DFA.h

StatePool Q;

// S must be a singleton set, or empty. |S| <= 1

StateSet S;

StateSet F; // final states

DTransRel T;

DTransRel :public StateTo<DTrans>

DtransRel: public StateTo<Dtrans>

StateTo<Dtrans>

DTrans :protected TransImpl

|  |  |  |  |
| --- | --- | --- | --- |
| 继承方式 | 基类的public成员 | 基类的protected成员 | 基类的private成员 |
| Public | 仍然为public成员 | 仍为protected成员 | 不可见 |
| Protected | 成为protected成员 | 成为protected成员 | 不可见 |
| Private | 成为private成员 | 成为private成员 | 不可见 |

// Implement the basics of a char to StateSet map (a set of TransPair's).

// It is used by DTrans and Trans.

class TransImpl

{

protected:

// Constructors, destructors, operator=

// Assume that delete[] 0 is okay.

inline TransImpl();

// Copy constructor allocates more memory.Should use(use-counting) for efficiency.

TransImpl(const TransImpl& r);

// Destructor is virtual, simple.

virtual ~TransImpl();

// operator=() must copy the memory.

const TransImpl& operator=(const TransImpl& r);

// Some member functions for making transitions:

// What are all of the labels on transitions in \*this?

CRSet out\_labels() const;

// What are all transition labels with destination in r?

CRSet labels\_into(const StateSet& r) const;

//Some special member functions :

// Clear out all previous transitions, and zero the domain.

inline void reincarnate();

// Add a transition to the set.

TransImpl& add\_transition(const CharRange a, const State q);

// Allow classes that inherit from TransImpl to have access to the real data:

inline TransPair& transitions(const int i) const;

// Output the transitions.

friend std::ostream& operator<<(std::ostream& os, const TransImpl& r);

// Maintain the class invariant.

inline int class\_invariant() const;

// Helpers

void ensure\_min\_size(int w);

// Implementation details:

// How many transitions are there.index of TransPair \*data (include expansion\_extra)

int howmany;

// How many transitions are there. [0,in\_use) index of TransPair \*data, it is is managed in add\_transition()

int in\_use;

// A dynamically allocated array of(CharRange, State) pairs(transitions). V × Q

// struct TransPair表示:T(a) = { q | a in V，q in Q }

TransPair \*data;

// For efficiency of the expansion helper function.

enum { expansion\_extra = 5 };

};

class DTrans :protected TransImpl

{

public:

// Constructors, destructors, operator=:

// By default, don't introduce any transitions.

inline DTrans();

// Copy constructor does a dynamic memory copy.

inline DTrans(const DTrans& r);

inline const DTrans& operator=(const DTrans& r);

// Normal member functions :

// Map a char to the unique next state.

State operator[](const char a) const;

// Map a CharRange to the corresponding State

// assuming that the CharRange is entirely contained in the label of a transition.

State range\_transition(const CharRange a) const;

// What are the labels of transitions out of \*this.

inline CRSet out\_labels() const;

// What are all transition labels into StateSet r?

inline CRSet labels\_into(const StateSet& r) const;

// Is there a valid out-transition on a?

int valid\_out\_transition(const CharRange a) const;

// What is the range(States) of this map?

// return a StateSet with domain dom.

StateSet range(int dom) const;

// Special member functions:

// Recycle this entire structure.

inline void reincarnate();

// Create a new out-transition.

inline DTrans& add\_transition(const CharRange a, const State q);

friend std::ostream& operator<<(std::ostream& os, const DTrans& r);

inline int class\_invariant() const;

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Map State's to class T. Used to implement transitions(TransRel and DTransRel)

// and State relations(StateRel and StateEqRel).

StateTo<T> 描述state to T(eg. StateSet)的映射(map)关系。

T \*data; 内部使用data[state]表示这种映射关系。

set\_domain(const int r)设置动态数组data的大小，表示存储这种关系的容量,对\*data分配了存储空间。

int howmany; 表示动态数组data大小，包含扩充部分(expansion\_extra常量)

int in\_use; 表示实际使用的data大小。

以下两个函数都是返回state r to T的映射(map),前者返回Not const T&，用于设置；后者返回const T&，用于查询。

二者均没有给data[r]赋值。留给继承类完成。

T& map(const State r); retun data[r]; Not const，用于设置；

const T& lookup(const State r) return data[r]，用于查询。在派生类中给data[r]赋值后才能使用。

StateTo<T>& StateTo<T>::disjointing\_union(const StateTo<T>& r)

合并this和r ==> this，保证不重叠, this.domain += r.domain;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<class T>

class StateTo

{

// Cannot be protected because the function in DFAseed.h(construct\_components) needs access.

public:

// Constructors, destructors, operator=:

// Default is to not map anything.

StateTo();

// Copying can be costly. Use-counts could make this cheaper.

StateTo(const StateTo<T>& r);

// Assume delete[]0 is okay.

virtual ~StateTo();

// Use counts could make this cheaper.

const StateTo<T>& operator=(const StateTo<T>& r);

// First, a const lookup operator.

inline const T& lookup(const State r) const;

// Used to associate a State and a T in the mapping. Note: not const.

inline T& map(const State r);

// Some domain members:

// How many States can \*this map ?

inline int domain() const;

// Set a new domain.

void set\_domain(const int r);

// Note the reincarnate() doesn't reincarnate the T's.

inline void reincarnate();

// Allow two mappings to be combined; the domain of \*this remains the same, while the

// domain of r is renamed to not clash with the domain of \*this.

StateTo<T>& disjointing\_union(const StateTo<T>& r);

//Some extras:

// 必须类内定义，使其派生类可以使用inline，在.h文件的类外定义std::ostream& operator<<()

friend std::ostream& operator<<(std::ostream& os, const StateTo<T>& r)

{

assert(r.class\_invariant());

for (int i = 0; i < r.in\_use; i++)

os << i << "->" << r.data[i] << std::endl;

return(os);

}

// Assert that everything's okay.

inline int class\_invariant() const;

private:

// Represent the map as a dynamically allocated array of T's (include expansion\_extra).

// indicating the size of the array

int howmany;

// indicating how many of the dynamically allocated array of T's elements are in use.

// [0,domain) -> index of T \*data;

int in\_use;

// dynamically allocated array of T's (include expansion\_extra).

T \*data;

// When the array is grown by a certain amount, it also grows by an extra

// buffer amount for efficiency.

enum { expansion\_extra = 5 };

};