High Level Program Representation for HPC

Gabriel Dos Reis and Bjarne Stroustrup

Department of Computer Science
Texas A&M University
301 H.R. Bright Building
College Station, TX 77843-3112

Abstract. We briefly present IPR, a general high-level representation of C++, and show how it allows concepts (that is basically a type system for types) to be applied to correctness and performance problems in high-performance computing. We use a concept Parallelizable as a simple example of how to achieve anti-aliasing without language modification. A companion paper presents the Pivot framework for C++ program analysis and transformation of which IPR is the central part.

1 Introduction

Type systems have been introduced in programming primarily for correctness and efficiency purposes. Knowing at translation time, for example, that an operation involving read and write accesses actually is alias free can be profitably exploited in efficient code generation. Some programming languages like FORTRAN are designed in a way that makes such an assumption always hold. Other general purpose programming languages such as C or C++ allow only a restricted set of type-based aliasing. For example a pointer of type `void*` can be used to access any kind to data, but a pointer of type `int*` cannot be effectively used to access data of type `double`.

The typeful programming discipline can make programs not only correct and efficient. Abstract representation or representation of programs naturally brings symbolic manipulation. In this paper, we present an approach to correctness and performance based on high-level, fully typed abstract representation of ISO C++ [2, 1] programs developed as part of *The Pivot* framework. We will use the notion of parallelizable vector operation as a running example. *The Pivot* infrastructure is discussed in a companion paper [3].

2 A notion of parallelizable

Consider the classic operation

```
z = a * x + y;
```

where `a` is scalar, `x`, `y` and `z` denotes vectors, and the operations `*` and `+` are component-wise. It can be parallelized if we know that the destination `z` does not overlap with the sources `x` and `y` in a way that display non-trivial data dependencies. That happens, for example, if we know no vector has its address taken. For exposition purpose, we will simplify the notion of Parallelizable to a collection of types whose objects support the operation `[]` (subscription) but not `&` (address-of). For instance, in the generic function

```
template<Parallelizable T>
void f(T v)
{
   v[2];    // #1: OK
   &v;      // #2: NOT OK.
}
```

line `#1` is valid but line `#2` is an error because it uses a forbidden operation. We've generalized the standard notation `template<typename T>` which reads "for all `T`", to `template<Parallelizable T>` meaning "for all `T` that are `Parallelizable`".

We use the word *concept* to a designate collection of properties that describes usage of values and types. There are proposals to integrate concepts into the C++ programming languages. However, using IPR we can handle concepts without waiting for the C++ standards committee to decide on the technical details of concepts.

A programmer might use Parallelizable to constrain the use of a vector:

```
vector<double> v(10000);
// ...
f(v);
```

Here we now know that `f()` will not use `&` on `v` eventhough the standard library vector actually allows that operation. We can use `f()` with its no-alias guarantee for any type that supports subscripting. For example we might use a STAPL `pvector`:

```
pvector<double> vd(100000);
// ...
f(vd);
```

The concept checking makes no assumptions about the construction of the types used beyond what the concept actually specifies (here, a Paralleizable provides `[]`). In particular, no hierarchical ordering or or run-time mechanisms are required.

Below, we will briefly present a high-level representation of C++ programs that support concept-based analysis and transformations.

3 Internal Program Representation

As part of the Pivot framework, developed at Texas A&M University, we have designed and implemented a C++ library for internal representation of C++

programs. It can be thought of as a fully typed abstract syntax tree. The IPR library models a superset of ISO Standard C++. The emphasis is on generality and preservation of type information.

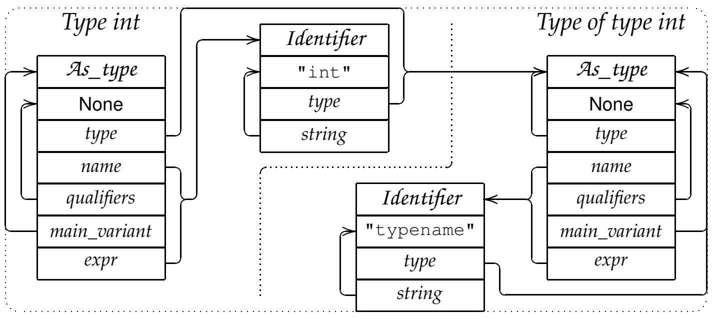

Fig. 1. IPR model for the C++ type `int`

A translation unit is represented as a graph with a distinguished root. That node denotes the sequence of top-level declarations. A declaration consists in three parts: (a) a name; (b) a type; and (c) an optional initializer.

A characteristic of IPR is that every entity in a C++ program is viewed as an expression possessing some type. In particular types have types, which we represent as concepts. Figure 1 depicts the representation of the built-in type `int` in IPR. The way it is constructed is as follows: first build an *Identifier* node with the string `"int"`. Then, state it is actually a type, through application of the type constructor *As_type*. Those steps are very close to the notion of *built-in*. The type of `int`, is the concept of type. There is a specific IPR node to denote the concept of type, built similar to the way we just explain for `int`.

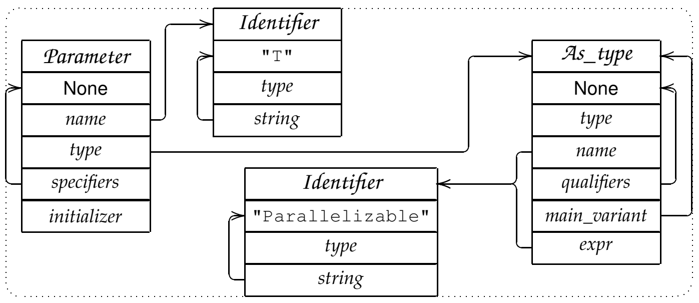

Fig. 2. IPR model `Parallelizable T`

In fig. 2, we've drawn a simplified view of the representation of the declaration `Parallelizable T`. The declaration of the template-paramater `T` has type `Parallelizable`, which we think of in this simplified situation as built-in.

Note how Parallelizable fits into the IPR framework without modification or special caseing. Thus, Parallelizable is simply a (deliberately trivial) example of what can be done with concepts in general.

4 Concept-based analysis and transformation

Concepts are the basis for checking usage of types in templates, just like ordinary types serve to check uses of values. Concept checking is done at two sites: (a) at template use site; and (b) at template definition site. If checking succeeds at both sites, then it is ensured that a template argument is used according to the semantics expressed in the concepts it models. In the particular case of Parallelizable, it means that no vector has its address taken, and consequently parallelization transformations could be safely applied.

Given a node that represents a function declaration, we can transform it into a templated version and concept-check it. Consequently, we can check whole program as if it were fully templated. One may conceive of linguistic support like

```
vector conformsto Parallelizable;
```

to assert conformance of usage of `vector` objects to the Parallelizable concept. However, since we're doing the verification and transformation on an abstract syntax tree, no language extension or source code modification is actually required.

5 Conclusion

We have shown an approach to semantics-based analysis and transformation that does not require existing language modification. It relies on a high-level representation of program with emphasis on type information. The IPR library is part of a larger framework, *the Pivot*, that is discussed in a companion paper.

References

1. International Organization for Standards, *International Standard ISO/IEC 14882. Programming Languages* — C++, 2nd ed., 2003.
2. B. Stroustrup, *The C++ Programming Language*, special ed., Addison-Wesley, 2000.
3. B. Stroustrup, G. Dos Reis, *The Pivot: A brief overview*. Companion paper.