High Level Program Representation for HPC

Gabriel Dos Reis and Bjarne Stroustrup

Department of Computer Science

Texas A&M Univeristy

A vector operation

```
void
combine(double* z, double a, const double* x,
        const double* y, int n)
{
   for (int i = 0; i < n; ++i)
      z[i] = a * x[i] + y[i];
}
```

Can `combine()` be parallalized?

- Yes, if e.g. `z` doesn't overlap with either `x` or `y`.
- No, if e.g. there is some overloapping between `z` and either `x` or `y`.

A vector operation (cont'd)

```
const int N = 2000;
double mat[N][N] = { /* .... */ };
combine(&mat[56][0], 0.3476, &mat[89][0],  &mat[67][0], N);
// ...
double fib[2 * N];
fib[0] = fib[1] = 1.0;
combine(&fib[0] + 2, 1.0, &fib[0], &fib[0] + 1, N);  // ???
```

A vector operation with types

```
template<Parallelizable Vector, Numeric T>
    where Same<Vector::value_type, T>
void
combine(Vector& z, T a, const Vector& x, const Vector& y)
{   z = a * x + y; }
```

`Numeric` and `Parallelizable` are **concepts** (set of assumptions), not classes.

- `Numeric` says `T` supports numeric operations;
- `Parallelizable` says:
  - it is OK to subscript a `Vector` (`[]`);
  - it is NOT OK to take the address (`&`) of a `Vector`.

Concept checking

Concepts are predicates on types (types of types).

Templates definitions can be checked separately from their uses:

- Definitions are checked based on the set of assumptions (concepts) listed in the interfaces;
- Uses are checked to ensure that assumptions in template definitions are fullfilled.

If definitions and uses pass concept checking then the program should link successfully.

Concept checking with IPR

Concepts — in IPR — will probably have a notation in C++0x.

- Concepts are represented directly as types and predicates
- IPR does not require a language extension — it is a typed abstract syntax tree that can be annotated
- It is easy to turn a non-template function into a template function – abstracting over set of types
- Operations are checked by looking at the abstract syntaxt trees listed in a concepts

Anatomy of a declaration

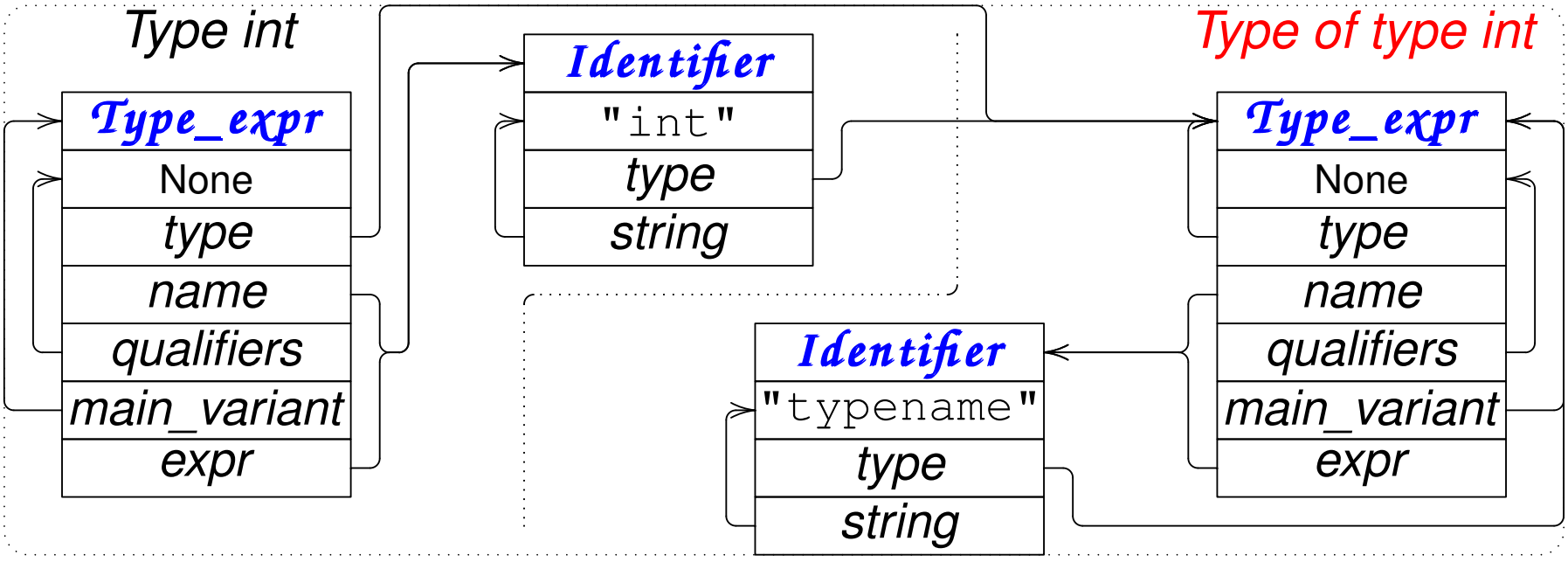
A declaration is a specification of a **type** for a **name**, in a given scope, with an optional **initializer**.

```
const int bufsz = 1024;
int abs(int);
class Var {
public:
    const string& name() const;
    const Type& type() const;
    const Expr& initializer() const;
    //...
};
template<typename T, int N> struct buffer {
    T data[N];
};
```

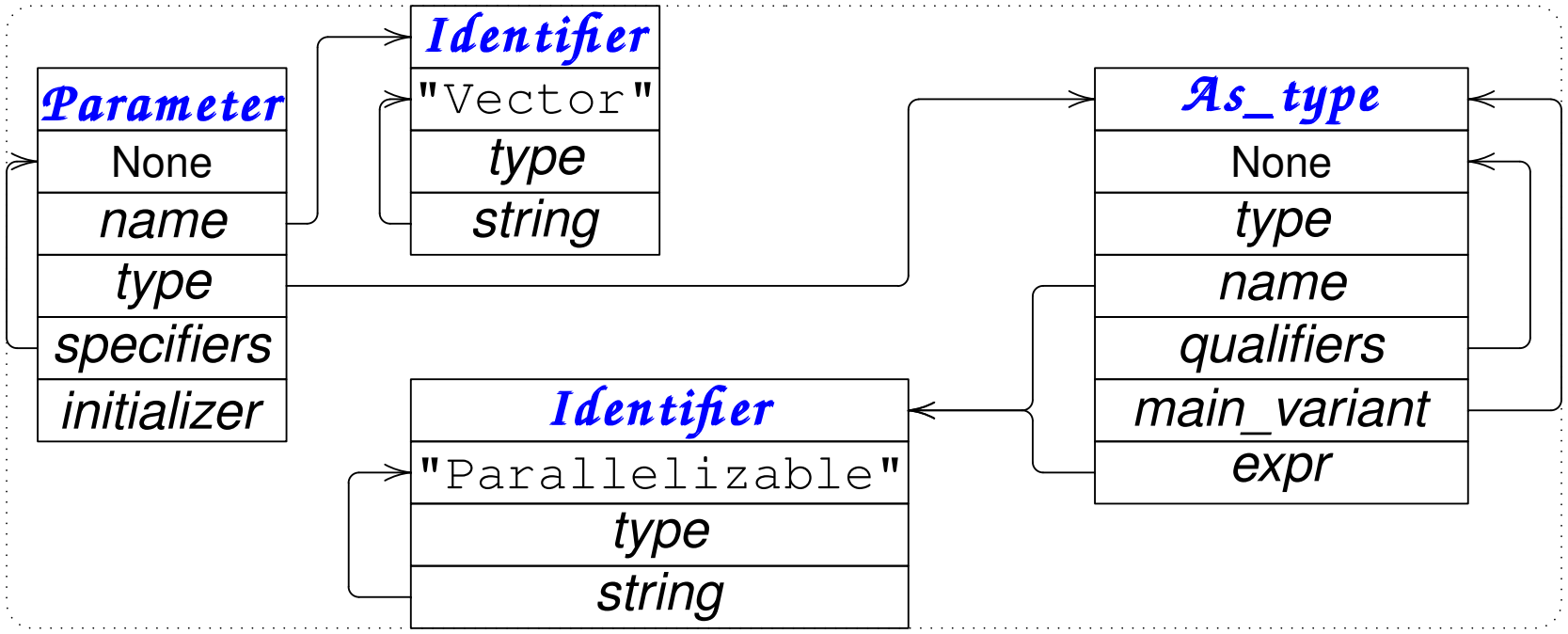
Generalized C++ type system

$$
\begin{aligned}
type ::= \; & \mathit{Pointer}(type) \mid \mathit{Reference}(type) \mid \mathit{Array}(type, expr) \\
& \mid \mathit{Const}(type) \mid \mathit{Volatile}(type) \\
& \mid \mathit{Function}(product, type, sum) \\
& \mid \mathit{Class}(scope, scope) \mid \mathit{Union}(scope) \mid \mathit{Enum}(scope) \\
& \mid \mathit{Ptr\_to\_member}(type, type) \\
& \mid \mathit{Namespace}(scope) \\
& \mid \mathit{Decltype}(expr) \mid \mathit{Type\_expr}(expr) \\
& \mid \mathit{Template}(\sigma_{type}, type) \\
& \mid \mathit{Product}(\sigma_{type}) \mid \mathit{Sum}(\sigma_{type}) \\
& \mid \mathit{Concept}(\sigma_{type}, scope)
\end{aligned}
$$

Representation of `int`



Parallelizable `Vector`



IPR as C++ library

- Expression-based language
  - E.g. statements, declarations are expressions too
- Simple, can represent incomplete or erroneous programs
- Two interfaces: Properly encapsulate implementation details from users:
  1. Purely “functional”, abstract classes, for most users
     - No mutation operation on abstract classes
     - Users don’t get pointers directly
  2. Mutating (operates on “concrete” classes)
     - Users get to use pointers for in-place transformation
- Library (not users) deals with memory management
- Traversal or “climbing” is based on the Visitor Design Pattern

A greating experiment (GCC-3.4.2)

```
#include <iostream>
int main() { std::cout << "Hello, World" << std::endl; }
```

- $29,248$ loc — from which $27,566$ non-blank, non-cpp directives loc;
- $60,855$ calls to type constructors
  - 60% for named types (only 1% are syntactically distinct),
  - 17% for pointer types,
  - 11% for `const`-qualified types,

Duplicate nodes for named types would take about $2\mathrm{Mb}$ (estimation from below).

Current and future work

- Complete infrastructure
  - EDG and GCC interfaces
  - Represent headers (modularity) directly
  - Complete type information in XPR
- Initial applications
  - Style analysis
    - including type safety and security
  - Analysis and transformation of STAPL programs
- Build alliances

References

- G. Dos Reis and B. Stroustrup: *Representing C++ Directly, Completely and Efficiently*, TAMU technical report 2005
- B. Stroustrup, *XTI: eXtented Type Information*, Tech. report, AT&T Research, unpublished, 2002.
- B. Stroustrup, G. Dos Reis, *Concepts – Design choices for template argument checking*, ISO/IEC JTC1/SC22/WG21 no. 1522, 2003.
- B. Stroustrup and G. Dos Reis, *Concepts - syntax and composition*, ISO/IEC JTC1/SC22/WG21 no. 1536, 2003.
- B. Stroustrup *Concept checking - A more abstract complement to type checking*, ISO/IEC JTC1/SC22/WG21 no. 1510, 2003.
- B. Stroustrup and G. Dos Reis, *A concept design*, ISO/IEC JTC1/SC22/WG21 no. 1536, 2003.