C++ Compile-Time Type System for the CodeStubAssembler

Attention: Shared Google-externally

Authors: tebbi@
Last Updated: 2017-10-25

CLs: Typed Nodes and Variables, Union Types

We want a static type system for the `CodeStubAssembler` to avoid confusion between the different runtime types defined in `objects.h` and untagged machine representations. This complements the existing `MachineGraphVerifier` in that it is easier to use (C++ compile errors instead of assertion failures) and allows for a finer-grained distinction between runtime classes.

Goals and Requirements

- The checks should be performed as early as possible, preferably at C++ compile time.
- There is a lot of existing untyped CSA code that needs to transition gradually to the new typed world.
- At places where the static type system is insufficient, it should be possible to easily insert dynamic checks for debug builds.
- No runtime overhead in release builds.

Proposal

We implement the type system by wrapping the existing type `Node*` for SSA values with a C++ template `TNode<A>` for typed nodes where `A` is a C++ class that serves as a type tag. In addition to all the classes from objects.h (e.g., `TNode<JSObject>` and `TNode<Smi>`), we define a hierarchy of tag classes for untagged values.

```
struct UntaggedT {};
struct IntegralT : UntaggedT {};
struct WordT : IntegralT {};
struct IntPtrT : WordT {};
struct Word64T : IntegralT {};
struct Int64T : Word64T {};
struct Word32T : IntegralT {};
struct Int32T : Word32T {};
struct Float64T : UntaggedT {};
struct BoolT : Word32T {};
...
```

The class hierarchy of untagged types as well as the type hierarchy in objects.h enable subtyping on typed nodes. For example, `TNode<HeapNumber>` is a subtype of

`TNode<HeapObject>`. All tagged values are contained in `TNode<Object>`, while all untagged typed nodes are subtypes of `TNode<UntaggedT>`.
To allow for a gradual transition of the existing code stubs to use typed nodes, `TNode<A>` is implicitly convertible to `Node*`. This is necessary to call legacy functions that still expect a `Node*` argument. However, when we transition a `CodeStubAssembler` helper function to typed nodes, we should still be able to call this helper function from untyped code stubs to allow for a gradual transition. In this case, we want an implicit conversion from `Node*` to `TNode<A>`. However, adding both implicit conversions leads to ambiguities and we do want to restrict the places where we convert an untyped node to a typed node. An explicit conversion conflicts with the desire to transition code stubs incrementally. We propose to temporarily use a second template `SloppyTNode<A>`, which allows for implicit conversion from and to `Node*`, but which is only used for arguments of helper functions. For example like this:

```
TNode<Int32T> Int32Add(SloppyTNode<Int32T> a, SloppyTNode<Int32T> b);
```

This way, the helper functions can still be used in untyped stubs, but accidental conversions from untyped to typed nodes are not possible. We plan to replace all uses of `SloppyTNode<A>` with `TNode<A>` as soon as the transition to typed code stubs is finished.
For explicit conversions, we offer the `CodeAssembler` methods `UncheckedCast<A>(x)` and `CAST(x)`, where the latter only works with tagged types and inserts a debug build runtime check. This runtime check invokes `Object::Is##A()` via a fast external C call. The `CAST()` macro infers the target type from the calling context (using C++ implicit conversions in the form of `operator TNode<...>()`). We use a macro to record the callsite for better runtime error messages.
Here is an example of control-flow dependent type conversions:

```
Branch(TaggedIsSmi(x), &if_issmi, &if_isheapnumber);

BIND(&if_issmi);
{
  TNode<Smi> x_smi = CAST(x);
  // ...
}
BIND(&if_isheapnumber);
{
  TNode<HeapNumber> x_hn = CAST(x);
  // ...
}
```

Despite the minimal annotation, this example generates debug build runtime checks that the static annotations match the actual runtime types in both cases of the branch. With the same

technique, it is possible to automatically insert runtime checks for parameters. For example, the following implicitly emits runtime checks for all parameters.

```
TF_BUILTIN(ArrayIsArray, CodeStubAssembler) {
 TNode<Object> object = CAST(Parameter(Descriptor::kArg));
 TNode<Context> context = CAST(Parameter(Descriptor::kContext));
 // ...
```

Typed Variables

Statically typed variables are straightforward. They are templated by their type and emit and consume `TNode<>` values. To improve readability, the new typed variables overload `operator=` as a replacement for `Bind()` and implicitly convert to the current `TNode<>` value.

```
TVARIABLE(Smi, x, CAST(Parameter(Descriptor::kX)));
Branch(SomeCondition(), &if_true, &if_false);

BIND(&if_true);
x = SmiConstant(8);
Goto(&merge);

BIND(&if_false);
x = SmiAdd(x, SmiConstant(8));
Goto(&merge);

BIND(&merge);
Return(x);
```

Union Types

The type `UnionT<T1, T2>` represents the union of `T1` and `T2`. For example, `TNode<UnionT<Smi, HeapNumber>>` is either a `Smi` or a `HeapNumber`. Such union types only make sense for tagged values, because they need a uniform representation and the ability to distinguish the types at runtime. Using a C++ type alias, we write `Number` instead of `UnionT<Smi, HeapNumber>`. Union types can be nested and have subtyping. So `TNode<Number>` is convertible to `TNode<UnionT<HeapObject, Smi>` and `TNode<Object>`. However, `TNode<Object>` is *not* convertible to `TNode<Smi, HeapObject>`, as this would require knowledge about all existing subclasses. It would be possible to allow such conversions, but it seems this is rarely necessary and probably slows down compile times.

Pairs of Values

Some Turbofan nodes return two values, which can be selected with projections. We introduce the special type `PairT<T1,T2>` for this situation. The projection function then becomes templated on the projection index:

```
template<class T1, class T2>
TNode<T1> Projection<0>(TNode<PairT<T1,T2>> value);
template<class T1, class T2>
TNode<T2> Projection<1>(TNode<PairT<T1,T2>> value);
```

Control Structures (controversial, unimplemented)

Prototype CL

CSA features an assembly-like coding style where labels and jumps are the only control structures. To improve readability, we propose to add traditional control structures like conditionals to CSA. Using C++ lambdas, the following syntax is possible:

```
If(cond1) - [&] {

} | ElseIf(cond2) - [&] {

} | Else - [&] {

};
```

This just inserts the necessary jumps and labels in the background, so manual jumps into and out of the conditional branches are still possible. Each branch is a lambda capturing all local variables from the context and is executed right away and exactly once, in the order of definition. Thus code inside of a lambda behaves almost exactly like code outside of it. One difference is that a C++ `return` only terminates the lambda, but a `return` in the middle of a CSA function does not make sense anyway. The syntax with operator overloading is chosen to reduce the number of parentheses while tricking clang-format into inserting the right line breaks.

A frequent use of conditionals (or equivalent label-jump constructions) is to determine the type of a value at runtime, for example checking if a tagged value is a `Smi`. While doing this with conditionals and CAST(...) is possible, we propose an alternative which is both more concise and offers better static checks:

```
Match(value) | [&](TNode<Smi> value) {

  // Executed if {value} is a {Smi}.

} | [&](TNode<HeapNumber> value) {

  // Executed if {value} is not a {Smi}.
  // Statically check that it has to be a {HeapNumber}

};
```

The runtime-checks to perform can be derived automaticaly from the argument types of the lambdas. It is possible to statically verify that the type checks are exhaustive (there is always a branch to match any value) and that all branches are reachable (the earlier branches do not subsume a later branch). In the example above, `value` needs to have type `TNode<Number>` and it should not be possible to add another branch that checks for `Smi`.