# 부록: 제네릭 프로그래밍 활용

- 특정 데이터 형식에 의존하지 않으면서 보다 일반적인 유형(type)에 대해 작동하는 알고리즘 작성 기법

- 동일한 알고리즘을 보다 일반적인 상황에 대해서 작동하도록 하는 프로그래밍 기법

- 제네릭 프로그래밍에 사용되는 기법은 다양함
    - 모듈 활용
        - 파이썬처럼 고차함수를 지원하는 프로그래밍언어의 경우 모듈화를 활용할 수 있음.
    - 다형성(polymorphism) 활용: 다양한 유형(자료형)을 마치 하나의 자료형 처럼 다루는 기법
        - 자바, 파이썬 등에서 제공하는 제네릭 프로그래밍 기법에 사용됨

## 모듈 활용

- n-퀸 문제의 되추적 함수 `backtracking_search_queens()`와 m-색칠하기 문제의 되추적 함수 `backtracking_search_colors()`는 사실상 동일한 알고리즘을 사용함.

- 차이점: 유망성을 확인하는 함수 `promissing_queens()`와 `promissing_colors()`가 서로 다름.

- 두 함수가 사용되는 위치를 `backtracking_search()` 함수의 매개변수(파라미터)로 추상화하여 사용 가능.
    - 대신에 두 개의 유망성 함수는 각각 독립된 모듈(파이썬 파일)로 저장. 서로 다른 모듈에 저장하기에 동일한 이름, 예를 들어 `promissing()` 등으로 지정 가능.

**되추적 알고리즘: 고차 함수 활용**

- `promissing` 매개변수 추가: 유망성 확인 함수를 인자로 받음.

- `promissing` 매개변수가 기대하는 함수 인자의 유형(type)
    - 매개변수 자료형: `int`와 `Dict[int, int]`
    - 반환값 자료형: `bool`
    - 따라서 `promissing` 함수의 유형: `Callable[[int, Dict[int, int]], bool]`

- 참조: 파이썬 유형 힌트 사용법 (https://python.flowdas.com/library/typing.html)

- 이어지는 코드는 두 개의 promissing 함수가 각각 `constraint_queens.py`와 `constraint_colors.py` 저장되어 있음을 가정함.

In [1]:

```python
from typing import List, Dict, Callable

def backtracking_search(promissing: Callable[[int, Dict[int, int]], bool],
                        assignment: Dict[int, int] = {}):
    """assignment: 각각의 변수를 키로 사용하고 키값은 해당 변수에 할당될 값"""

    # 모든 변수에 대한 값이 지정된 경우 조건을 만족시키는 해가 완성된 것임
    if len(assignment) == len(variables):
        return assignment

    # 아직 값을 갖지 않은 변수들이 존재하면 되추적 알고리즘을 아직 할당되지 않은 값을 대상으로 이어서 진행
    unassigned = [v for v in variables if v not in assignment]
    first = unassigned[0]
    for value in domains[first]:
        # 주의: 기존의 assignment를 보호하기 위해 복사본 활용
        # 되추적이 발생할 때 이전 할당값을 기억해 두기 위해서임.
        local_assignment = assignment.copy()
        local_assignment[first] = value
        # local_assignment 값이 유망하면 재귀 호출을 사용하여 변수 할당 이어감.
        if promissing(first, local_assignment):
            result = backtracking_search(promissing, local_assignment)
            # 유망성을 이어가지 못하면 되추적 실행
            if result is not None:
                return result

    return None
```

### 예제: 4-퀸과 4-색칠하기 문제

- 아래 코드는 4-퀸과 4-색칠하기 문제를 준비하는 세팅과정임.
    - 4-색칠하기 문제의 경우 3-색칠하기가 가능하면 3 가지 색상만 사용함.
    - 되추적 알고리즘의 특성상 가능하면 적은 열/색상을 사용함.

In [2]:
```
# 변수: 네 개의 퀸 또는 네 마디의 번호, 즉, 1, 2, 3, 4
variables : List[int] = [1, 2, 3, 4]

# 도메인: 네 개의 퀸이 위치할 수 있는 가능한 모든 열 또는 각각의 마디에 사용될 수 있는 가능한 모든 색상
domains: Dict[int, List[int]] = {}
columns = [1, 2, 3, 4]  # 네 개의 열 또는 색상
for var in variables:
    domains[var] = columns
```

### *4-퀸 문제 해결*

In [3]:
```
from constraint_queens import promissing

print(backtracking_search(promissing))
```

```
{1: 2, 2: 4, 3: 1, 4: 3}
```

### *4-색칠하기 문제 해결*

- 3 가직 색상만 사용되는 것에 주의할 것.

In [4]:
```
from constraint_colors import promissing

print(backtracking_search(promissing))
```

```
{1: 1, 2: 2, 3: 3, 4: 2}
```

## 다형성 활용

- n-퀸과 m-색칠하기 문제에서 변수 또는 도메인 값들이 숫자가 아닌 문자열 등 다른 자료형이 사용될 수 있음.
    - 체스판의 열은 원래 1에서 8까의 숫자 대신에 a 부터 h 사이의 알파벳으로 표시됨.
    - 지도를 색칠할 때 영역은 번호 보다는 이름으로 표현함.

- 하지만 자료형이 달라진다 하더라도 되추적 알고리즘 자체가 변하지는 않음. 다만, 유망성을 확인하는 함수가 사용되는 자료형에 따라 조금 수정될 필요는 있음.

- 예를 들어, n-퀸 문제의 유망성에서 대각선상에 위치하는지 여부를 판정할 때 숫자를 다루는 경우와 문자열을 다루는 경우는 조금 다를 수밖에 없음.

**클래스 활용**

- 서로 연관된 속성과 함수들을 하나의 클래스로 묶어서 활용할 수도 있음.
    - 되추적 알고리즘과 유망성 확인 함수를 하나의 클래스로 묶어서 사용
    - 문제에 따라 클래스의 인스턴스를 다르게 생성

- 주의사항
    - 되추적 알고리즘(`backtracking_search()`)는 변하지 않음.
    - 유망성 확인 메서드(`promissing()`)는 문제(인스턴스)별로 달라짐.

***제네릭 클래스***

- 문제마다 사용되는 자료형이 다를 수 있음.
    - 열 또는 색상: 1, 2, 3, 4 대신에 a, b, c, d 등 문자열 사용할 수도 있음.

- 되추적 알고리즘은 사용되는 자료형에 상관없이 작동함.

- 따라서 일반적인 유형에 대해 작동하는 제네릭 클래스의 메서드로 구현 가능

- 제네릭 클래스의 사용법은 다음과 같음.

```
from typing import Generic, TypeVar
  # 제네릭 변수. 여기서는 두 개의 제네릭 변수 지정
  # 해당 클래스의 인스턴스를 선언하면 적절한 자료형으로 자동 대체됨.
  V = TypeVar('V')
  D = TypeVar('D')

  # 아래 제네릭클래스에서 사용되는 변수 또는 값의 일부가 V 또는 D 자료형과 연관됨
  class 제네릭클래스(Generic[V, D]):
      클래스본문
```

***추상클래스와 추상메서드***

- 유망성 확인 함수인 `promissing()` 메서드는 추상메서드(abstract method)로 지정한 후 문제에 따라 다르게 정의해야 함.

- 추상메서드: 클래스 내에서 함수로 선언만 되고 정의되지 않은 상태로 남겨진 메서드

- 추상클래스: 추상메서드를 한 개 이상 포함한 클래스

- 추상클래스와 추상메서드 사용법은 다음과 같음.
    - 전제조건: abc 모듈에서 ABC 클래스와 abstractmethod 장식자 불러오기
    - 추상클래스는 ABC 클래스를 상속해야 함.

```
from abc import ABC, abstractmethod

class 추상클래스명(ABC):
  @abstractmethod
  def 추상메서드(self, 매개변수, ..., 매개변수):
      ...

  # 기타 메서드 및 속성
```

**되추적 알고리즘 구현: 클래스 활용**

- 이후에 사용되는 코드는 고전 컴퓨터 알고리즘 인 파이썬 (https://github.com/davecom/ClassicComputerScienceProblemsInPython) 의 3장 코드를 단순화하였음.

### *제네릭 클래스 선언*

- 제네릭 클래스 선언에 필요한 제네릭 변수 지정
    - `V`: 주어진 문제에 사용되는 변수들의 자료형.
        - 예를 들어 1, 2, 3 등의 int 또는 서울, 경기, 인천 등의 str
        - 활용: `variables: List[V]`
    - `D`: 주어진 문제에 사용되는 도메인에 포함된 값들의 자료형.
        - 예를 들어 1, 2, 3 등의 int 또는 a, b, c, 빨강, 녹색, 파랑 등의 str
        - 활용: `domains: List[D]`

In [5]:
```
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod

V = TypeVar('V')
D = TypeVar('D')
```

- Constraint 클래스 선언
  - 유망성 확인함수를 추상메서드로 가짐.
  - Generic 클래스와 ABC 클래스 모두 상속

In [6]:
```
class Constraint(Generic[V, D], ABC):
    @abstractmethod
    def promissing(self, variable: V, assignment: Dict[V, D]) -> bool:
        ...
```

- `Backtracking` 클래스 선언
    - 되추적 알고리즘을 메서드로 가짐.
    - `Generic` 클래스 상속
- 주의: `Optional` : 경우에 따라 `None` 이 값으로 사용될 수 있음을 암시함.
    - `Optional[T]`의 의미: 기본적으로 자료형 T의 값을 사용하지만 `None` 이 사용될 수도 있음.

In [7]:

```
class Backtracking(Generic[V,D]):
    # 제약조건을 지정하면서 인스턴스 생성
    def __init__(self,
                 variables: List[V],
                 domains: Dict[V, List[D]],
                 constraint: Constraint[V,D]) -> None:
        self.variables: List[V] = variables
        self.domains: Dict[V,List[D]] = domains
        self.constraint = constraint

    # promissing 메서드가 Constraint 클래스에서 선언되었기에 따로 인자로 받지 않음.
    def backtracking_search(self,
                            assignment: Dict[V,D] = {}) -> Optional[Dict[V,D]]:
        if len(assignment) == len(self.variables):
            return assignment
        unassigned: List[V] = [v for v in self.variables if v not in assignment]
        first: V = unassigned[0]
        for value in self.domains[first]:
            local_assignment = assignment.copy()
            local_assignment[first] = value
            if constraint.promissing(first, local_assignment):
                result = self.backtracking_search(local_assignment)
                if result is not None:
                    return result

        return None
```

### *8-퀸 문제 해결하기*

- 8개의 퀸과 8개의 열 사용하기
    - 열을 a, b, ..., h 로 사용

- 퀸이 위치하는 열을 번호가 아닌 문자열로 처리하기에 `promissing()` 메서드 정의가 이전과 조금 달라짐.
    - 동일한 열 또는 동일한 대각선 상에 있는지 여부는 순서가 중요함.
    - `enumerate()` 함수를 활용하여 항목의 인덱스를 활용할 수 있음.

In [8]:
```
# 변수: 여덟 개의 퀸 1, 2, ..., 8
variables : List[int] = [1, 2, 3, 4, 5, 6, 7, 8]

# 도메인: 여덟 개의 퀸이 위치할 수 있는 가능한 모든 열. 문자열 사용
domains: Dict[int, List[str]] = {}
columns = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

for var in variables:
    domains[var] = columns

# columns의 순서 지정하기
col2ind = {column : index+1 for (index, column) in enumerate(columns)}
```

- 문제에 맞는 `Constraint` 클래스의 인스턴스 생성하려면 먼저 `Constraint` 클래스를 상속하는 구상클래스를 선언해야 함.
    - 이유: 추상 클래스는 바로 인스턴스를 만들 수 없음.
    - `promissing()` 메서드를 구현해야 함.
        - 이전에 사용한 constraint_queens 모듈의 `promissing()` 함수와 동일.
    - 주의: 열의 순서가 중요하기에 `col2ind` 사전을 활용함

In [9]:

```python
class ConstraintQueens(Constraint[int, str]):

    # promissing 메서드 정의해야 함.
    def promissing(self, variable: int, assignment: Dict[int, str]) -> bool:
        # q1r, q1c: 첫째 퀸이 놓인 마디의 열과 행
        # 키값이 문자열이기에 해당 문자열의 인덱스를 대신 사용함. 즉, col2ind 활용
        for q1r, q1c in assignment.items():
            q1c = col2ind[assignment[q1r]] # 첫째 퀸의 열
            # q2r = 첫째 퀸 아래에 위치한 다른 모든 퀸들을 대상으로 조건만족여부 확인
            for q2r in range(q1r + 1, len(assignment) + 1):
                q2c = col2ind[assignment[q2r]] # 둘째 퀸의 열
                if q1c == q2c: # 동일 열에 위치?
                    return False
                if abs(q1r - q2r) == abs(q1c - q2c): # 대각선상에 위치?
                    return False

        # 모든 변수에 대해 제약조건 만족됨
        return True
```

In [10]:
```
constraint = ConstraintQueens()
backtracking = Backtracking(variables, domains, constraint)

backtracking.backtracking_search()
```

Out[10]:
```
{1: 'a', 2: 'e', 3: 'h', 4: 'f', 5: 'c', 6: 'g', 7: 'b', 8: 'd'}
```

## 4-퀸 문제

In [11]:
```python
# 변수: 네 개의 퀸 1, 2, 3, 4
variables : List[int] = [1, 2, 3, 4]

# 도메인: 네 개의 퀸이 위치할 수 있는 가능한 모든 열. 문자열 사용
domains: Dict[int, List[str]] = {}
columns = ['a', 'b', 'c', 'd']

for var in variables:
    domains[var] = columns

# columns의 순서 지정하기
col2ind = {column : index+1 for (index, column) in enumerate(columns)}
```

In [12]:
```python
constraint = ConstraintQueens()
backtracking = Backtracking(variables, domains, constraint)

backtracking.backtracking_search()
```

Out[12]:
```
{1: 'b', 2: 'd', 3: 'a', 4: 'c'}
```

### 4-색칠하기

- 색상을 번호가 아닌 RGB(빨강, 녹색, 파랑)으로 표현
    - 이전에 사용한 constraint_colors 모듈의 `promissing()` 함수와 동일.
    - n-퀸 문제와는 달리 도메인 값의 순서가 중요하지 않음.

In [13]:
```
# 변수: 네 개의 퀸 또는 네 마디의 번호, 즉, 1, 2, 3, 4
variables : List[int] = [1, 2, 3, 4]

# 도메인: 네 개의 퀸이 위치할 수 있는 가능한 모든 열 또는 각각의 마디에 사용될 수 있는 가능한 모든 색상
domains: Dict[int, List[str]] = {}
columns = ['R', 'G', 'B']
for var in variables:
    domains[var] = columns
```

- 문제에 맞는 Constraint 클래스를 구상클래스로 상속하기

In [14]:
```python
class ConstraintColors(Constraint[int, str]):

    # promissing 메서드 정의해야 함.
    def promissing(self, variable: int, assignment: Dict[int, str]) -> bool:
        # 각 마디에 대한 이웃마디의 리스트
        constraints = {
            1 : [2, 3, 4],
            2 : [1, 3],
            3 : [1, 2, 4],
            4 : [1, 3]
        }

        for var in constraints[variable]:
            if (var in assignment) and (assignment[var] == assignment[variable]):
                return False

        return True
```

In [15]:
```
constraint = ConstraintColors()
backtracking = Backtracking(variables, domains, constraint)

backtracking.backtracking_search()
```

Out[15]:
```
{1: 'R', 2: 'G', 3: 'B', 4: 'G'}
```