Values, Types and Expressions

Overview

- Typology of values and types:
  - primitive, composite and recursive
  - set theoretical representation
- Using values: first class values and second class values
- Typology of expressions

**Typology of Values and Types**

- **Values**
  - A *value* is any entity that exists during a computation
  - Alternatively, a value is anything that may be manipulated by a program
  - Operational definition (in Pascal): anything that may be passed as an argument to a subroutine
  - Two ways to classify values in a language:
    - By their *structure* (the way they are defined)
    - By their *functionality* (the way they are manipulated)

- **Value Structure**
  - **Primitive value**: *is not* composed of other values
    - truth values, characters, integers, reals, pointers
  - **Composite value**: *is* composed of other values
    - records, arrays, sets, files
  - The ways to create composite values in a language are usually independent of its implementation
  - The *set* of legal values in a language’s implementation:
    - a closure of the primitive values in this *implementation* under the mechanisms the language *specification* allows for creating composite values
Value Manipulation

- Operations on values
  - Passing them to procedures as arguments
  - Returning them through an argument of a procedure
  - Returning them as the result of a function
  - Assigning them into a variable
  - Using them to create a composite value
  - Creating/computing them by evaluating an expression
  - …

- A value for which all these operations are allowed is called a **first-class value**

- We are used to integer or character values, but function values are also possible!

Values’ Status in Pascal

- **First-class values**
  - Only the primitive values: truth values, characters, enumerands, integers, reals, pointers.

- **Lower-class values** – can be passed as arguments, but cannot be stored, or returned, or used as components in other values
  - composite values (records, arrays, sets and files): cannot be returned!
  - procedure and function abstractions
  - references to variables (unless disguised as pointers)

Values in ML

- All the values in ML are first-class values:
  - **Primitive values**: truth values, integers, reals, strings.
  - **Composite values**: records, tuples (records w/o field names), constructions (tagged values), lists, arrays.
  - **Function abstractions**
  - **References to variables**

- What we can do in ML but not in Pascal:
  - create a record composed of two functions
  - write a function that gets a function \( f: \text{int} \rightarrow \text{int} \) and returns the composition of \( f \) with itself
  - write an expression whose value is a reference to a variable

Expressions

- **Expression**: A part of a program that is translated into a value (*evaluated*) during computation

- Some examples:
  - \( 3.1416 \times '\%' \cdot \text{"Hello, world"} \) (Pascal)
  - \( 2 \cdot a[i] + 7 \cdot \text{sqr}(4) \cdot q^\cdot \text{head} \) (Pascal)
  - if leap (thisyear) then 29 else 28 (ML)

- We’ll see more later….
The Need for Types
- In machine language – all values are just bit patterns => they are untyped
- In assembler languages – there is already a difference between addresses and data
  - The sum of two addresses is ill-defined => tagged architectures that add type information to values in runtime
  - Problem: usually deals only with predefined types, but not with user-defined types
- Observation: a type is also a property of
  - memory cells – where typed values are stored
  - expressions – from which typed values are evaluated
- High-level languages: attach types also to expressions (as will be seen later)
- Type checking (of expressions, arguments, etc.) is crucial to discovering more errors automatically

Historical Background
- Initially, an assumption of set theory:
  - For every imaginable property, there exists a set of objects that satisfy this property
- Russell’s Paradox: The following set R leads to a contradiction: \( R = \{ x \mid x \notin x \} \)
  - If we assume that R is a member of R we must conclude that R is not a member of R, and vice versa!
  - In a town, a barber shaves precisely those men who do not shave themselves; who shaves the barber?
- Resolution: Use tagging. The following is allowed only if \( x \) is restricted to range over a set \( P \), where the set of possible \( Ps \) is predefined
  \( S_p = \{ x \in P \mid \ldots \} \)
- Each set carries a tag. These tags evolved into types in programming languages

Types as Sets of Values
- Every type corresponds to a set of values
  - A value \( v \) is of type \( T \) => \( v \) is in a set that \( T \) defines
  - An expression \( E \) is of type \( T \) => the result of evaluating \( E \) is a value of type \( T \)
- However, not every set of values corresponds to a type!
- A type is a set of values with operations that can be applied uniformly for every value in the set
- For example:
  - \{3.14, false, "November"\} does not correspond to any type in any language
  - \{false, true\} corresponds to a type in many languages
- Conclusion: the definition of the set of types is a pragmatic decision, based on the objectives of the language, and not on the formal properties of its values

Types
- Primitive type: a type that has only primitive values
- Composite type: a type that has (many) composite values
- Recursive type: a type that has (many) values that are composed both of values of the same type as well as values of other (base) types
Primitive Types

- A **type system** is a collection of sets of values and the relations among these sets.
- The primitive types are the building blocks of this collection: their values **cannot be broken** further into smaller values.
- Primitive types are subdivided into:
  - **Rudimentary**: These types are built-in into the programming language (e.g. truth value, integer, real, character).
  - **Non-Rudimentary**: User defined primitive types.
- Examples of non-rudimentary primitive types:
  - **Enumerated types** (Ada, Pascal, C, ...)
    ```pascal
    TYPE Month = (January, February, March, April, May, Jun, July, August, September, October, November, December)
    TYPE DayOfMonth = 1..31
    ```
  - **Subranges** (Pascal, ...): a range of consecutive values.

Primitive Types (cont’d)

- **Choice of rudimentary primitive types** tells much about the intended purpose of the programming language:
  - **Fortran** – numerical computation. Choice of precision of real and complex numbers.
  - **Cobol** – data processing. Fixed length strings, fixed point numbers.
  - **Snobol** – string processing. Variable length strings.
- **Point of confusion**: Different programming languages use different names for the same primitive type.
  - Pascal: `Boolean, Integer, Real`
  - ML: `bool, int, real`
- **Point of difficulty**: Different implementations of the language may use different sets of values for the same type.
  - Some languages have several types for integers and reals:
    - C and C++ have `float` and `double` for reals.
    - Java has `byte, short, int, long` for differing ranges of integers.

Cardinality of a Type

- The number of values in the set corresponding to that type.
- For type T, denoted #T
- #Boolean = 2
- #char = 256 (ISO Latin set) or 101,713 (Unicode, as of v5.1)
- #Months = 12
- #integer = ???
  - Usually implementation dependent, or defined for special types of integers (#byte = 256)
  - In Pascal: 2*(maxint+1)
- #real – even more complicated (not unique)
Ordering Primitive Types

- **Ordered type**: a type for which a total order relation is defined.
  - Useful in iterators and conditional expressions
- **Unordered types**: complex (FORTRAN, Mathematica, Matlab)
- **Discrete type**: an ordered type whose set of values has a one-to-one order preserving mapping with a (range of) integers. Examples: `Boolean`, `char`, `integer`.
- **Ordered indiscrete types**: `string`, `real`.
- **Pascal**: Only discrete types can be used in:
  - indices for loops and arrays (reasonable policy)
  - `Case` statements (arbitrary policy, as in machine language)
- **C++**: Only discrete types can be used as template arguments.

Semi-Primitive Types

- **Semi-primitive types 1**:
  - **Atomic**: values cannot be broken down into values of smaller types
  - **Non-rudimentary**: defined by the programmer
  - **Example**: enumerated types
- **Semi-primitive types 2** (pseudo-primitive types):
  - **Non-atomic**: values can be broken down into values of smaller types
  - **Rudimentary**: are built-in into the programming language
  - **Example**: strings in several programming languages
    - Values can be broken into smaller strings
    - Type string is built into the language

Strings

- **String**: a sequence of characters. Useful data type. Supported in one way or another by all modern programming languages.
- **No consensus. Issues**:
  - Primitive (as in Icon) or composite (as in Pascal) type?
  - Fixed length (as in Cobol) or variable length (as in C)?
  - What string operations are supported?
  - String literals? Delimiters or quotes?
- **ML**: a primitive type of any length. Operations: equality test, concatenation, decomposition are built-in.
- **Pascal and Ada**: an array of characters. All array operations are available. Disadvantage: all string operations must be defined in terms of these fixed length arrays.
- **Algol-68**: same as Pascal and Ada, but with flexible arrays (size changed during runtime).
- **C**: semi-flexible arrays. String literals.
- **Prolog and Miranda**: a list of characters. Only the first character can be selected. All other string operations, e.g., ordering, substrings, etc. must be explicitly defined.

Composite Types

- **Type operators**: create a new composite type out of primitive and composite types:
  - tuples, records, variants, unions, arrays, sets, strings, lists, trees, serial files, direct files, relations, etc.
- **All can be understood in terms of set theory operators**:
  - Cartesian products: `tuples and records`.
  - Disjoint unions: `variants and unions`.
  - Mappings: `arrays and functions`.
  - Power sets: `sets`.
  - Enclosure under operator: `recursive types, dynamic data structures`.
- **There are two mathematical notations that provide a common foundation for composite types in many programming languages**:
  - **Set theory**: primitive types are (arbitrary) sets of values, composite types are other sets defined in terms of these sets
  - **Symbolic functions**: primitive types are formal variables, composite types are polynomials over these variables
Cartesian Products:_tuples and records

- Given two types \( S \) and \( T \), we denote their Cartesian product by \( S \times T \).
  - \( S \times T = \{(x, y) \mid x \in S; y \in T\} \)
  - \(#(S \times T) = #S \times #T\)
  - This can be generalized to more than two sets: \( S_1 \times S_2 \times \ldots \times S_n \)

- The tuple of ML, the records of Cobol, Pascal, Ada, Icon and ML, and the so-called structures Algol-68 and C can all be understood in terms of Cartesian products.

- ML tuples:
  - \( \text{type person} = \text{string} \times \text{string} \times \text{int} \times \text{real} \)
  - A decomposition of a value `someone` of type `person`:
    - `val (surname, forename, age, height) = someone if age >= 18 then ... else ...`

- ML records (labeled Cartesian product):
  - \( \text{type person} = \{ \text{surname: string, forename: string, age: int, height: real} \} \)

---

The “Unit” Type

- Primitive types have cardinalities, e.g.,
  - \(#\text{Boolean} = 2\)
  - \(#\text{integer} = 2^n\)
  - \(#\text{real} = \ldots\)

- A type with cardinality 1?
  - Values of such a type require no storage!
  - C’s `void`, Pascal’s `nil`, etc.

- Technically, the Unit type does not have to be a primitive type
  - It can be thought of as a record with no fields in it: the neutral element of the record “type operator”

---

Cartesian Products (cont’d)

- Homogenous tuples: a special case of the Cartesian product is one where all the tuple components are chosen from the same set:
  - \( S^n = S \times S \times \ldots \times S \)
  - \(#(S^n) = (#S)^n\)

- We observe that \( S^0 \) has exactly one value: the 0-tuple.
  - Unit = \{ () \}
  - Unit is not the empty set: it contains a single value which happens to be a tuple with no components.

- Unit corresponds to the unit type of ML, and void type of C.

---

Type void in C and Other Subtle Points

- `void f1(int)`
  - is not a function that does not return anything
  - It is rather a function that can return in only one way, i.e. returns type unit

- `int f2(void)`
  - Takes no arguments
  - Can be thought of as taking an argument of type unit

- `int f3(...)`;
  - function is declared to have a variable number of arguments

- `extern f4()`:
  - return type is implicitly int
  - In K&R C, no declaration of arguments
  - In C++, function takes no argument (an argument of type unit)
Unit Type in C

*void* is not an ordinary type in C. Although there is a *void* return type and a *void* argument, there is no way to make variables of type *void*. However, we can emulate unit type in C using one of the following tricks:

```
typedef enum Unit {
    unit
} Unit;
```

Or...

```
typedef struct {
} Unit;
```

Or...

```
typedef int Unit[0];
```

Disjoint Unions:  
Choice Type Operators

- Each value is chosen from either set $S$ or set $T$
  - $S + T = \{ \text{left } x \mid x \in S \} \cup \{ \text{right } y \mid y \in T \}$
  - $\#(S + T) = \#S + \#T$

- Examples:
  - C's union
  - Pascal's variant record
  - ML's constructions
  - Java and Eiffel: missing! Typical to OO languages - the need for choice type operators is diminished with OO features.

- Useful for representing *pointers*:
  - Pointer to type $X$:
    - either points to a value of type $X$, or is nil/void/0 (depending on your terminology)
    - can be thought of as a choice between Unit type and the type $(X)$

Variant Records in Pascal

- Syntax of definition:
  ```
  record case I: T of 
    L_i: (I_i: T_i); 
    ... 
    L_n: (I_n: T_n)
  end
  ```

- Example:
  ```
  type Accuracy = (exact, approx);
  Number = record case tag: Accuracy of
    exact: (ival: Integer);
    approx: (rval: Real)
  end
  ```

  The values of type *Number* are:
  - {..., exact -2, exact -1, exact 0, exact 1, exact 2,...}
  - {..., approx -1.0, ..., approx 0.0, ..., approx 1.0, ...}

Using Pascal’s Variant Record

- In Pascal, Turbo Pascal and in C, a disjoint union type may be accessed in the same way as ordinary Cartesian product elements. This is unsafe!

- Consider the following code:
  ```
  VAR n: Number;
  ...
  n.tag := exact; (* n.ival is still undefined *)
  n.ival := 7;
  ...
  n.tag := approx; (* n's value is changed in one step from exact 7 to approx undefined *)
  ```

- Safe decomposition of a variant record
  ```
  function round_num(n: Number): Integer;
  case n.tag of
    exact: round_num := n.ival;
    approx: round_num := round(n.rval)
  end;
  ```
Variant Records in ML

- ML construction definitions:
  
  ```
  datatype number = exact of int | approx of real;
  ```

- Construction literals:
  
  ```
  exact(i + 1)  approx(r/3.0)
  ```

- Decomposing a construction:
  
  ```
  case n of exact i => i | approx r => round(r);
  ```

- Observe that such a mechanism may enhance a dynamic typing system, in the same way that C’s ternary `?:` operator does.

Tagging in Choice Types

- Tag: A mechanism for storing the selection made in a choice type.

- Different programming languages provide different levels of support for tagging:
  - **ML**: Tagging is built into the language.
    - The tag is implicit: You cannot access a “Tag Field” directly.
    - Correct tagging is enforced: There is no way to store a value into one choice selection and read it from another.
  - **Pascal**: The compiler forces a definition of a tag field. When a tag field is updated, the record’s corresponding fields are created accordingly.
    - But: the compiler does not enforce correct access to record according to tag.
  - **Turbo-Pascal**: A Pascal extension which allows the programmer to get away without defining a tag field.
  - **C**: Responsibility lies entirely with programmer, both for defining (or not) a tag field and for using it correctly.
  - **Pointers in all languages**: Tagging is implicit in tests for null pointer. However, not all languages complain if you try to read from or write to a null pointer.

- We presume that tagging exists in all choice type operators, be it by language design or by programmer responsibility.

The Need for Tagging

- Recall that tagging is needed for safe decomposition.
- But the need is also observed in the set-theoretical representation.
- Suppose that the representation of `height` were simply the union set $I \cup I$.
- Then we would get: $I \cup I = I$

  ```
  typedef union height {
    int cm;
    int in;
  };
  ```

- With tagging we have e.g.:
  
  `\(I_{\text{in}}\times\{\text{in}\} \cup I_{\text{cm}}\times\{\text{cm}\}\)`

- And we can simply represent `height` by a disjoint union of `I_{\text{cm}}` and `I_{\text{in}}`.

More on tags

- The basic operations on disjoint union $S+T$:
  - Construction: build a disjoint union value by taking a value from $S$ or $T$ and adding the appropriate tag
  - Tag test: determine if the variant came from $S$ or $T$ (`right v` is from $T`)
  - Projection: get the value by removing the tag (`right v` will return `v`)

- A less desirable (albeit equivalent) alternative for representing `height` would be a structure of a Boolean and an integer.
Choice and Enumerated Types

- An enumerated type can be simulated as a choice between units:

```c
typedef enum Suit {
    diamond, heart, spade, clover,
} Suit;
```

- A choice between four empty structures with different names:

```c
typedef union Suit {
    struct {} diamond;
    struct {} heart;
    struct {} spade;
    struct {} clover;
} Suit;
```

The “none” Type

- `none`: a type with cardinality 0
  - This type has no legal values
  - It would be meaningless to define variables of this type, since no value could ever be stored into these variables.
- **Example**: The function `exit()` in C, which never returns.
  - The return type of this function should not be `void` but rather `none`.

Emulating `none` in C

- Enumerated type, with no possible values:

```c
typedef enum {} none;
```

- A union which has no fields. There is no legal way to assign a value to such a union:

```c
typedef union {} none;
```

Mappings: arrays and functions

- Arrays as mappings:
  - the type "array [S] of T" (in Pascal S must be discrete) corresponds to a mapping S→T.
  - the type "array [S₁, S₂, ..., Sₙ] of T" corresponds to a mapping (S₁×S₂×...×Sₙ)→T,
  - alternatively, using **Currying**, to (S₁→(S₂→(...→Sₙ))))→T
- Functions as mappings:
  - "function even (n: Integer) : Boolean" corresponds to a function Integer→Boolean.
  - **Note**: functions in most programming languages are algorithmic implementations of mathematical functions/mappings. They have however other properties that functions do not have:
    - **efficiency**
    - **side effects**

```c
#S = #T#S
```
Power Sets: sets

- \(\#(\wp(T)) = 2^T\)
- corresponds to "set of T" in Pascal
- Ideally: isomorphic to T\(\rightarrow\)Boolean
- In Pascal: for efficiency T must be discrete
  => in effect similar to "array [T] of Boolean"

Power Sets: sets

- \(\#(\wp(T)) = 2^T\)
- corresponds to “set of T” in Pascal
- Ideally: isomorphic to T\(\rightarrow\)Boolean
- In Pascal: for efficiency T must be discrete
  => in effect similar to “array [T] of Boolean”

Recursive Types

- A recursive type is defined in terms of itself.
- Modern Languages like ML allow direct definition.
- Example 1 - IntList = nil Unit + cons(Integer x IntList)
  => lists in ML:
    datatype intlist = nil | cons of int * intlist
    This is an abbreviation for nil of unit | cons of ...
- Values of intlist:
  nil
  cons(11,nil)
  cons(2,cons(3,cons(5,cons(7,cons(11,nil))))))

- These can be obtained by substituting values so far in the right-hand side of the definition, to get new values.
- We could also consider infinite lists, but usually don’t...

More on recursive lists

- We will see that such recursive definitions are useful and can be efficient.
- Each list is finite, but there is no global limit on the number of elements (so there are infinitely many lists defined here)
- Further, ML has a pre-defined type constructor called list:
  int list
  bool list
  int list list
- Operations
  - Basic Predefined: test for emptiness, select head, select tail.
  - Concatenation and length (could be defined in terms of basic ops)

- In Ada/C/Pascal, recursive types must be defined with pointers.

Another Recursive Type

- Example 2: trees
  datatype inttree = leaf of int | branch of inttree * inttree
- Possible values:
  - leaf 11
  - branch (leaf 11, leaf 5)
  - branch (branch (leaf 11, leaf 5), leaf 11)
- Note: most values of this recursive type are composed of values of the same type.
- Set theoretical representation:
  Integer_Tree = Integer \(\cup\) (Integer_Tree x Integer_Tree)