm{zero} \\
 & & \qquad\qquad \mid \mathsf{inr}\, x' \Rightarrow \mathsf{let}\, x'_1 \otimes x'_2 = c \mathbin{\hat{}} x'\ \mathsf{in}\ (\mathrm{succ} \mathbin{\hat{}} x'_1) \otimes (\mathrm{succ} \mathbin{\hat{}} x'_2) \\
\mathrm{delete} & : & \mathrm{nat} \multimap \mathbf{1} \\
 & = & \mathbf{fix}\, d{:}\mathrm{nat} \multimap \mathbf{1} \\
 & & \quad \hat{\lambda} x{:}\mathrm{nat}.\ \mathsf{case}\ \mathrm{unfold}\, x \\
 & & \qquad\qquad \mathsf{of}\ \mathsf{inl}\, \star \Rightarrow \mathbf{1} \\
 & & \qquad\qquad \mid \mathsf{inr}\, x' \Rightarrow \mathsf{let}\, \star = d \mathbin{\hat{}} x'\ \mathsf{in}\ \mathbf{1} \\
\mathrm{promote} & : & \mathrm{nat} \multimap\, !\mathrm{nat} \\
 & = & \mathbf{fix}\, p{:}\mathrm{nat} \multimap\, !\mathrm{nat} \\
 & & \quad \hat{\lambda} x{:}\mathrm{nat}.\ \mathsf{case}\ \mathrm{unfold}\, x \\
 & & \qquad\qquad \mathsf{of}\ \mathsf{inl}\, \star \Rightarrow\, !\mathrm{zero} \\
 & & \qquad\qquad \mid \mathsf{inr}\, x' \Rightarrow \mathsf{let}\, !u' = p \mathbin{\hat{}} x'\ \mathsf{in}\ !(\mathrm{succ}\, u')
\end{array}
$$

Lazy Natural Numbers. Lazy natural numbers are a simple example of *lazy data types* which contain unevaluated expressions. Lazy data types are useful in applications with potentially infinite data such as streams. We encode such lazy data types by using the $!A$ type constructor.

$$
\begin{array}{rcl}
\mathrm{lnat} & = & \mu\alpha.\, !(\mathbf{1} \oplus \alpha) \\
\mathrm{lzero} & : & \mathrm{lnat} \\
 & = & \mathrm{fold}^{\mathrm{lnat}}\, !\, (\mathsf{inl}^{\mathrm{lnat}}\, \star) \\
\mathrm{lsucc} & : & \mathrm{lnat} \to \mathrm{lnat} \\
 & = & \lambda u{:}\mathrm{lnat}.\ \mathrm{fold}^{\mathrm{lnat}}\, !\, (\mathsf{inr}^{\mathbf{1}}\, u)
\end{array}
$$

There is also a linear version of successor of type, $\mathrm{lnat} \multimap \mathrm{lnat}$, but it is not as natural since it evaluates its argument just to build another lazy natural number.

$$
\begin{array}{rcl}
\mathrm{lsucc}' & : & \mathrm{lnat} \multimap \mathrm{lnat} \\
 & = & \hat{\lambda} x{:}\mathrm{lnat}.\ \mathsf{let}\, !u = \mathrm{unfold}\, x\ \mathsf{in}\ \mathrm{fold}^{\mathrm{lnat}}\, (!\, \mathsf{inr}^{\mathbf{1}}\, (\mathrm{fold}^{\mathrm{lnat}}\, (!u)))
\end{array}
$$

The "infinite" number number ω can be defined by using the fixpoint operator. We can either use lsucc as defined above, or define it directly.

$$
\begin{array}{rcl}
\omega & : & \mathrm{lnat} \\
 & = & \mathbf{fix}\, u{:}\mathrm{lnat}.\ \mathrm{lsucc}\, u \\
 & \cong & \mathbf{fix}\, u{:}\mathrm{lnat}.\ \mathrm{fold}^{\mathrm{lnat}}\, !\, (\mathsf{inr}^{\mathbf{1}}\, u)
\end{array}
$$

Note that lazy natural numbers are not directly observable (except for the $\mathrm{fold}^{\mathrm{lnat}}$), so we have to decompose and examine the structure of a lazy natural number successor by successor, or we can convert it to an observable natural number (which might not terminate).

$$
\begin{array}{lcl}
\text{toNat} & : & \text{lnat} \multimap \text{nat} \\
 & = & \mathbf{fix}\, t{:}\text{lnat} \multimap \text{nat} \\
 & & \quad \hat{\lambda} x{:}\text{lnat}.\ \mathsf{case}\ \text{unfold}\, x \\
 & & \qquad\qquad \mathsf{of}\ !\mathsf{inl}^{\text{lnat}}\, \star \Rightarrow \text{zero} \\
 & & \qquad\qquad \mid\ !\mathsf{inr}^{\mathbf{1}}\, x' \Rightarrow \text{succ}\,\hat{}\,(t\,\hat{}\,x')
\end{array}
$$

Lists. To avoid issues of polymorphism, we define a family of data types list_A for an arbitrary type A.

$$
\begin{array}{rcl}
\text{list}_A & = & \mu\alpha.\ \mathbf{1} \oplus (A \otimes \alpha) \\
\text{nil}_A & : & \text{list}_A \\
 & = & \text{fold}^{\text{list}_A}\, (\mathsf{inl}^{\text{list}_A}\, \star) \\
\text{cons}_A & : & A \otimes \text{list}_A \multimap \text{list}_A \\
 & = & \hat{\lambda} p{:}A \otimes \text{list}_A.\ \text{fold}^{\text{list}_A}\, (\mathsf{inr}^{\mathbf{1}}\, p)
\end{array}
$$

We can easily program simple functions such as append and reverse which are linear in their arguments. We show here reverse; for other examples see Exercise 6.14. we define an auxiliary tail-recursive function rev which moves element from it first argument to its second.

$$
\begin{array}{rcl}
\text{rev}_A & : & \text{list}_A \multimap \text{list}_A \multimap \text{list}_A \\
 & = & \mathbf{fix}\, r{:}\text{list}_A \multimap \text{list}_A \multimap \text{list}_A \\
 & & \quad \hat{\lambda} l{:}\text{list}_A.\ \hat{\lambda} k{:}\text{list}_A. \\
 & & \qquad \mathsf{case}\ \text{unfold}\, l \\
 & & \qquad\quad \mathsf{of}\ \mathsf{inl}^{A \otimes \text{list}_A}\, \star \Rightarrow k \\
 & & \qquad\quad \mid\ \mathsf{inr}^{\mathbf{1}}\, (x \otimes l') \Rightarrow r\,\hat{}\,l'\,\hat{}\,(\text{cons}_A\, (x \otimes k)) \\
\text{reverse}_A & : & \text{list}_A \multimap \text{list}_A \\
 & = & \hat{\lambda} l{:}\text{list}_A.\ \text{rev}\,\hat{}\,l\,\hat{}\,\text{nil}_A
\end{array}
$$

To make definitions like this a bit easier, we can also define a case for lists, in analogy with the conditional for booleans. It is a family indexed by the type of list elements A and the type of the result of the conditional C.

$$
\begin{array}{rcl}
\text{listCase}_{A,C} & : & \text{list}_A \multimap (C \& (A \otimes \text{list}_A \multimap C)) \multimap C \\
 & = & \hat{\lambda} l{:}\text{list}_\text{A}.\ \hat{\lambda} n{:}C \& (A \otimes \text{list}_A \multimap C). \\
 & & \qquad \mathsf{case}\ \text{unfold} l \\
 & & \qquad\quad \mathsf{of}\ \mathsf{inl}^{A \otimes \text{list}_A}\, \star \Rightarrow \mathsf{fst}\, n \\
 & & \qquad\quad \mid\ \mathsf{inr}^{\mathbf{1}}\, p \Rightarrow (\mathsf{snd}\, n)\,\hat{}\,p
\end{array}
$$

Lazy Lists. There are various forms of lazy lists, depending of which evaluation is postponed.

$\text{llist}^1_A = \mu\alpha.\ !(\mathbf{1} \oplus (A \otimes \alpha))$. This is perhaps the canonical lazy lists, in which we can observe neither head nor tail.

$\mathrm{llist}^2_A = \mu\alpha.\ \mathbf{1} \oplus\, !(A \otimes \alpha)$. Here we can observe directly if the list is empty or not, but not the head or tail which remains unevaluated.

$\mathrm{llist}^3_A = \mu\alpha.\ \mathbf{1} \oplus (A \otimes\, !\alpha)$. Here we can observe directly if the list is empty or not, and the head of the list is non-empty. However, we cannot see the tail.

$\mathrm{llist}^4_A = \mu\alpha.\ \mathbf{1} \oplus (!A \otimes \alpha)$. Here the list is always eager, but the elements are lazy. This is the same as $\mathrm{list}_{!A}$.

$\mathrm{llist}^5_A = \mu\alpha.\ \mathbf{1} \oplus (A \& \alpha)$. Here we can see if the list is empty or not, but we can access only either the head or tail of list, but not both.

$\mathrm{infStream}_A = \mu\alpha.\ !(A \otimes \alpha)$. This is the type of infinite streams, that is, lazy lists with no nil constructor.

Functions such as append, map, etc. can also be written for lazy lists (see Exercise 6.15).

Other types, such as trees of various kinds, are also easily represented using similar ideas. However, the recursive types (even without the presence of the fixpoint operator on terms) introduce terms which have no normal form. In the pure, untyped λ-calculus, the classical examples of a term with no normal form is $(\lambda x.\ x\, x)\, (\lambda x.\ x\, x)$ which β-reduces to itself in one step. In the our typed λ-calculus (linear or intuitionistic) this cannot be assigned a type, because x is used as an argument to itself. However, with recursive types (and the fold and unfold constructors) we can give a type to a version of this term which β-reduces to itself in two steps.

$$\begin{array}{rcl} \Omega & = & \mu\alpha.\ \alpha \to \alpha \\ \omega & : & \Omega \to \Omega \\ & = & \lambda x{:}\Omega.\ (\mathrm{unfold}\, x)\, x \end{array}$$

Then

$$\begin{array}{l} \omega\, (\mathrm{fold}^{\Omega}\, \omega) \\ \quad \longrightarrow_\beta (\mathrm{unfold}\, (\mathrm{fold}^{\Omega}\, \omega))\, (\mathrm{fold}^{\Omega}\, \omega) \\ \quad \longrightarrow_\beta \omega\, (\mathrm{fold}^{\Omega}\, \omega). \end{array}$$

At teach step we applied the only possible β-reduction and therefore the term can have no normal form. An attempt to evaluate this term will also fail, resulting in an infinite regression (see Exercise 6.16).

6.5 Exercises

Exercise 6.1 Prove that if $\Gamma; \Delta \vdash M : A$ and $\Gamma; \Delta \vdash M : A'$ then $A = A'$.

Exercise 6.2 A function in a functional programming language is called *strict* if it is guaranteed to use its argument. Strictness is an important concept in the implementation of lazy functional languages, since a strict function can evaluate its argument eagerly, avoiding the overhead of postponing its evaluation and later memoizing its result.

In this exercise we design a λ-calculus suitable as the core of a functional language which makes strictness explicit at the level of types. Your calculus should contain an unrestricted function type $A \rightarrow B$, a strict function type $A \twoheadrightarrow B$, a vacuous function type $A \dashrightarrow B$, a full complement of operators refining product and disjoint sum types as for the linear λ-calculus, and a modal operator to internalize the notion of closed term as in the linear λ-calculus. Your calculus should not contain quantifiers.

1. Show the introduction and elimination rules for all types, including their proof terms.
2. Given the reduction and expansions on the proof terms.
3. State (without proof) the valid substitution principles.
4. If possible, give a translation from types and terms in the strict λ-calculus to types and terms in the linear λ-calculus such that a strict term is well-typed if and only if its linear translation is well-typed (in an appropriately translated context).
5. Either sketch the correctness proof for your translation in each direction by giving the generalization (if necessary) and a few representative cases, or give an informal argument why such a translation is not possible.

Exercise 6.3 Give an example which shows that the substitution $[M/w]N$ must be capture-avoiding in order to be meaningful. *Variable capture* is a situation where a bound variable w' in N occurs free in M, and w occurs in the scope of w'. A similar definition applies to unrestricted variables.

Exercise 6.4 Give a counterexample to the conjecture that if $M \longrightarrow_\beta M'$ and $\Gamma; \Delta \vdash M' : A$ then $\Gamma; \Delta \vdash M : A$. Also, either prove or find a counterexample to the claim that if $M \longrightarrow_\eta M'$ and $\Gamma; \Delta \vdash M' : A$ then $\Gamma; \Delta \vdash M : A$.

Exercise 6.5 The proof term assignment for sequent calculus identifies many distinct derivations, mapping them to the same natural deduction proof terms. Design an alternative system of proof terms from which the sequent derivation can be reconstructed uniquely (up to weakening of unrestricted hypotheses and absorption of linear hypotheses in the $\top$R rule).

1. Write out the term assignment rules for all propositional connectives.
2. Give a calculus of reductions which corresponds to the initial and principal reductions in the proof of admissibility of cut.
3. Show the reduction rule for the dereliction cut.

4. Show the reduction rules for the left and right commutative cuts.

5. Sketch the proof of the subject reduction properties for your reduction rules, giving a few critical cases.

6. Write a translation judgment $S \Longrightarrow M$ from faithful sequent calculus terms to natural deduction terms.

7. Sketch the proof of type preservation for your translation, showing a few critical cases.

Exercise 6.6 Supply the missing rules for $\oplus$E in the definition of the judgment $\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : A$ and show the corresponding cases in the proof of Lemma 6.4.

Exercise 6.7 In this exercise we explore the syntactic expansion of *extended case expressions* of the form $\mathsf{case}\, M\, \mathsf{of}\, m$.

1. Define a judgment which checks if an extended case expression is valid. This is likely to require some auxiliary judgments. You must verify that the cases are exhaustive, circumscribe the legal patterns, and check that the overall expression is linearly well-typed.

2. Define a judgment which relates an extended case expression to its expansion in terms of the primitive let, case, and abort constructs in the linear λ-calculus.

3. Prove that an extended case expression which is valid according to your criteria can be expanded to a well-typed linear λ-term.

4. Define an operational semantics directly on extended case expressions.

5. Prove that your direct operational semantics is correct on valid patterns with respect to the translational semantics from questions 2.

Exercise 6.8 Define the judgment $M \longrightarrow^*_\beta M'$ via inference rules. The rules should directly express that it is the congruent, reflexive and transitive closure of the β-reduction judgment $M \longrightarrow_\beta M'$. Then prove the generalized subject reduction theorem 6.6 for your judgment. You do not need to show all cases, but you should carefully state your induction hypothesis in sufficient generality and give a few critical parts of the proof.

Exercise 6.9 Define *weak β-reduction* as allows simple β-reduction under $\otimes$, inl, and inr constructs and in all components of the elimination form. Show that if M weakly reduces to a value v then $M \hookrightarrow v$.

Exercise 6.10 Prove type preservation (Theorem 6.8) directly by induction on the structure of the evaluation derivation, using the substitution lemma 6.2 as necessary, but without appeal to subject reduction.

Exercise 6.11 Prove the subject reduction and expansion properties for recursive type computation rules.

Exercise 6.12 [*An exercise exploring the use of type conversion rules without explicit term constructors.*]

Exercise 6.13 Define a linear multiplication function mult : nat $\multimap$ nat $\multimap$ nat using the functions copy and delete.

Exercise 6.14 Defined the following functions on lists. Always explicitly state the type, which should be the most natural type of the function.

1. append to append two lists.
2. concat to append all the lists in a list of lists.
3. map to map a function f over the elements of a list. The result of mapping f over the list $x_1, x_2, \ldots, x_n$ should be the list $f(x_1), f(x_2), \ldots f(x_n)$, where you should decide if the application of f to its argument should be linear or not.
4. foldr to reduce a list by a function f. The result of folding f over a list $x_1, x_2, \ldots x_n$ should be the list $f(x_1, f(x_2, \ldots, f(x_n, init)))$, where *init* is an initial value given as argument to foldr. You should decide if the application of f to its argument should be linear or not.
5. copy, delete, and promote.

Exercise 6.15 For one of the form of lazy lists on Page 130, define the functions from Exercise 6.14 plus a function toList which converts the lazy to an eager list (and may therefore not terminate if the given lazy lists is infinite). Make sure that your functions exhibit the correct amount of laziness. For example, a map function applied to a lazy list should not carry out any non-trivial computation until the result is examined.

Further for your choice of lazy list, define the infinite lazy list of eager natural numbers $0, 1, 2, \ldots$.

Exercise 6.16 Prove that there is no term v such that $\omega\,(\mathrm{fold}^{\Omega}\,\omega) \hookrightarrow v$.

Exercise 6.17 [*An exercise about the definability of fixpoint operators at various type.*]

Exercise 6.18 Prove Lemma **??** which states that all values evaluate to themselves.

Bibliography

[ABCJ94] D. Albrecht, F. Bäuerle, J. N. Crossley, and J. S. Jeavons. Curry-Howard terms for linear logic. In ??, editor, *Logic Colloquium '94*, pages ??–?? ??, 1994.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[ACS98] Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchrounous π-calculus. *Theoretical Computer Science*, 195(2):291–423, 1998.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

[AP91] J.-M. Andreoli and R. Pareschi. Logic programming with sequent systems: A linear logic approach. In P. Schröder-Heister, editor, *Proceedings of Workshop to Extensions of Logic Programming, Tübingen, 1989*, pages 1–30. Springer-Verlag LNAI 475, 1991.

[AS01] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. *Submitted*, 2001. A previous version presented at LICS'00.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh, September 1996.

[Bib86] Wolfgang Bibel. A deductive solution for plan generation. *New Generation Computing*, 4:115–132, 1986.

[Bie94] G. Bierman. On intuitionistic linear logic. Technical Report 346, University of Cambridge, Computer Laboratory, August 1994. Revised version of PhD thesis.

[BS92] G. Bellin and P. J. Scott. On the π-calculus and linear logic. Manuscript, 1992.

[Cer95] Iliano Cervesato. Petri nets and linear logic: a case study for logic programming. In M. Alpuente and M.I. Sessa, editors, *Proceedings of the Joint Conference on Declarative Programming (GULP-PRODE'95)*, pages 313–318, Marina di Vietri, Italy, September 1995. Palladio Press.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[CHP00] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000. Special issue on Proof Search in Type-Theoretic Languages, D. Galmiche and D. Pym, editors.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[Doš93] Kosta Došen. A historical introduction to substructural logics. In Peter Schroeder-Heister and Kosta Došen, editors, *Substructural Logics*, pages 1–30. Clarendon Press, Oxford, England, 1993.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. Translated under the title *Investigations into Logical Deductions* in [Sza69].

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir93] J.-Y. Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[Her30] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et de Lettres de Varsovic*, 33, 1930.

[HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.

[Hod94] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[Hof00a] Martin Hofmann. Linear types and non-size increasing polynomial time computation. *Theoretical Computer Science*, 2000. To appear. A previous version was presented at LICS'99.

[Hof00b] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, November 2000. To appear. A previous version was presented as ESOP'00.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[HP97] James Harland and David Pym. Resource-distribution via boolean constraints. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, pages 222–236, Townsville, Australia, July 1997. Springer-Verlag LNAI 1249.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*, pages 133–147, Geneva, Switzerland, July 1991. Springer-Verlag LNCS 512.

[Hue76] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

[Kni89] Kevin Knight. Unification: A multi-disciplinary survey. *ACM Computing Surveys*, 2(1):93–124, March 1989.

[Lin92] P. Lincoln. Linear logic. *ACM SIGACT Notices*, 23(2):29–37, Spring 1992.

[Mil92] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the Workshop on Extensions of Logic Programming*, pages 242–265. Springer-Verlag LNCS 660, 1992.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[ML94] Per Martin-Löf. Analytic and synthetic judgements in type theory. In Paolo Parrini, editor, *Kant and Contemporary Epistemology*, pages 87–99. Kluwer Academic Publishers, 1994.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[MM76] Alberto Martelli and Ugo Montanari. Unification in linear time and space: A structured presentation. Internal Report B76-16, Istituto di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, July 1976.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MOM91] N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic through categories: A survey. *Journal on Foundations of Computer Science*, 2(4):297–399, December 1991.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[Pol98] Robert Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, August 1998. Proceedings of a Congress held in Venice, Italy, October 1995.

[Pra65] Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Rob71] J. A. Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

[Sce93] A. Scedrov. A brief guide to linear logic. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 377–394. World Scientific Publishing Company, 1993. Also in Bulletin of the European Association for Theoretical Computer Science, volume 41, pages 154–165.

[SHD93] Peter Schroeder-Heister and Kosta Došen, editors. *Substructural Logics*. Number 2 in Studies in Logic and Computation. Clarendon Press, Oxford, England, 1993.

[Sza69] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen.* North-Holland Publishing Co., Amsterdam, 1969.

[Tro92] A. S. Troelstra. *Lectures on Linear Logic.* CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California, 1992.

[Tro93] A. S. Troelstra. Natural deduction for intuitionistic linear logic. Prepublication Series for Mathematical Logic and Foundations ML-93-09, Institute for Language, Logic and Computation, University of Amsterdam, 1993.

[WW01] David Walker and Kevin Watkins. On linear types and regions. In *Proceedings of the International Conference on Functional Programming (ICFP'01).* ACM Press, September 2001.