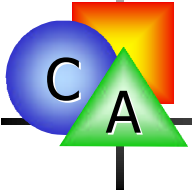


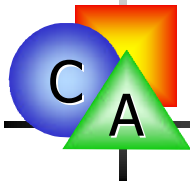
Storing Properties in Grouped Tagged Tuples



Eberhard Karls Universität Tübingen
Computeralgebra, Wilhelm-Schickard-Institut

Roland J. Weiss
Volker Simonis

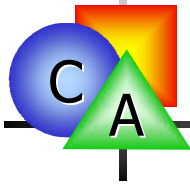
PSI 2003 – July 9th 2003



Overview

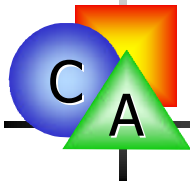
- Motivation and Introduction
- Named Objects Revisited
- Properties and More
- Performance
- Conclusions and Future Work

Motivation and Introduction



Motivation

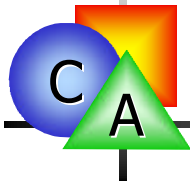
- Cartesian product types:
Basic building block for composite types
 - Pascal, Ada: records
 - C/C++: structs
 - ML, Haskell, ...: tuple types
- Topic: C++ support for tuples and related special purpose constructs (properties)



Tuples in C++

- No built-in tuple type
- Classes considered too heavy-weight/tedious for data passing
 - Namespace pollution
- Jakko Järvi: Boost Tuple Library
 - Access to tuple elements by index or type
 - Handles multiple return values, combine parameters
- Emily Winch
 - Access tuple elements by name
 - Operations on elements synthesized from formal description
 - Constructor, assignment operator, multiple value manipulations, ...

➔ C++ Template Meta-Programming



C++ Template Meta-Programming

- C++ template mechanism takes place at compile time (mandated by C++ standard)
- Expressiveness
 - Values: compile time constants & types
 - Conditions:
 - Pattern matching during partial template specialization for types
 - ?-operator for integral values
 - Loops: recursive instantiation

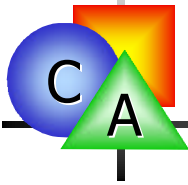


Typelists

- Compile time data structure that stores types
- Well suited to control code generation
 - Visitor pattern: visitation class hierarchy
 - Object factories: product types

```
typedef Loki::TypeList<
    unsigned char, unsigned short,
    unsigned int, unsigned long
>::type unsigned_types;
// Calculate length of type list.
const int l = Loki::TL::Length<unsigned_types>::value;
```

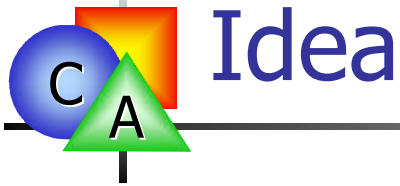
Named Objects Revisited



Source of Inconsistency

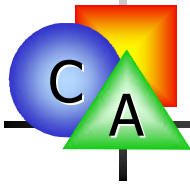
- Problem in Winch's object creation
 - Name type: used to access element
 - Implementation type: actually stored data type
 - Name type and Implementation type pairing

```
// Name types.  
struct myBigClass {}; struct age {};  
struct myDatabase {};  
// Name and implementation type pairing in type constr.  
typedef makeVarlistType3<  
    BigClass*, myBigClass, int, age,  
    Database&, myDatabase  
>::list VarlistType;
```



- Permanently associate name and implementation types: OK, use nested type definitions
- Pass typelist of name types with nested implementations types to type constructor of Tagged_Tuple

```
// Name and implementation type pairing.  
struct myBigClass { typedef BigClass* type; };  
struct age { typedef int type; };  
struct myDatabase { typedef Database& type; };  
// Type constructor.  
typedef Tagged_Tuple<  
    TypeList<myBigClass, age, myDatabase>::type  
> PropType;
```



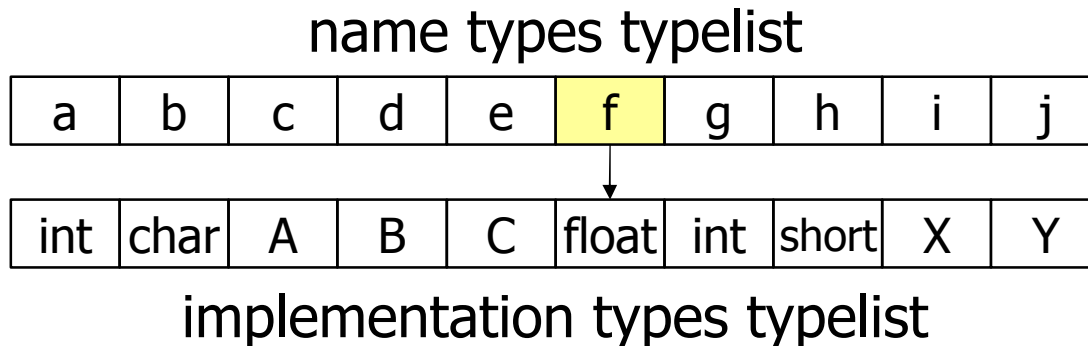
Extract Implementation Types

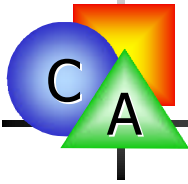
- How to extract implementation types?
→ needed for tagged tuple's internal data
 - Use C++ template meta-program:
`ExtractTypes: TL -> TL`
 - `ExtractTypes` generates new typelist that holds implementation types
 - New typelist can be passed to tuple constructor for internal data



Element Access

- Complication:
 - Element access by name type
 - Implementation tuple knows only impl. types
- Name type and implementation type located at same position in type lists
→ element index into implementation tuple





Element Access

```
// Access element with tag PropT.
template <class PropT>
typename return_t<PropT, tuple_type, TL>::type at() {
    return Loki::Field<
        Loki::TL::IndexOf<props_t1, PropT>::value
    >(m_props);
}
// Usage:
std::string name = data.at<Name>();
```

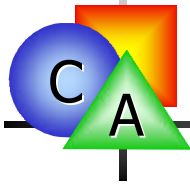
- Same functionality as Winch's approach
- Name and implementation type pairing fixed with typelists and template meta-programming

Properties and More



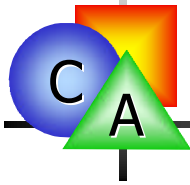
Properties

- Section of internal data describing an object's state that are...
- Publicly exposed through standardized access methods
- Examples: color, name, ...
- Discussed Tagged_Tuple type typical linear property container



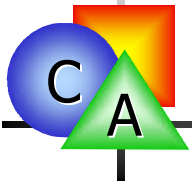
Well Known: Java Properties

- Basic component technology in Java
- Tools inspect code for standard naming schemes (e.g. get-/set-methods)
- Programmer has to provide access methods for every property
- Tagged_Tuple: access methods generated for all listed properties



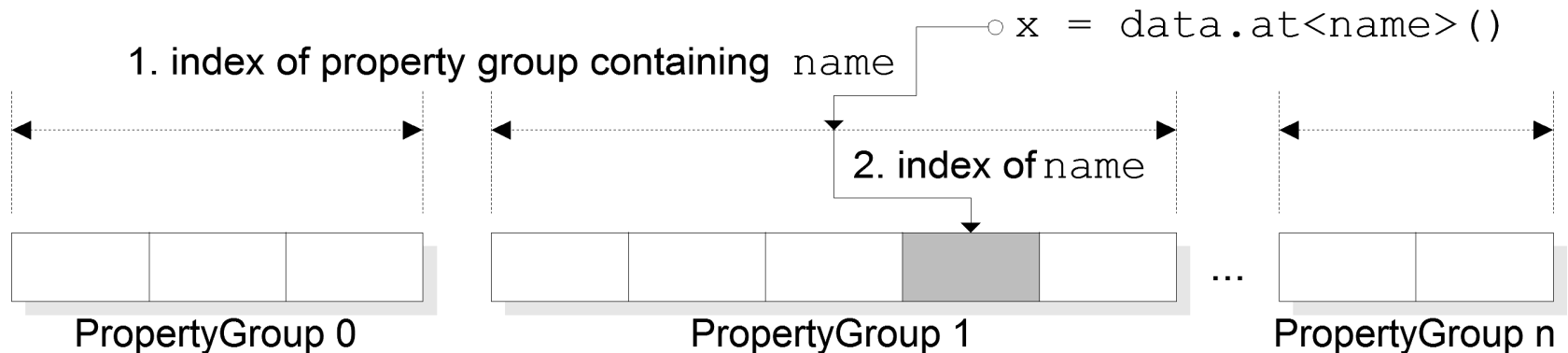
Groups of Properties

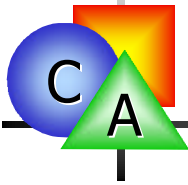
- Related properties commonly reused in several classes/components
 - Debug information, colors, ...
 - HTML/XML attributes, nodes in symbol table, ...
- Class template's `Named_Properties` requirements
 1. Type-safe element access
 2. Related properties grouped
 3. Combine groups of properties
 4. Flat access to properties by name
- R1, R2: `Tagged_Tuple`
- R3: Tuple of `Tagged_Tuple(s)`
- R4: Extensive meta-programming



Flat Element Access

- Two actions
 - Determine named property's type
 - Locate named property's data
- Both two-level processes
 1. Locate tagged tuple containing name type
 2. Access type/data at correct position in located tagged tuple



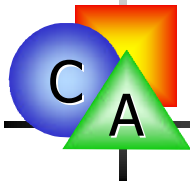


Flat Element Access

```
// Access element with tag PropT.  
template <class PropT>  
typename return_t<PropT, tuple_type, TL>::type at() {  
    return Loki::Field< IndexOfNP<TL, PropT>::value >  
        (m_props).template at<PropT>();  
}
```

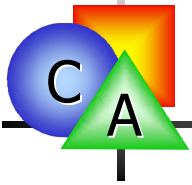
- 3-level access possible, name clashes likely
- Error handling:
 - PropT tag not present: compile time error
 - PropT tag present multiple times: returns first found

Performance



Performance

- How do generated data structures compare to handwritten data structures w.r.t. ...
 1. Memory efficiency?
 2. Runtime performance?
- Artificial benchmarking application
 1. Create homogeneous container
 2. Read and write to all properties present in element type



Element Type

Handwritten

```
struct debug_props {
    size_t debug_line_prop;
    size_t debug_column_prop;
    string debug_name_prop;
    string debug_source_prop;
    // Access operations.
    size_t& at_dlp() { return debug_line_prop; }
    size_t& at_dcp() { return debug_column_prop; }
    string& at_dnp() { return debug_name_prop; }
    string& at_dsp() { return debug_source_prop; }
};
struct id_props {
    string id_prop;
    double rate_prop;
    // Access operations.
    string& at_ip() { return id_prop; }
    double& at_rp() { return rate_prop; }
};
// Property group.
struct test_props {
    debug_props p1;
    id_props p2;
};
```

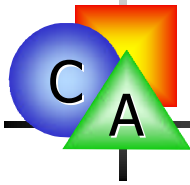
Generated

```
struct debug_line_prop { typedef size_t type; };
struct debug_column_prop { typedef size_t type; };
struct debug_name_prop { typedef string type; };
struct debug_source_prop { typedef string type; };
// Associated type list.
typedef Loki::TypeList<
    debug_line_prop, debug_column_prop,
    debug_name_prop, debug_source_prop
>::type debug_prop_t1;

struct id_prop { typedef string type; };
struct id_rate_prop { typedef double type; };
// Associated type list.
typedef Loki::TypeList<id_prop, id_rate_prop>::type
    id_prop_t1;

// Single properties.
typedef Tagged_Tuple<debug_prop_t1> debug_np;
typedef Tagged_Tuple<id_prop_t1> id_np;

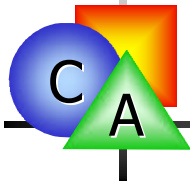
// Property group.
typedef Named_Properties<
    Loki::TypeList<debug_np, id_np>::type> test_nps;
```



Memory Efficiency

	Handwritten	Generated
g++ 3.2	32	32
Metrowerks 8.3	56	56
Visual Studio 2003	104	112

- Object sizes differ because of string implementations:
GCC (4 Bytes) – MW (12 Bytes) – VS (28 bytes)

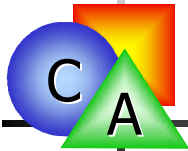


Runtime Performance

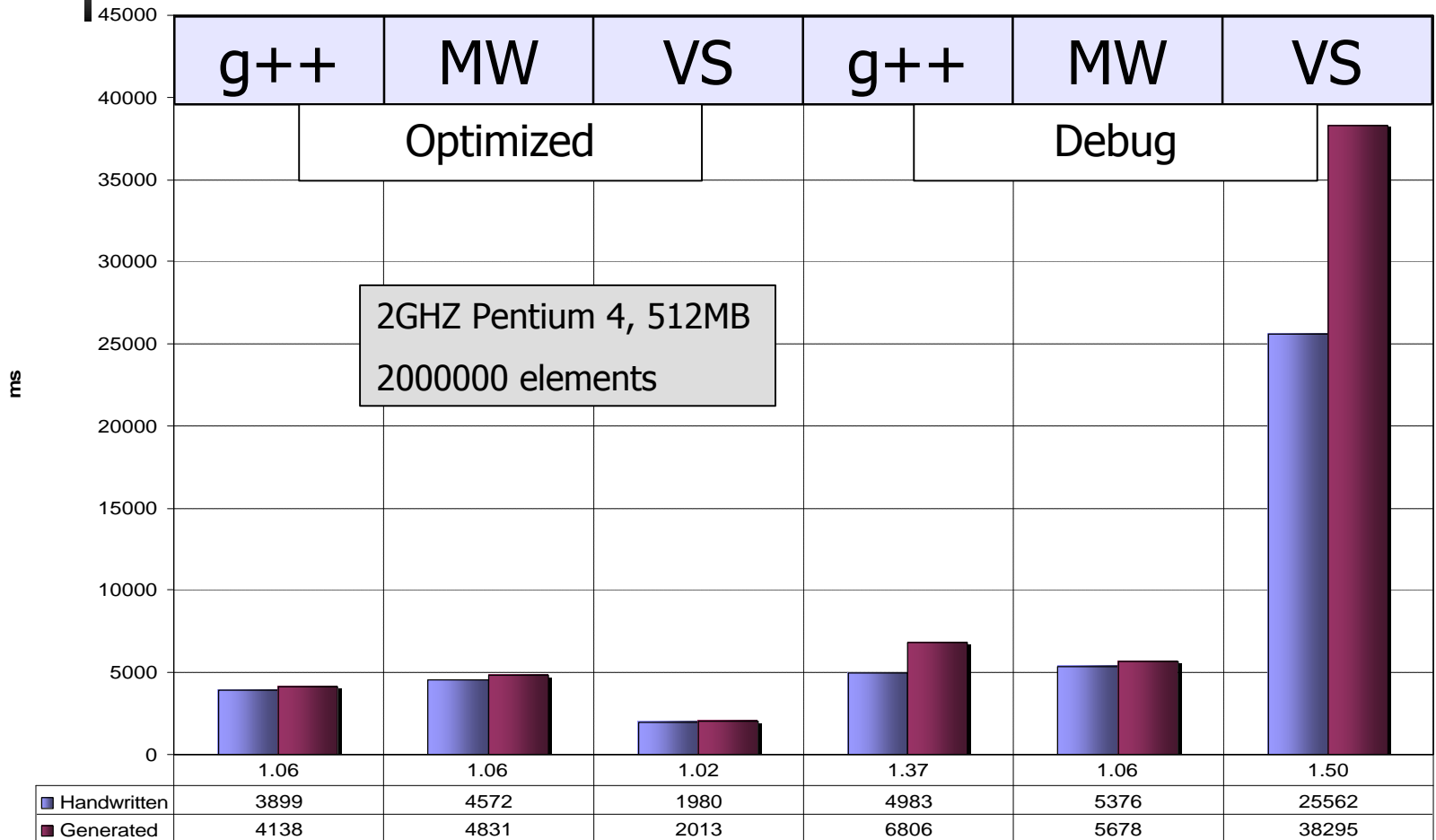
- Measure abstraction penalty

$$AP = \frac{\textit{runtime}(\textit{abstract version})}{\textit{runtime}(\textit{low level version})}$$

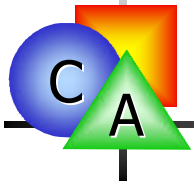
- Abstract version: generated properties group
- Low level version: handwritten properties group



Runtime Performance: Results

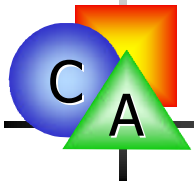


Conclusions and Future Work



Conclusions

- `Named_Properties` provide convenient and type-safe property storage
- Template meta-programming allows efficient data structure generation (BGL, MTL, Blitz++)
- C++ compiler technology now mature (5 years after standard's publication)



Future Work

- Add bounds/constraints to named properties like JavaBeans
- Enable programming languages with intended inherent code generation facility
- C# Attributes: provide meta-data in code