Handling of parameterized data types in LEDA

Christian Lennerz and Sven Thiel

April 28, 1997

Abstract

In this paper we discribe how parametrized data types in LEDA deal with their parameter data types. We mainly address people who want to implement new parameterized data types for LEDA. We start with a brief introduction to the *GenPtr*-concept, the main concept underlying the interactions between parameterized data types and the parameter data types. Then we explain the LEDA macros, which allow uniform access to objects of a parameter data type. Finally we describe the implementation of the functions and the macros in the file param_types.h.

Contents

1 Introduction

In LEDA parameterized data types store copies of values of arbitrary types T with the aid of a pointer p of type *GenPtr*, i.e. *void*$*$ (cf. global.h). To a certain extend one can think of p as a pointer to an object of type T.

LEDA uses p in two different ways depending on the size of T:

1. $sizeof(T) > sizeof(GenPtr)$
   In this case T is called a "big" or "large" type and p is treated as a pointer to an object of type T. The object, to which p points, is stored in the dynamic memory.
2. $sizeof(T) \leq sizeof(GenPtr)$
   In this case T is called a "small" type. Then p is not used as a pointer, which means it does not store the address of an object of type T, but it stores the value itself. Thus memory is saved and unnecessary pointer indirection is avoided.

Apart from saving memory there are some reasons why the *GenPtr*-concept has been implemented in LEDA:

- The main reason is that this technique prevents code bloathing. In LEDA many parameterized data types are implemented in two layers. The high-level layer is templatized with a parameter type T, which has the effect that for all different parameter types new code is generated. The low-level layer operates on *GenPtr*s so that the code for this layer is generated only once, even if the parameterized data type is used with different parameter types T. e.g.: The parameterized data type `list<T>` in LEDA is based on the data type `dlist` whose items are referenced via *GenPtr*s.
- Another reason is a historical one. *GenPtr*s were helpful when templates did not exist.

LEDA provides some macros which allow uniform access to *GenPtr*s like p, no matter if p points to an object x of type T or if the object x is stored directly in p. In both cases the object x shall be called the "object stored at p" in the future.

2 The LEDA macros

We now list the LEDA macros, which can be found in param_types.h. We describe their function and give some examples of their application in LEDA. If it is convenient we will mention how C++ pointer operations can be replaced by using LEDA macros. In these parts of our description we do not want to describe how the macros are implemented and we do not distinguish between small and large types, but we want to show how non-LEDA programms that are using pointers to objects of type T could be adapted to the *GenPtr*-concept of LEDA.
Let us introduce some notation: T is always the name of a type, x is an object of type T and p and q are *GenPtr*s. p' and q' are of type $T*$.

`T& LEDA_ACCESS(T, GenPtr& p)`

A reference to the object stored at p is returned.

Application in LEDA:

$\langle list.h \rangle \equiv$
```
E& list::operator[](list_item it)
{ return LEDA_ACCESS(E,dlist::entry(it)); }
```

The *GenPtr*& that *dlist::entry* provides (cf. dlist.h) is used as input for the `LEDA_ACCESS` macro, so that a reference to the object stored at the *GenPtr* can be returned.

Replacement of C++ pointer-operations:

```
x = *p';        ⟶   x = LEDA_ACCESS(T,p);
*p' = x;        ⟶   LEDA_ACCESS(T,p) = x;
```

Note that the *GenPtr*-concept is tricky concerning assignment of small types as we do not really handle pointers but storage places itself. The following code chunk

```
GenPtr A[5];
LEDA_ACCESS(char,A[2]) = 'a';
GenPtr q = A[2];
LEDA_ACCESS(char,q) = 'b';
cout << LEDA_ACCESS(char,A[2]);
```

does not have the same effect as

```
char* A[5];
*A[2] = 'a';
char* q = A[2];
*q = 'b';
cout << *A[2];
```

The output of the first sequence would be a, whereas the output of the second would be b. (By declaring q as a *GenPtr&* one could make the first sequence semantically equal to the second.) The reason for the different behaviour of the two sequences is that *char* is a small type. Thus in the first sequence q does not store a pointer to $A[2]$, but it stores a copy of the value of $A[2]$ so that the access in the forth line only changes that copy and not the original object.

`const T& LEDA_CONST_ACCESS(T, const GenPtr& p)`

A constant reference to the object stored at p is returned.

Application in LEDA: *segment_tree* :: *x0* in seg_tree.h

`void LEDA_CREATE(T, GenPtr& p)`

A new object of type T is created by a call to the standard constructor and stored at p.

Application in LEDA:

⟨*array.h*⟩≡

```
void array::init_entry(GenPtr& x) const
{ LEDA_CREATE(E,x); }
```

The `LEDA_CREATE` macro is used to initialize the object stored at the *GenPtr&* x with the value created by the default constructor of the type E. This is not only the case for this occurrence of the macro but for all occurrences in the current LEDA code.

Replacement of C++ pointer-operations:

```
p' = new T;     ⟶   LEDA_CREATE(T,p);
```

`void LEDA_COPY(T, GenPtr& p)`

Makes a copy of the object stored at p and stores the copy at p. After a call of this macro write accesses to the object stored at p only change the copy and not the original object.

Application in LEDA:

$\langle$*prio.h*$\rangle\equiv$
```
void priority_queue::copy_key(GenPtr& x) const
{ LEDA_COPY(I,x); }
```

The `LEDA_COPY` macro appears in functions called *copy_entry*, *copy_key*, *copy_inf*, ... usually exactly in the way it is used in the example above. We now discuss an example which uses the *copy_key* function of a priority queue:

$\langle$*list_pq.h*$\rangle\equiv$
```
inline void list_pq::decrease_key(list_pq_item it, GenPtr x)
{
  copy_key(x);
  clear_key(it->key);
  it->key = x;
}
```

First the object stored at x is copied, then the old key of item *it* is cleared and finally the copied(!) object stored at x is made the new key of the item. This is necessary if the parameter data type E of the priority queue, i.e. the type of the object stored at x, is a large type. Then x is used as a pointer to an object of type E. Hence if no copy was made and the object stored at x was destroyed outside the function body then $it \rightarrow key$ would point to some undefined data.
Note that the *decrease_key* function is not called with a *GenPtr&*, because otherwise the function would not work correctly if E was a large type. When called for a large type the `LEDA_COPY` macro changes the address that x points to. So if a *GenPtr&* was given to the function, the address of the original object stored at x would be lost outside the function.

Replacement of C++ pointer-operations:
`p' = new T(*p');` $\longrightarrow$ `LEDA_COPY(T,p);`

`int LEDA_INT_TYPE(T)`

The macro is used to determine whether T is an integer type for LEDA. By default the macro returns 0. If T is *int* or if T is *long* and $sizeof(int) == sizeof(long)$ then 1 is returned.

Application in LEDA:
All the *int_type* functions of the LEDA data types are of the same form as *array*:: *int_type*:

$\langle$*array.h*$\rangle+\equiv$
```
int array::int_type() const
{ return LEDA_INT_TYPE(E); }
```

The following chunk shows how the *sort* function of the data type *array* uses the *int_type* function in order to decide whether to use an integer-optimized sort-routine or the default one.

$\langle$*array.h*$\rangle+\equiv$
```
void array::sort()
{
  cmp_ptr = compare;
  if (int_type()) gen_array::int_sort(low(),high());
  else gen_array::sort(low(),high());
}
```

char* LEDA_TYPE_NAME(T)

The name of the type T is returned, if known. By default the macro returns "unknown". For elementary types T the names "char", "short", ... are returned.

Application in LEDA: *GRAPH*::*node_type* in graph.h

GenPtr LEDA_CAST(const T& x)

The macro returns a *GenPtr* with the property that x is the object stored at the *GenPtr*. To be more precise:

- If $sizeof(T) > sizeof(GenPtr)$ then the address of x is simply casted to *GenPtr*.
- If $sizeof(T) < sizeof(GenPtr)$ then the value of x is copied into the *GenPtr* by calling $T.operator =$.
- If $sizeof(T) == sizeof(GenPtr)$ then the value of x is copied bitwise into the *GenPtr*.

void LEDA_CLEAR(T, GenPtr& p)

The object stored at p is deleted. For large types the destructor is called.

Application in LEDA:

⟨*slist.h*⟩≡
```
E slist::pop()
{
  GenPtr x = SLIST::pop();
  E y = LEDA_ACCESS(E,x);
  LEDA_CLEAR(E,x);
  return y;
}
```

First the *GenPtr*-entry from an SLIST is popped, then a copy of the object stored at x is stored in y, after that the object stored at x is destroyed and finally y is returned. This example also shows the two layer concept of LEDA mentioned in the introduction. *SLIST* is the low-level layer and operates on *GenPtr*s. The corresponding high-level layer is the parameterized data type *slist<E>* which operates on objects of type E. The interface between the two layers is provided by the LEDA macros.

Replacement of C++ pointer-operations:

`delete p';` $\longrightarrow$ `LEDA_CLEAR(T,p);`

`int LEDA_COMPARE(T, const GenPtr& p, const GenPtr& q)`

The objects stored at p and q are compared with the aid of the function *int compare* (*const*& T, *const*& T), which must exist for T. If T.*operator* $<$ and T.*operator* $>$ are implemented then the *compare* function can be defined by *DEFINE_COMPARE*(T).

Application in LEDA: *dictionary*::*cmp* in dictionary.h

Replacement of C++ pointer-operations:
`res=compare(*p',*q');` ⟶ `res=LEDA_COMPARE(T,p,q);`

`int LEDA_HASH(T, const GenPtr& p)`

A hash value for the object stored at p is returned with the aid of the function *int Hash*(*const* T&), which must exist for T. For the elementary types the *Hash* functions are already implemented.

Application in LEDA: *_dictionary*::*hash_fct* in _dictionary.h

Replacement of C++ pointer-operations:
`h=hash(*p');` ⟶ `h=LEDA_HASH(T,p);`

`void LEDA_PRINT(T, const GenPtr& p, ostream& out)`

The object stored at p is written to the stream *out*. The macro calls the function *Print*(*const* T&, *ostream*&); if no special version of this function for the type T has been implemented then a function template, which is declared in param_types.h, is used to print objects of type T by calling T.*operator* $\ll$.

Application in LEDA: *array*::*print_el* in array.h

Replacement of C++ pointer-operations:
`out << *p';` ⟶ `LEDA_PRINT(T,p,out);`

`void LEDA_READ(T, GenPtr& p, istream& in)`

An object of type T is read from the stream *in* with the aid of *Read*(T&, *istream*&) or T::*operator* $\gg$ and stored at p (cf. `LEDA_PRINT`). Similarly as with `LEDA_ACCESS` it is important that the macro is called with a reference parameter p.

Application in LEDA: *array*::*read_el* in array.h

Replacement of C++ pointer-operations:
`in >> *p';` ⟶ `LEDA_READ(T,p,in);`

`LEDA_TYPE_PARAMETER(T)`

In some respects this macro does not fit in the lists of the other macros above, because it is not meant to be used for programming a parameterized data type. It is only needed if the compiler does not support function templates. Then the user has to call this macro for every non-elementary type T before he can use T as a parameter for a parameterized data type. The call of the macro explicitly declares all the functions necessary for the macros above to work.

3 The Implementation

We define some functions that allow us to operate on data of parameterized data types. Because of various implementations of C++-Compilers, we decided to not present every special treatment and to assume that the compiler supports template functions and function templates. If this is not the case, see the actual file param_types.h for further information.

⟨*param_types.h*⟩≡

```
#ifndef PARAM_TYPE_H
#define PARAM_TYPE_H
```

⟨*define functions to copy data of a parameterized data type*⟩

⟨*define functions to create data of a parameterized data type*⟩

⟨*define functions to destroy data of a parameterized data type*⟩

⟨*define functions to access data of a parameterized data type*⟩

⟨*define functions to compare data of a parameterized data type*⟩

⟨*define functions to support various operations*⟩

⟨*define macros*⟩

```
#endif
```

3.1 Functions to copy data of a parameterized data type

First we introduce the base function *leda_copy* that copies the value of a constref object x of type T and returns the copied object itself or a pointer to the copied object. The GenPtr construction allows us to return in both cases an object of type *GenPtr*.
There are also two cases to consider:

1. *T is a small data type*
   We copy the value of object x into a *GenPtr* p and return p.
2. *T is a big data type*
   We make a copy of x in the dynamic memory using the new operator and return the pointer to this copy as a *GenPtr*.

⟨*define functions to copy data of a parameterized data type*⟩≡

⟨*define COPY_INTO_WORD*⟩

```
template<class T>
inline GenPtr leda_copy(const T& x)
{ GenPtr p=0;
  if (sizeof(T) <= sizeof(GenPtr)) COPY_INTO_WORD(T,x,p);
  if (sizeof(T) >  sizeof(GenPtr)) p = new T(x);
  return p;
 }
```

To copy a value of a type T into a *GenPtr* we need an in-place new operator. If the compiler supports in-place new, we define our own inline version, because the operator is not always declared inline.
But if in-place new is not supported, we cast the address of the *GenPtr* p to a pointer to T.

⟨*define COPY_INTO_WORD*⟩≡

```
#if defined(__INPLACE_NEW__)
inline void Avoid_Unused_Warning(GenPtr) {}
inline void* operator new(size_t, void* where, int dummy) { return where; }
#define COPY_INTO_WORD(T,x,p) Avoid_Unused_Warning(new((T*)&p, 0) T(x))

#else
#define COPY_INTO_WORD(T,x,p) *(T*)&p = x
#endif
```

3.2 Functions to create data of a parameterized data type

First we introduce the base function *leda_create* that provides memory space to hold an object of type T and returns an instance or a pointer to an allocated memory range, in which an object of type T can be stored. The *GenPtr* construction allows us to return in both cases an object of type *GenPtr*.
There are also two cases to consider:

1. *T is a small data type*
   We cast the address of a *GenPtr* p into a pointer of type T in order to store an object of type T directly in p
2. *T is a big data type*
   We allocate enough memory space to hold an object of type T using the new operator and return the pointer to the allocated memory range as a *GenPtr*.

⟨*define functions to create data of a parameterized data type*⟩≡

⟨*define CONSTRUCT_WORD*⟩

```
template <class T>
inline GenPtr leda_create(const T*)
{ GenPtr p=0;
  if (sizeof(T) <= sizeof(GenPtr)) CONSTRUCT_WORD(T,p);
  if (sizeof(T) >  sizeof(GenPtr)) p = new T;
  return p;
 }
```

We cast the address of a *GenPtr* p into a pointer of type T, so that we can allocate enough memory space to store an object of type T directly in p, using the in-place new operator. If the compiler doesn't support the in-place new operator, we create a dummy object using *leda_copy*.

⟨*define CONSTRUCT_WORD*⟩≡

```
#if defined(__INPLACE_NEW__)
#define CONSTRUCT_WORD(T,p) Avoid_Unused_Warning(new((T*)&p, 0) T)

#else
#define CONSTRUCT_WORD(T,p) { T x; COPY_INTO_WORD(T,x,p); }
#endif
```

3.3 Functions to destroy data of a parameterized data type

First we introduce the base function *leda_clear* that deletes an object of type T. The GenPtr construction allows us to handle both cases, small or big type, by the *leda_clear* function.

1. *T is a small data type*
   If the compiler doesn't expect an explicit destruction, there is nothing to do, because the object will be deleted automatically when its scope is left.
2. *T is a big data type*
   The allocated memory space has to be given back to the system. For we have created the object by the new operator, we have to delete it by using the delete operator.

⟨*define functions to destroy data of a parameterized data type*⟩≡

⟨*define DESTROY_WORD*⟩

```
template <class T>
inline void leda_clear(T& x)
{ T* p = &x;
  if (sizeof(T) <= sizeof(GenPtr)) DESTROY_WORD(T,p);
  if (sizeof(T) >  sizeof(GenPtr)) delete p;
 }
```

If T is a small type and the compiler doesn't support explicit destruction, we have to call the class destructor. If not, we have nothing to do and call therefore a dummy function.

⟨*define DESTROY_WORD*⟩≡

```
#if defined(__EXPLICIT_DESTRUCTION__)
#define DESTROY_WORD(T,p) p->~T()

#else
inline void leda_explicit_destruction(GenPtr) {}
#define DESTROY_WORD(T,p) leda_explicit_destruction((T*) p);
#endif
```

3.4 Functions to access data of a parameterized data type

First, we introduce a function *leda_cast* that takes a constref object of type T as argument and returns an object of type *GenPtr*. If we have converted the object of type T to a *GenPtr*, we could pass it as function parameter to functions *leda_access* and *leda_const_access* that we will discuss below. Differing from our proceeding to distinguish only two cases, we will treat the case sizeof(T) = sizeof(*GenPtr*) in a different way.

1. $sizeof(T) < sizeof(GenPtr)$
   When *GenPtr* has a large word size — for example on 64-bit systems — and we store a small data type in it, a copy in the size of the *GenPtr* would not be efficient. So we cast the address of the *GenPtr* p into a pointer to T, and do the copy of smaller volume.
2. $sizeof(T) = sizeof(GenPtr)$
   As in this case the casting of case 1 is more expensive than the cast from T to *GenPtr*, we do the latter.
3. $sizeof(T) > sizeof(GenPtr)$
   We cast the address of object x into *GenPtr* and assign it to an object p of type *GenPtr*.

⟨*define functions to access data of a parameterized data type*⟩≡

```
template <class T>
inline GenPtr leda_cast(const T& x)
{ GenPtr p=0;
  if (sizeof(T) == sizeof(GenPtr)) p = *(GenPtr*)&x;
  if (sizeof(t) < sizeof(GenPtr)) *(T*)&p = x;
  if (sizeof(T) > sizeof(GenPtr)) p = (GenPtr)&x;
  return p;
 }
```

Now we can define the principal functions to access data of a parameterized data type. The names of these functions are *leda_access* and *leda_const_access*. As we have seen above, we hide different treatment of data types under the *GenPtr* layer, on which we don't have to think about differences between the data types. It should be obvious that from time to time we have to bring the actual data type back to the programming surface. This is exactly what the access functions do. The arguments of the functions are a pointer to a constant object of type T and a constref object of type *GenPtr*. The first argument indicates which data type is accessed and the second argument determines the object that should be accessed. The functions return a reference to the object, that is determined by the *GenPtr*. *leda_const_access* returns a const object of type T.

1. *T is a small data type*
   In this case the object is stored directly in the *GenPtr* p, so that we only have to cast the address of p into a pointer to T and return the dereferenced object.
2. *T is a big data type*
   In this case we store a pointer to the object in the *GenPtr* p, so that we only have to cast p into a pointer to T and return the dereferenced object.

⟨*define functions to access data of a parameterized data type*⟩+≡

```
template <class T>
inline T& leda_access(const T*, const GenPtr& p)
{ if (sizeof(T) <= sizeof(GenPtr))
     return *(T*)&p;
  else
     return *(T*)p;
 }

template <class T>
inline const T& leda_const_access(const T*, const GenPtr& p)
{ if (sizeof(T) <= sizeof(GenPtr))
     return *(T*)&p;
  else
     return *(T*)p;
 }
```

3.5 Functions to compare data of a parameterized data type

Many parameterized data types require a linear order on the plugged-in data-types. That's why a compare function define macro is declared for an arbitrary type T. To define a linear order on our data type we only have to define the *operator* $<$ and *operator* $>$. The macro DEFINE_COMPARE(T) includes the definition of the compare function. The compare function takes to constref objects x and y of type T and returns:

$$compare(x,y) = \begin{cases} -1 & : \quad x < y \\ 0 & : \quad x = y \\ +1 & : \quad x > y \end{cases}$$

For C++provides *operator* $<$ and *operator* $>$ on some elementary data types, we are able to define the compare function yet.

⟨*define functions to compare data of a parameterized data type*⟩≡

```
#define DEFINE_COMPARE(T)
inline int compare(const T& x, const T& y)
{ if (x < y) return -1;
  else if (x > y) return 1;
  else return 0;
 }

DEFINE_COMPARE(char)
DEFINE_COMPARE(int)
DEFINE_COMPARE(unsigned int)
DEFINE_COMPARE(long)
DEFINE_COMPARE(unsigned long)
DEFINE_COMPARE(float)
DEFINE_COMPARE(double)
```

3.6 Function to support various operations

3.6.1 Input/output functions for data of parameterized data types

Sometimes we want to put data of a prameterized data type on an input/output stream. That's why we provide the functions *Print* and *Read*. To use this functions, you only have to make sure that *operator* $\ll$ and *operator* $\gg$ are defined for type T used as data in the parameterized data type.

⟨*define functions to support various operations*⟩≡

```
template <class T>
inline void Print(const T& x, ostream& out) { out << x; }

template <class T>
inline void Read(T& x, istream& in) { in >> x; }
```

3.6.2 Functions to support hashing

To support hashing we define functions that convert any integer compatible object to a int return value.

⟨*define functions to support various operations*⟩+≡

```
inline int Hash(const GenPtr& x) { return int(long(x)); }
inline int Hash(const char&   x) { return int(x); }
inline int Hash(const short&  x) { return int(x); }
inline int Hash(const int&    x) { return int(x); }
inline int Hash(const long&   x) { return int(x); }
inline int Hash(const double& x) { return int(x); }
inline int Hash(const float&  x) { return int(x); }
```

3.6.3 Functions to get information on data types

During the programming phase the template concept doesn't allow us to determine the data type our functions are operating on. But sometimes we need this information. Therefore we want to provide functions that return the type name of an object as a string. It should be obvious that we can provide this function only for the elementary types and that it is left to the programmer to define this function for the data type he has written.

⟨*define functions to support various operations*⟩+≡

```
inline char* leda_tname(const char*  ) { return "char";    }
inline char* leda_tname(const short* ) { return "short";   }
inline char* leda_tname(const int*   ) { return "int";     }
inline char* leda_tname(const long*  ) { return "long";    }
inline char* leda_tname(const float* ) { return "float";   }
inline char* leda_tname(const double*) { return "double";  }
inline char* leda_tname(const T*     ) { return "unknown"; }
```

Sometimes we want to know if a data type T can be treated as *int*.

⟨*define functions to support various operations*⟩+≡

```
template <class T>
inline int leda_itype(const T*   ) { return 0; }
inline int leda_itype(const int* ) { return 1; }
inline int leda_itype(const long*) { return sizeof(long) == sizeof(int); }
```

4 The Mapping: Macros + Functions

As we have discussed above, we provide macros for frequently used function calls. To make function arguments more comprehensive, we define two pointer:

1. MKNULLPTR(T) provides a nil pointer for type T
2. ACCESSPTR(p) provides a pointer p

⟨*define macros*⟩≡

```
#define MKNULLPTR(T)           (T*)0
#define ACCESSPTR(p)           p
#define LEDA_ACCESS(T,p)       leda_access(MKNULLPTR(T),ACCESSPTR(p))
#define LEDA_CONST_ACCESS(T,P) leda_const_access(MKNULLPTR(T),ACCESSPTR(p))
#define LEDA_CREATE(T,x)       x=leda_create(MKNULLPTR(T))
#define LEDA_COPY(T,x)         x=leda_copy(LEDA_CONST_ACCESS(T,x))
#define LEDA_INT_TYPE(T)       leda_itype(MKNULLPTR(T))
#define LEDA_TYPE_NAME(T)      leda_tname(MKNULLPTR(T))
#define LEDA_CAST(x)           leda_cast(x)
#define LEDA_CLEAR(T,x)        leda_clear(LEDA_ACCESS(T,x))
#define LEDA_COMPARE(T,x,y)    compare(LEDA_CONST_ACCESS(T,x),LEDA_CONST_ACCESS(T,y))
#define LEDA_HASH(T,x)         Hash(LEDA_CONST_ACCESS(T,x))
#define LEDA_PRINT(T,x,out)    Print(LEDA_CONST_ACCESS(T,x),(ostream&)out)
#define LEDA_READ(T,x,in)      Read(LEDA_ACCESS(T,x),(istream&)in)
```