4.5 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 4.2.

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 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.

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

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

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

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

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

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.

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

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

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

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 $\mathrm{inl}^B\, M$ or $\mathrm{inr}^A\, M$). 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.

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

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

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

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

Void Type. The void type **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 $\mathrm{abort}^A\, M$, since it may be viewed as a global program abort.

Unrestricted Function Type. The unrestricted function type $A \supset B$ (also written as $A \rightarrow B$ 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.

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

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

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

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.

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

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

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
		$\mid$	$\mathrm{inl}^B\, v \mid \mathrm{inr}^A\, v$	$A \oplus B$	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.

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

The elimination of $\star$ in the definition of the conditional is necessary, because a branch $\mathrm{inl}^{\mathbf{1}}\, w_1 \Rightarrow N_1$ 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 4.7).

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

An *extended case expression* has the form **case** M **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 **if** M **then** N_1 **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

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

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

Proof: See Exercise 4.8 □

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

Theorem 4.8 *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. $\square$

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 4.9). Type preservation is a simple consequence of the previous two theorems. See Exercise 4.10 for a direct proof.

Theorem 4.9 (Type Preservation) *If $\cdot; \cdot \vdash M : A$ and $M \hookrightarrow v$ then $\cdot; \cdot \vdash v : A$.*

Proof: By Theorem 4.8, $M \longrightarrow_\beta^* v$. Then the result follows by generalized subject reduction (Theorem 4.7). $\square$

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

Theorem 4.10 (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. $\square$

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.