Glossary of Language Theory Concepts

COMP 150 — Dataflow

February 14, 2010

Algebraic data type Sum of named products, possibly recursive. Each name is a *value constructor*. A fully saturated application of a value constructor introduces a value of algebraic type; scrutiny in a `case` expression eliminates it.

Evaluation judgment Says what a term evaluates to. Comes in a wide variety of forms, but here are some common examples:

- $\rho, \sigma \vdash e \Downarrow v$ (Big-step semantics: in environment ρ and state σ, evaluation of e terminates and produces value v.)
- $e \rightsquigarrow e'$ (Small-step semantics: e is transformed to e' in a single evaluation step.)

Formal judgment A claim, not necessarily true, in a proof system. Examples include *formal typing judgments*, *formal kinding judgments*, and *evaluation judgments*.

Formal typing judgment Claims a term has a given type. Typically $\Gamma \vdash e : \tau$, although some type systems may require more context to the left of the turnstile, e.g a system

The Γ gives the types of the free term variables. Annoyingly, Haskell writes the type ascription $e{::}\tau$, even though theorists and other languages use the single colon. The problem is all David Turner's fault.

Formal kinding judgment Claims a type has a given kind. Typically $\Delta \vdash \tau :: \kappa$, although some type systems may require more context to the left of the turnstile. The Δ gives the kinds of the free type variables, if any.

Haskell A pure, lazy functional language. Productive for programmers while also enabling rigorous reasoning.

Math A good neighborhood in the world of ideas. Used to build tractable models of real programming languages.

Product The mathematical basis for record/structure types. Written $A \times B$. In set theory, Cartesian product. In domain theory, similar to Cartesian product but also admits of a bottom element. In both, $\times$ is a binary operator and is neither associative nor commutative. Typically binary but can be n-ary, so that $A \times (B \times C)$, $(A \times B) \times C$, and $A \times B \times C$ are all different (but isomorphic).

In programming languages, always n-ary; when fields of the product are named (required in C, Java, etc), product formation may be associative as well. Few languages provide a product with unlabelled fields, but those languages include Haskell and ML.

Sum The mathematical basis for a type that represents a choice among alternatives. Written $A + B$. Also called *discriminated union*. The sum operator is a poor stepchild in set theory and in programming-language design. It has an honored place in domain theory, where it is a binary operator and is neither associative nor commutative. Typically binary but can be n-ary, so that $A + (B + C)$, $(A + B) + C$, and $A + B + C$ are all different (but isomorphic).

Usually badly supported in programming languages; when found it is always n-ary with named summands. Pascal's "variant record" (also found in CLU) is a classic sum type with named alternatives. The "enumeration literals" found in the Pascal/Modula/Ada and C/C++ families of languages are a degenerate form of sum where the type being summed is always the unit type. I know of no language that provides a sum type with anonymous summands, although the Haskell Prelude defines the `Either` type which mimics a classic mathematical sum:

```
data Either a b = Left a | Right b
```

In statically typed functional languages, the mathematical sum is found as part of an *algebraic data type*, which is always an n-ary mechanism for defining a recursive sum of products; summands are named by *value constructors*. In Scheme, there is exactly one type, "value", which is a sum type.

Term Theorist's word for an expression in a language, except that something called a "term" rarely has side effects. Terms are often represented by a metavariable e; in the terms of the λ-calculus, metavariables M and N are also popular. Here's an example grammar for terms:

$e \Rightarrow$	x	variable
$\mid$	$\lambda x{:}\tau.e$	function abstraction
$\mid$	$e_1\ e_2$	function application
$\mid$	$\Lambda\tau{::}\kappa.e$	type abstraction
$\mid$	$e\ [\tau]$	type application (aka "instantiation")
$\mid$	C	value constructor
$\mid$	$\texttt{case}\ e\ \texttt{of}\{C_i\ x_{i1}\ \ldots\ x_{in_i} \rightarrow e_i\}$	

Examples:

- $x + 1$ (plus applied to x, all applied to 1)
- $\lambda x.x$ (identity function, untyped calculus)
- $\Lambda a{::}\star.\lambda x{:}a.x$ (polymorphic identity function, typed calculus)

Type A way of classifying terms. Types are often represented by a metavariable τ. Here's a grammar for types:

$\tau \Rightarrow$	a	a type variable
$\mid$	C	a type constructor like `Bool` or `Maybe`
$\mid$	$\tau_1\tau_2$	construction of a type
$\mid$	$\forall a{::}\kappa.\tau$	a polymorphic type

What you see above is the basic core, but we can add some extra sugar:

$\tau \Rightarrow \tau_1 \times \cdots \times \tau_n$ Product, Haskell says $(\tau_1, \ldots, \tau_n)$

Type system A formal tool used both as compiler-checked documentation and as a way of rejecting programs that are likely to go wrong at run time. An expressive type system can be a helpful guide to the programmer and a joy to use; an inexpressive type system can be a ball and chain.

A typical type system comprises several components:

- *terms*, *types*, and *kinds*, which are the system's *objects of discourse*
- *formal judgments*, which are *claims* that can be made about the objects of discourse
- *inference rules*, which may be used to create *syntactic proofs* of judgments, thereby establishing the truth of the claims

Some type systems dispense with kinds and kinding judgments and instead use *type-formation rules*.

Typing derivation A *syntactic proof* of a *formal typing judgment*, thereby establishing the truth of the judgment. Here we use the judgment form $\Delta, \Gamma \vdash e : \tau$, where Δ gives the kinds of free type variables and Γ gives the types of free term variables:

$$\dfrac{\dfrac{\dfrac{x : a \in x : a}{a :: \star;\ x : a \vdash x : a}\ \textsc{Var}}{a :: \star; \vdash \lambda x{:}a.x : a \rightarrow a}\ {\rightarrow}\text{-}\textsc{Intro}}{; \vdash \Lambda a{::}\star.\lambda x{:}a.x : \forall a.a \rightarrow a}\ \forall\text{-}\textsc{Intro}$$

Value constructor Names a summand in an *algebraic data type*. Required for pattern matching in case expressions. Some predefined value constructors in Haskell include `True`, `False`, `Just`, and `Nothing`.