Roadmap to a Better Quick C-- Compiler

Christian Lindig

November 2, 2010

The current QC-- compiler design is in its first iteration and not surprisingly, it shows some flaws. Below I am trying to explain some of the problems.

The single most important data structure in the compiler is the control flow graph (CFG). It connects the front end with the back end. The front end represents a program as an abstract syntax tree (AST) and the back end emits assembly code.

- The CFG is a directed graph that is built from a tree-shaped abstract syntax tree (AST). The AST includes named references across the tree that become explicit edges in the CFG. Constructor functions must gap this impedance mismatch. The current design knows two broad classes of nodes: labeled nodes and unlabeled nodes. The labeled nodes correspond to named entities in the AST that might be referenced. During the construction of a CFG, a labeled node might be referenced before it is defined. Currently it is difficult to create a node that branches to a labeled node when the labeled node is not also created locally.

- The current design forces to built a CFG bottom-up: constructor functions take successor nodes as arguments and return a new node. This allows to encode constraints like the number of successors into constructor functions, but also limits the code that builds a CFG.

- Nodes are not created equal but come in different flavors: a branch node is created by a different constructor than an assign node. This leads to a quite heavy implementation of nodes. A layered approach might be an alternative, where all nodes are structurally equal, but are carrying different payloads.

- Each node contains an instruction. It is desirable to change the representation of that instruction during the lifetime of a CFG. This is impossible with the current design. A possible and rather heavy solution is to create during run-time an isomorphic copy of a CFG and let that isomorphism also change the representation of instructions. Because of the imperative nature of the CFG and the many informations encoded into a node (see above) this is difficult to implement. It might be possible with a more lightweight node representation. Another possibility offer existential types as they are provided by objects. If an instruction that is part of a node just has to conform to a class type, any object implementing that class type can be stored in a node. It would be possible to update instructions in an existing CFG without having to copy it first.

- The instruction inside a node is target specific. This forces CFG transformations that want to insert new nodes to be target specific. In particular, adding jumps requires to know the target specific jump instruction. I believe now that construction of the CFG should be target un-specific to allow easier manipulation of the CFG, in particular to insert new jumps and labels.

- A label node is associated with a string and a symbol. A symbol is an abstract value handed out by an assembler for this label. The string identifies the node inside the CFG. Knowing the string, the node can be looked up in order to add an edge to it. Because of the additional (target specific) symbol it is difficult to add new label in transformation stages.

- A CFG may include edges that are there just to inform the data flow analysis. Since edges are implemented as references it is not clear how to represent these artificial edges. One possibility is the introduce artificial nodes on these edges for which no code is generated.

- Some nodes fall into a label node, others jump to it. This requires careful linearization such that the node that falls into a label really precedes it. Linearization would be easier if all transfers to a label were explicit gotos.

An important special case of an RTL location is a register. A register is the main focus of liveness analysis and register allocation in the compiler. In the current design, a register is *not* part of the RTL framework but instead

treated outside. This requires conversions and thus complicates the design. Why are registers not an RTL location, analogously to variables?

The RTL framework refers to location spaces but does not include data structures to describe them. Like in the case of registers, spaces are treated outside of the RTL framework.

The `Talloc` module provides fresh temporaries for the register allocator. To obtain a new temporary, a client has to supply a space and the width of a new temporary. The width information is redundant since every space has a unique width. The width of each space is known by a `Talloc.Multiple.t` value because it is created from a list of space descriptions.

Calling conventions and placement of global variables are handled by automata. Because calling conventions and global variables are target dependent, it is desirable to define these automata where targets are defined. This is not possible with the current design: the automaton type is abstract and thus can have only one implementation. A better design might use existential types, hence objects. I propose to use objects directly rather than hiding them behind a layer of universal types like in `Automaton`.

Calling conventions are implemented as automata. When the signature of a function is pushed trough an automaton it responds with abstract locations that describe where the corresponding parameters are stored. The only automaton currently defined is a toy example and probably not good for anything.

The code expander re-writes an RTL into a sequence of RTLs such that the sequence has the same effect as the original RTL and each new RTL is representable as a machine instruction. Code expansion is implemented with BURG. The current expander re-writes a node in the CFG into a straight sequence. However, it is not possible to create nodes that diverge and join. This would be useful to re-write boolean expression into control flow using the expander. Expansion of boolean expression is therefore handled outside of the BURG-based expander which might hinder optimizations.

The BURG implementation OCamlBurg generates purely functional code. More imperative code might provide better (memory) performance. This would require to no change to the compiler but work on the code generator of OCamlBurg.

The compiler's back end is controlled by Lua code. To make this feasible, essential functions are written in Objective Caml and exported to Lua.

Currently two styles are in place to register such a function in the Lua interpreter: The first registers a large chunk of functions centrally registered in the `Driver` module. The second is decentralized: each module provides a sub-module that is linked into the Lua interpreter and registers functions there. This seems to be more flexible and should be the preferred style.

The primitives for C-- floating-point comparison must be re-thought. Currently they return a 2-bit *value* and thus allow the user to store them. This is a serious challenge to code generation which would prefer to keep all comparison results in the processors status register. This is not a problem with boolean results from integer comparison because booleans are not values. When a user can store the result the code generator has not full control over them and thus cannot guarantee, that only a single comparison result is alive at any one time.

A machine instruction is represented as an RTL. The function that creates it is automatically generated, as well as a pattern (as part of a matching function) that identifies it. The generated constructors and patterns exhibit the following problems:

- The RTL representation for some instructions is huge (more than 100 constructors). This could be avoided with a simpler machine description at the cost of less semantic accuracy. The trick is to find the right balance, such that an RTL still captures the semantics of an instruction that an optimizer must know.
- An RTL can include an abstract bit vector. Because the representation of the bit vector is abstract, it cannot be matched in a pattern. The current solution is to match the bit vector with a variable and to apply an observer function to it in a guarding expression for the pattern.
- Current constructor functions cannot build instructions that include temporaries rather than hardware registers. This is not a problem for the recognizing pattern. Before a pattern sees an instruction the register allocator replaces each temporary with a hardware register.
- Both constructors and patterns must be able to deal with relocatable addresses, which they cannot at the moment. The hand-written constructors and patterns use an RTL encoding for relocatable addresses that they agree on.

- On the SPARC some synthetic instructions are represented by two machine instructions. They do not fit into the current framework where a constructor functions always returns one instructions. Synthetic instructions are currently hand-coded in the code expander.

To what degree should C-- and a C-- compiler for a 32 bit target support variables of size 8 and 16 bit? The QC-- compiler's front end accepts more programs than the back end can handle. In fact, the front end has no idea which cases the back end cannot handle.

An RTL operator is represented as a pair of a string and a list of integers. Since the number of operators is fixed, the string should be replaced by a sum type or similar. C-- operators are instances of RTL operators. They should be represented by distinct types. Currently both have the same representation.

After register allocation all memory requirements for a procedure are known and its stack layout can finally be assembled. In this step, a final pass is made over all RTLs to replace late compile-time constants. The problem is, that the replacements might lead to RTLs that violate the machine invariant that was established by the code expander previously. Another code expansion and register allocation step might be necessary. Since this can lead to new spilling that affects memory requirements, a fix-point iteration would be necessary.

The picture for relocatable addresses is still not clear. Both `Reladdr` and `Sledlib` provide relocatable addresses. Do relocatable addresses to be parameterized over the datatype of constants? Since it is a basic data type this parameterization would ripple trough the entire compiler.

A continuation is a pointer to a pair of two values located in the frame of procedure. While this general concept is probably target independent, the details might be not. How should the block of two value be aligned, and how are values inside the block to be aligned? Should a target define an automaton for to handle this?

Expression evaluation needs to be unified. The correct way is to translate an expression into an RTL expressions and then to evaluate it. Currently the constant evaluation phase of the `Elab` module evaluates constant expressions directly.

The assembler interface for DOT graph layouter calls a function that returns

an RTL as a string to use this string as a node description. The RTL string representation is too detailed for this purpose.

1 Further refactoring from NR

1. The abstraction `Proc.t` needs to be studied. There are connections to the generic expander and the stack layout.
2. Index width for temporary spaces could be a problem.
3. Since "register" *means* a compile-time index, perhaps this property ought to be reflected in the implementation.
4. The `Target2` and `Targets` modules are very heavyweight. This material should be integrated with the new back-end material (e.g., `Postexpander.S`, `Space.Standard`, and other suspects) so that one can easily specify a target in Lua. One hopes that the parts that are appropriate to do in Caml are at most the expander and the recognizer, and that the rest can be done in domain-specific fashion using Lua.

2 Misc

Restrictions on source code:

- The `bit` and `bool` operators are not supported.
- A nontrivial guard may appear only on a branch instruction.