Roadmap to a Working Quick C-- Compiler

Christian Lindig
Norman Ramsey

Early June, 2002

The purpose of this document is to explain what needs to be done by whom to lead to a working Quick C-- compiler. This document emphasize pieces that are missing, broken, or incomplete. Pieces that need refactoring are covered in the *Roadmap to a Better Quick C-- Compiler*.

[**A**]

1. The control-flow graph should be refactored, as outlined in *Roadmap to a Better Quick C-- Compiler*. This is a major item to be covered by the "control-flow graph working group." In addition to the problems listed in that other document, the working group should also address the following problems.

   July '02: Well under way

   (a) At least one operation in the current implementation of the CFG requires the introduction of a new label together with a new symbol for that label. Symbols are issued by an assembler, but this context is missing. As a hack, a symbol is created locally. However, this is wrong because it does not reflect the rules of the actual assembler. I would like to see a design where new branches can be introduced without worries about target specific symbols. This plays into the observation, that C-- has really two classes of labels: local labels introduced by the translation of statements like `if`, and global labels that are define in a C-- source file. At least local branches should not depend on symbols.

   (b) When the control-flow graph changes, what clients should be redone first? Picking the right clients means we get good early feedback on the redesign. Possible clients might include `ast2ir`, a register allocator, and a code expander (especially for short-circuit Booleans).

Christian, Glenn, John, Kevin, and Norman.

[**B**+] 2. *Identify suitable technology for writing machine-dependent components.* Generating everything from a machine is a research problem, and the solutions are not ready for use in the compiler. We must identify techniques that will produce solid code, which we expect to be primarily hand-written code. *Norman.* **July '02: Largely done**

[**B**+] 3. Build a *back end for a new target*, developing detailed porting instructions on the way. This process will include solutions to the following problems: **July '02: Largely done**

(a) A target description provides RTLs for instructions that change the control flow, like `jump`, `branch`, or `return`. These RTLs have the strange property that they are target specific on one hand, but are not representable as a single machine instruction on the other. The reason is, that they include complex expressions. The invariant of these RTLs should be clearly defined.

The solution is likely to involve reading the PC from a conventional location, writing a PC to a conventional location, and possibly a private invariant shared with the code expander. —NR **July '02: Req'd by expander**

(b) How does the optimizer know if it is safe to negate the condition on a conditional branch without violating the machine invariant? And if it not safe, what can the optimizer do? **July '02: Open**

(c) Figure out how to *reserve a stack slot in the code expander.* The SPARC cannot move a value from an integer register to a floating-point register. The code expander, which must implement such a move, needs a memory cell as temporary storage. What is the address of this cell? In principle, we need only one such cell per program. (Or one per thread, if scheduling is pre-emptive.) We might wish to allocate one per activation, in the stack slot of the current frame, because getting to a cell on the stack is faster and doesn't cost a register. **July '02: Done**

(d) *Stack-frame layout* should be part of the target specification. Currently the overall stack layout is hardcoded into the `freeze` function in the `Ast3IR` module.

Norman.

[**B**] 4. Write *variable placers* by hand. *Kevin.* **July '02: Open**

[**B**] 5. Develop either hand-written or machine-generated *operator context charts* to guide selection of temporaries. For use both in variable placement and in code expansion. *Kevin.* **July '02: Open (preliminary in x86)**

[**D**] 6. RTL operators describe compile-time, link-time, and run-time operations. The list of *operators available at compile time* is extremely limited. Someone should analyze the situation and work on the following items. **July '02: Open**

- We need precise documentation (perhaps even formal, via Lua?) of what operators are supported at compile time. N.B. an "operator" means a fully instantiated operator at a particular type, not a polymorphic promise that can be redeemed at any type.
- We need actual implementations of the compile-time operators.
- Some current implementations are dummies that return constant results; these implementations must be removed.

Kevin.

[**A**] 7. Identify the *critical compile-time operators* and make sure they are implemented correctly. These might include add and subtract, bit insert, bit extract, and shifts. *Christian.* **July '02: Open**

[**C**] 8. Partial *map of variables and callee-saves registers to locations* at each call site. *John or Kevin.* **July '02: Incomplete?**

[**C**] 9. Implement `span`, with support for lookup at each call site. *John or Kevin.* **July '02: Partial**

[**B**] 10. Specify and implement *calling conventions.* Christian is quite interested in this problem. *John or Christian.* **July '02: Partial**

[**A**] 11. Develop and successfully compile a small *test suite* for SPARC, including some C programs. Because of limitations of the compiler, it is likely that any C programs we can successfully compile and run will be carefully crafted for this purpose. It is OK if these programs don't do anything useful. *Christian.* **July '02: No longer relevant?**

[**B+**] 12. Implement *small arithmetic.* The compiler currently lacks a comprehensive plan how to deal with values of smaller than the size of a register. *Kevin and Norman.* **July '02: Incomplete**

[**A**] 13. Support a *complete set of RTL operators*, probably with new representation. Minimum here is to make sure that we have all 60+ operators and that everything typechecks. A new representation could be a bonus. *Kevin.* **July '02: Open**

[**A−**] 14. Identify, allocate, and initialize escaping *continuations*. This entails repairing the following problems: **July '02: Done**

(a) Document the representation of a continuation, including space set aside for incoming overflow parameters. Some internal abstraction for a continuation's run-time representation, similar to `Automaton.loc` might be appropriate.

(b) Identify which continuations escape.

(c) The translation of continuations in `Ast3ir` is bogus. A continuation *value* is a pointer into the stack where two values are stored: a pointer to the code of the continuation, and a stack pointer. The current representation of a continuation value is a link-time constant, which is wrong. A continuation *value* is a late compile-time expression: a sum of the stack pointer value and some late compile-time constant. The stack pointer in turn, depends on the currently active calling convention. *I have introduced a new abstraction* `Contn` *that addresses some of the problems. However, most things are not yet integrated and the translation of expressions with included continuation values is still wrong. – CL*

Here are some notes on this problem:

- The target must hide the representation of a continuation.
- The target will provide these values:
  - A block for continuations
  - Function from parameter list to block (or size/alignment) for a particular continuation
  - Function from stack location, code label to RTLs that initialize a continuation
  - An implementation of `cut to`.

  The front end will

  (a) Find which continuations escape

  (b) Allocate space for each one, managing the name of a continuation exactly as we manage a label in `stackdata`

(c) Emit initialization code obtained from the target

Christian.

[**B**+] 15. Implement `cut to` correctly. The implementation of `cut to` currently expects to put overflow parameters into the *incoming* area for the procedure containing the continuation. This is a bad plan; it is better for each continuation to have its own overflow area reserved on the stack. We believe this strategy will perform acceptably well because we expect most continuations to take at most one or two parameters, which can easily be passed in registers. (Consider that any callee-saves registers is fair game for a continuation parameter.) *Christian.* **July '02: Done**

[**B**] 16. Solve the following problem, which NR does not understand: *The control-flow graph lacks proper edges to indicate the data flow in the presence of continuations.* The first step is to identify exactly what the problem is, which Christian will do. *Christian.* **July '02: In progress**

[**C**] 17. Add a target-specific map for hardware registers, and ensure that the front end rejects named hardware registers that do not appear in the map. Might require changes in structure of front end to make it aware of target, or the front end might accumulate a set of such registers and have the checking done as an intermediate step. N.B. Christian recommends that exposing the target to the front end will solve this and some other problems. *Christian.* **July '02: Unknown**

[**C**] 18. Ensure that the front end rejects a named calling convention that is not known to the back end. *Christian.* **July '02: Unknown**

[**A**] 19. Targets provide code that spills and reloads registers. This code must respect the machine invariant, but currently does not. Machine specific spilling and reloading code must be defined. *John or Norman.* **July '02: Open**

(This code might be tied to the question how small values of 8 or 16 bit widths are handled. Are these values ever spilled?)

[**B**−] 20. Support `also cuts to k` on `cut to`. Here's an example use: **July '02: Done**

```
import put_ready, get_ready;
yield () {
  bits32 r;
  put_ready(k);
  r = get_ready();
  cut to r() also cuts to k;
  continuation k:
  return;
}
```

Christian

[**B**−] 21. The *semantics of the assembler interface* needs to be defined clearly. The problem stems for an anticipated use for both textual assemblers and binary emitters. Especially the role of symbols is not clear and should be explained with respect to the better known textual assemblers. The first step is to *review the assembly interface*, which should happen after the redesign of the flow graph. *Norman.* **July '02: Unknown**

22. In the assembler interface, there is a problem with the functions used to emit initialized data—some ambiguity about how they are supposed to be used. Christian will identify the problem more precisely so it can be fixed. *Christian and Norman.* **July '02: Open**

[**C**] 23. The semantics of `fcmp` should be specified precisely. In particular, it is not known how the results map to two-bit values. If the answer is target-specific, then the `Target` module should be adapted accordingly. This item has relatively low priority because the new floating-point comparisons are more convenient than `fcmp`. *?????* **July '02: Open**

[**B**+] 24. Alter the `lcc` back end to use the new floating-point comparisons. *????* **July '02: Open**

25. The function `proc` that controls the translation of a procedure in module `Ast3ir` must be broken up and cleaned up. The proper time to do this is *after* the flow-graph revision. *Christian* **July '02: In progress**

[**A**−] 26. The treatment of the `*` operator in C-- source code must be corrected. For purposes of type checking, this operator can be treated as a single polymorphic operator of that takes two operands of width n and returns a result also of width n. For purposes of translation, however, the front end must generate a composition of operators: first apply `mulu` to produce a result of size $2n$, then apply `lobits`n to the result.[1] **July '02: Unknown**

For some reason, it appears to be difficult to handle this problem in the front end. In that case, it will be reasonable to invent a new RTL

[1]We need to check the manual to see whether `*` abbreviates a signed or an unsigned multiply.

operator, a truncating multiply, perhaps to be called `mulu_trunc`.
Christian.

[**A**] 27. The algorithm that linearizes the control-flow graph does not work properly. It fails to detect the difference between nodes that fall into a label and those that jump to a label. A probable fix to the problem is always to jump to a label and never to fall into it. When the design of the CFG is reconsidered, this problem should be addressed, too. *I have worked around the problem by never falling into a label, but using an explicit* `goto`*. This revealed some more problems with the CFG design: it is neccessary to create new nodes inside the CFG and thus no target-specific knowledge should be needed. I had to give up the polymorphism of the CFG which wasn't working anyway. –CL (Tue May 7 16:44:58 EDT 2002).* **July '02: In progress**

[**C**] 28. We need support for IEEE 754 floating-point literals. Mapping a literal to a bit vector is quite complicated and involved (see papers from PLDI'96), but there should be reliable C code on the net to solve this problem, so that all we have to do is integrate it into the compiler. David Gay or Will Clinger may know where to find such code.
Christian. **July '02: Open**

[**C**] 29. We need a policy decision about whether to support floating-point operations at compile time. Someone will have to investigate to see if there is suitable "soft floating-point" code available. **July '02: Open**

In the process, we should remove the suspect code from `rtl/unit64p.c`. This code provides casting of 64 bit floating point values to 64 bit integer values, but it is platform sensitive and maybe the wrong way.
Christian.

[**D**] 30. Implement support for string literals (low priority). In order to support Unicode, we might need changes to the C-- language definition. **July '02: Open**

[**D**] 31. Identify a principled way to reject programs that a particular back end cannot successfully compile. Reasons for rejection might include use of C-- variables or operators at unsupported widths. **July '02: Open**

This problem is more important than a priority of **D** may suggest, but it is a far-future item. We will have a much better idea how to handle this problem after we have built a few back ends.

[**B**−] 32. Ensure that *global-variable declarations are consistent* across modules. **July '02: Open**

When multiple compilation units share their global variables one of them must define them, while the other import them on an assembler level. No mechanism exists to mark the "main" compilation unit that defines all global variables.

N.B. There is already code to compute a suitable MD5 signature. Primarily what needs to be done is arrange suitable import/export and implement a command-line option that designates the "main" compilation unit.

[**D**] 33. Implement `switch`. **July '02: Open**

[**C**] 34. Export substitution of late compile-time constants and constant folding as finalizing step of a procedure translation to Lua. Currently these steps are hard-coded into the `freeze` function in module `Ast3IR`. **July '02: Unknown**

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. See also the reasons for this in module `Elab.Const.Eval`: **July '02: Open**

> *Several reasons: We would need a translation from AST to* `Rtl.env`*. The existing one in* `ast3ir` *is written against the* `Fenv.Clean` *interface, but we need one written against the* `Fenv.Dirty` *interface. The translation of an application needs the widths of its arguments. Therefore type-checking using* `Expcheck` *would be done twice: once before translation and then again during translation. Detailed error messages are simpler to generate with the existing approach.*

Old problems that have been solved After an RTL has left the register allocator, it may still contain late compile-time constants. After these are substituted by bit vectors, the RTL must undergo a constant folding process to make sure it respects the machine invariant. Such a constant folding routine currently does not exist. Since the front end also needs to evaluate compile-time and link-time RTL expression, one general RTL evaluation mechanism would be appropriate. The current module `rtleval` can only handle compile-time and link-time RTLs. *I hope this problem is solved in* `Rtleval2`. *–CL*

Items whose status is unclear It is not clear whether this is a problem that needs immediate attention or a suggestion for refactoring.

- Relocatable addresses have undergone several changes in their representation. There are still two slightly different representations used in the compiler that wait to be unified: `Reloc` and one in `Sledlib`.

Obsolete items

- Integration of machine generated code. Basically it does not work because machine generated code is not flexible enough to handle temporaries and late compile-time constants. Adjusting the machine generated code manually is time-consuming and error prone. Machine generated code provides RTL constructors for machine instructions, a recognizer for machine instructions in RTL form, and an emitter of assembly code. *This item will be delayed until after we have a working compiler.*

Suggestions for refactoring The main overall problem is complexity. The development of the compiler will benefit from trying to cut down the number of modules, abstractions, and moving parts in general.

- There is some confusion in the way C-- primitive operators are translated to RTL operators. C-- operators are monomorphic in their result type, whereas RTL operators are not. A polymorphic RTL operator takes an additional argument compared to its monomorphic C-- counter part. This argument should be supplied first to take advantage of currying: the a polymorphic RTL operator supplied with its first argument implements the monomorphic C-- operator. Currently the additional argument is the last argument. *I hope this problem is solved in* `Rtlop`. *The above explanation of operator specialization is not entirely correct. The specializing parameter is not a formal argument of a primitive, but part of its* width list. *–CL*
- The semantics of assembler symbols have leaked into the front end, the fat environment, and the CFG. Symbols are handed out by an assembler which means, a reference to the current assembler must be available to create new symbols. Because the CFG does not include

such a reference, it is difficult to create `goto` nodes (during CFG manipulation) outside the `Ast3IR` module. This is related both to the point above, and the unclear semantics of the assembler interface, and symbols in particular. I would like to avoid dealing with target specific symbols as much as possible and would like to delegate their creation into the last phases of compilation. Upon further reflection I now believe that the *construction* of a CFG should not require any machine-specific knowledge to allow for the easy implementation of CFG transformations.