JitBuilder 2.0

Mark Stoodley

2020-03-12

JitBuilder 2.0 Goals and Motivation

- Improve usability
  - Dual mental model (JitBuilder, OMR compiler) can be difficult for users
  - Logging/debugging/analysis with JitBuilder concepts would be nicer
- Improve extensibility
  - Make it easier to add new types and operations
  - Operate on JitBuilder as an IR before it flows to code generation
- Easier experimentation path for new compiler concepts
  - More freedom to try new things outside existing OMR compiler framing
  - (Wild n Crazy) evolutionary path from OMR compiler IL to something “else”?
- Decoupling from OMR compiler (don’t get upset!)
  - More freedom to ”play” without OMR compiler and dependencies
  - Easier to have “client” and “compiler”
  - Easier to bring JitBuilder to other languages?
  - Distinct from OpenJ9 so fewer constraints on how it evolves forward

Prototype Exploration

- What follows describes a prototype (incomplete) implementation I primarily wrote over the 2019 Christmas holidays
- Completely independent of OMR (but only because I didn’t write the code generator yet)
- Implements enough of JitBuilder for MatrixMultiply
- Includes some rudimentary logging (including pretty printing JitBuilder “IL”)
- Demonstrates some ”dialect reducing”
    - E.g. translating ForLoopUp into lower level operations like IfCmpGreaterThan
- Roughly demonstrated adding a Complex type and ConstCompex operation
    - Then a transformer to reduce all operations from Complex to Double operations

JitBuilder 2.0 Key Elements

- Familiar stuff (some with twists):
  - Builder (a.k.a. IlBuilder)
    - FunctionBuilder (a.k.a MethodBuilder)
    - Other builder types
  - Type (a.k.a. IlType)
  - TypeDictionary
  - Value

- Lots of new stuff
  - Config
  - Dialect
  - Operation (using Action enum)
    - Add, Sub, etc.
  - Symbol
    - ParameterSymbol
  - TypeGraph
  - Visitor - PrettyPrinter
  - Transformer - DialectReducer

JitBuilder 2.0 Key Elements

- Familiar stuff (some with twists):
  - Builder (a.k.a. IlBuilder)
    - FunctionBuilder (a.k.a MethodBuilder)
    - Other builder types
  - Type (a.k.a. IlType)
  - TypeDictionary
  - Value

- Lots of new stuff
  - Config
  - Dialect
  - Operation (using Action enum)
    - Add, Sub, etc.
  - Symbol
    - ParameterSymbol
  - TypeGraph
  - Visitor - PrettyPrinter
  - Transformer - DialectReducer

Some general improvements/changes

- Everything is a data structure
  - Values, Types, Symbols, Operations, Builders, etc.
  - Every instance has an ID and potentially a name (often “”)
- Everything can be queried and traversed
  - op.getOperand(i), op.getResult(i), value.getType(), …
- Aimed for “backwards compatibility”
  - Some of the names changed, mostly just search and replace
  - Used references rather than pointers in prototype (may change)
  - Removing “double pointer” arguments that allocate IlBuilders automatically
  - Kept everything in OMR::JitBuilder namespace
  - Expecting client API generation to get simpler
    - Maybe even become a first class part of using JitBuilder
    - Implementation objects become “code generation”

OMR::JitBuilder::Value is some kind of *data*

```
public:
   uint64_t id() const            { return _id; }
   Type & type() const            { return _type; }
   bool usedBeyondParent() const { return _usedBeyondParent; }
```

- Cannot create a value directly (no public constructor or factory)
  - Only Builder gets this privilege currently
  - Helps maintain correctness as nothing user can do to get it “wrong”
- Type “categorizes” legal data
- New in this prototype: you can ask for a Value’s Type
  - Previously, you could really only ask for a primitive type (a.k.a. OMR compiler’s TR::DataType)
  - Operations compute the result Value’s new Type based on the Type of the operands
- Open question: immutability of Value

OMR::JitBuilder::Operation *does* something

- Abstract base class for operations using Action enum
  - enum Action { aAdd, aSub, aLoad, aStoreAt, aIfThenElse, aForLoopUp … }

    Action action() const
  - Has a parent builder where this operation resides

    Builder & parent() const
  - Defines virtual getters and setters for generic characteristics of an Operation

    e.g.
    ```
    int32_t numOperands(), Value *operand(i)
    int32_t numResults(), Value *result(i)
    int32_t numTypes(), Type * type(i)
    float getLiteralFloat()
    ```
    and other primitive types
  - Iterators too:

    e.g.
    ```
    ValueIterator OperandsBegin(), OperandsEnd()
    ValueIterator ResultsBegin(), ResultsEnd()
    TypeIterator TypesBegin(), TypesEnd()
    BuilderIterator BuildersBegin(), BuildersEnd()
    ```
- Concrete subclasses for each kind of action
  - Add, Sub, Load, StoreAt, IfThenElse, ForLoopUp, etc.

OMR::JitBuilder::Builder is a sequence of operations

- Basically a label and a sequence of operations
    - Call operations on a builder object to append to its sequence
- Has a containing FunctionBuilder
- May be *bound* to an operation
    - Bound means control to and from builder is determined by the operation
    - E.g. IfCmpGreaterThan() branches to an *unbound* builder (no implicit merge back)
    - E.g. IfThenElse() uses two *bound* builders (because IfThenElse creates merge)
    - More abstract operations more likely to use bound builders
- Operations are validated as they are added
    - Uses TypeGraph which maps operand Types + Action to result Type(s*)
    - Every Operation class registers valid combinations
    - Creating a PointerTo(type) registers valid addressing for the resulting type

FunctionBuilder is a callable Builder object

- Previously known as “MethodBuilder”
- Has a name
- Can take parameters which have Types
- Can return one (or more, in principle) Values

TypeGraph ensures valid operations

- TypeGraph object is part of TypeDictionary
  - Initializes itself and must ask all Operations to initialize their type ”productions”
- E.g.

```
void
OMR::JitBuilder::Add::initializeTypeProductions(TypeDictionary & types, TypeGraph &
graph)
   {
   Type & I8 = types.Int8; graph.registerValidOperation(I8, aAdd, I8, I8);
   graph.registerValidOperation(types.Int16 , aAdd, types.Int16 , types.Int16 );
   graph.registerValidOperation(types.Int32 , aAdd, types.Int32 , types.Int32 );
   graph.registerValidOperation(types.Int64 , aAdd, types.Int64 , types.Int64 );
   graph.registerValidOperation(types.Float , aAdd, types.Float , types.Float );
   graph.registerValidOperation(types.Double, aAdd, types.Double, types.Double);

   graph.registerValidOperation(types.Address, aAdd, types.Address, types.Word);
   }
```

TypeDictionary adds pointer types to TypeGraph

```
void
OMR::JitBuilder::TypeDictionary::registerPointerType(PointerType & pointerType)
   {
   _graph.registerType(pointerType);

   Type & baseType = pointerType.BaseType();
   _graph.registerValidOperation(pointerType, aAdd, pointerType, Word);
   _graph.registerValidOperation(Word, aSub, pointerType, pointerType);

   _graph.registerValidOperation(pointerType, aIndexAt, pointerType, Word);
   _graph.registerValidOperation(baseType, aLoadAt, pointerType);
   _graph.registerValidOperation(NoType, aStoreAt, pointerType, baseType);
   }
```

Some very basic traversal support: Visitor

- Visitor:

```
// subclass Visitor and override these functions as needed
   virtual void visitFunctionBuilderPreOps(FunctionBuilder & fb)  { }
   virtual void visitFunctionBuilderPostOps(FunctionBuilder & fb) { }
   virtual void visitBuilderPreOps(Builder & b)                   { }
   virtual void visitBuilderPostOps(Builder & b)                  { }
   virtual void visitOperation(Operation & op)                    { }
```

Example Visitor: PrettyPrinter

- 450 lines in header and source to produce JitBuilder logs, e.g.

```
[ FunctionBuilder MB0 "matmult"
   [ types td0 ]
   [ origin MatMult.cpp::40 ]
   [ returnType t0]
   [ parameter "C" t8 0 ]
   [ parameter "A" t8 1 ]
   [ parameter "B" t8 2 ]
   [ parameter "N" t3 3 ]
   [ local "sum" t6 ]
   [ local "i" t3 ]
   [ local "j" t3 ]
   [ local "k" t3 ]
   [ operations
      op0: v0 = Load "A"
      op1: v1 = Load "B"
      op2: v2 = Load "C"
      op3: v3 = Load "N"
      op4: v4 = ConstInt32 0
      op5: v5 = ConstInt32 1
      op6: ForLoopUp "i" : v4 to v3 by v5 body B1 ""
      op31: Return nil
   ]
   ….
]
```

Some very basic traversal support: Transformer

- Transformer:

```
// To implement any transformation, subclass Transformer
//   and override the virtual functions below as needed

// called once on each FunctionBuilder before any other processing
virtual void transformFunctionBuilder(FunctionBuilder & fb) { }

// called once each Builder object before its operations are processed
virtual void transformBuilderBeforeOperations(Builder & b) { }

// called once on each operation
// the operation will be replaced by the contents of any non-NULL Builder object returned
virtual Builder * transformOperation(Operation & op) { return NULL; }

// called once each Builder object after its operations are processed
virtual void transformBuilderAfterOperations(Builder & b) { }
```

Example Transformer: AppendBuilderInliner

```
class AppendBuilderInliner : public Transformer
   {
public:
   AppendBuilderInliner(FunctionBuilder &fb)
      : Transformer(fb)
      {
      }

protected:
   virtual Builder * transformOperation(Operation & op)
      {
      if (op.action() != aAppendBuilder || op.builder(0)->isTarget())
         // ignore anything but AppendBuilder and AppendBuilders that are targets of branches
         return NULL;

      // replace AppendBuilder operation with the operations from the appended builder
      return op.getBuilder(0);
      }
   };
```

Debugging

- Logs can be produced
- Visitor and Transformer can log what they do
- Transformer supports “performTransformation()”
  - FunctionBuilder counts transformations
  - Automatically prints before and after trees

ForLoopUp reduction transformer

```
• ( 2 ) Transformation:
• op12: ForLoopUp "k" : v4 to v3 by v5 body B3 ""
•     [ Builder B12 "" notTarget
•         [ operations
•             op52: Store "k" v4
•             op53: v33 = Load "k"
•             op54: IfCmpGreaterOrEqual v33 v3 then B15 ""
•             op60: AppendBuilder B13 ""
•             op61: AppendBuilder B15 "" (Label)
•         ]
•     ]
```

What else?

- Implemented a Complex Type on top of JitBuilder
  - Added an operation called “ConstComplex R,I”
- Modified Matrix Multiply in terms of Complex
  - Changed Double to Complex
  - Changed ConstDouble 0.0 to ConstComplex 0,0
- Write a ComplexReducer transformer
  - Replaces all Complex Values with two Double values
  - Maps all operations with Complex operands to two Double operations

What next?

- Write up an issue
- Push my current code to a git repo for
- Get feedback
- Keep going ☺