9.4 The Curry-Howard Correspondence

The "→" type constructor comes with typing rules of two kinds:

1. an introduction rule (T-Ass) describing how elements of the type can be created,
2. an elimination rule (T-Arr) describing how elements of the type can be used.

When an introduction form (A) is an immediate subterm of an elimination form (application), the result is a redex—an opportunity for computation.

The terminology of introduction and elimination forms is frequently useful in discussing type systems. When we come to more complex systems later in the book, we'll see a similar pattern of linked introduction and elimination rules for each type constructor we consider.

9.4.1 Exercise [†]: Which of the rules for the type Bool in Figure 8-1 are introduction rules and which are elimination rules? What about the rules for Nat in Figure 8-2?

The introduction/elimination terminology arises from a connection between type theory and logic known as the Curry-Howard correspondence or Curry-Howard isomorphism (Curry and Fests, 1958; Howard, 1960). Briefly, the idea is that, in constructive logics, a proof of a proposition \( P \) consists of concrete evidence for \( P \).\(^3\) What Curry and Howard noticed was that such evidence has a strongly computational feel. For example, a proof of a proposition \( P \rightarrow Q \) can be viewed as a mechanical procedure that, given a proof of \( P \), constructs a proof of \( Q \)—or, if you like, a proof of \( Q \) abstracted on a proof of \( P \). Similarly, a proof of \( P \rightarrow Q \) consists of a proof of \( Q \) together with a proof of \( P \).

This observation gives rise to the following correspondence:

\(^3\) The characteristic difference between classical and constructive logics is the omission from the latter of proof rules like the law of the excluded middle, which says that, for every proposition \( Q \), either \( Q \) holds or \( \neg Q \) does. To prove \( Q \rightarrow \neg \neg Q \) in a constructive logic, we must provide evidence either for \( Q \) or for \( \neg Q \).

9.5 Erasure and Typability

In Figure 9-1, we defined the evaluation relation directly on simply typed terms. Although type annotations play no role in evaluation—we don't do any sort of run-time checking to ensure that functions are applied to arguments of appropriate types—we do carry along these annotations inside of terms as we evaluate them.

Most compilers for full-scale programming languages actually avoid carrying annotations at run time: they are used during typechecking (and during code generation, in more sophisticated compilers), but do not appear in the
9 Simply Typed Lambda-Calculus

compiled form of the program. In effect, programs are converted back to an untyped form before they are evaluated. This style of semantics can be formalized using an erasure function mapping simply typed terms into the corresponding untyped terms.

9.5.1 Definition: The erasure of a simply typed term $t$ is defined as follows:

$$\text{erase}(x) = x$$
$$\text{erase}(\lambda x : T_1 . t_2) = \lambda x . \text{erase}(t_2)$$
$$\text{erase}(t_1 . t_2) = \text{erase}(t_1) \cdot \text{erase}(t_2)$$

Of course, we expect that the two ways of presenting the semantics of the simply typed calculus actually coincide: it doesn't really matter whether we evaluate a typed term directly, or whether we erase it and evaluate the underlying untyped term. This expectation is formalized by the following theorem, summarized by the slogan "evaluation commutes with erasure" in the sense that these operations can be performed in either order—we reach the same term by evaluating and then erasing as we do by erasing and then evaluating:

9.5.2 Theorem:

1. If $t \rightarrow t'$ under the typed evaluation relation, then $\text{erase}(t) \rightarrow \text{erase}(t')$.
2. If $\text{erase}(t) \rightarrow \pi'$ under the typed evaluation relation, then there is a simply typed term $t'$ such that $t \rightarrow t'$ and $\text{erase}(t') = \pi'$.

Proof: Straightforward induction on evaluation derivations.

Since the "compilation" we are considering here is so straightforward, Theorem 9.5.2 is obvious to the point of triviality. For more interesting languages and more interesting compilers, however, it becomes a quite important property: it tells us that a "high-level" semantics, expressed directly in terms of the language that the programmer writes, coincides with an alternative, lower-level evaluation strategy actually used by an implementation of the language.

Another interesting question arising from the erasure function is: Given an untyped lambda-term $t$, can we find a simply typed term $t$ that erases to $t$?

9.5.3 Definition: A term $m$ in the untyped lambda-calculus is said to be typeable in $\Lambda$ if there are some simply typed term $t$, type $T$, and context $\Gamma$ such that $\text{erase}(t) = m$ and $\Gamma \vdash t : T$.

We will return to this point in more detail in Chapter 22, when we consider the closely related topic of type reconstruction for $\Lambda$...

9.6 Curry-Style vs. Church-Style

We have seen two different styles in which the semantics of the simply typed lambda-calculus can be formulated: as an evaluation relation defined directly on the syntax of the simply typed calculus, or as a compilation to an untyped calculus plus an evaluation relation on untyped terms. An important commonality of the two styles is that, in both, it makes sense to talk about the behavior of a term $t$, whether or not $t$ is actually well typed. This form of language definition is often called Curry-style. We first define the terms, then define a semantics showing how they behave, then give a type system that rejects some terms whose behaviors we don't like. Semantics is prior to typing.

A rather different way of organizing a language definition is to define terms, then identify the well-typed terms, then give semantics just to these. In these so-called Church-style systems, typing is prior to semantics: we never even ask the question "what is the behavior of an ill-typed term?" Indeed, strictly speaking, what we actually evaluate in Church-style systems is typing derivations, not terms. (See §15.6 for an example of this.)

Historically, implicitly typed presentations of lambda-calculi are often given in the Curry style, while Church-style presentations are common only for explicitly typed systems. This has led to some confusion of terminology: "Church-style" is sometimes used when describing an explicitly typed syntax and "Curry-style" for implicitly typed.

9.7 Notes

The simply typed lambda-calculus is studied in Hindley and Seldin (1986), and in even greater detail in Hindley's monograph (1997).

Well-typed programs cannot "go wrong.” —Robin Milner (1978)