

Prototype Compiler Internals[†]

Preliminary Notes

Jonathan Shapiro, Ph.D.
The EROS Group, LLC

April 5, 2010

Abstract

This note describes the general structure of the BitC prototype compiler, including documentation of what each pass does. It is intended as a (very) rough guide to the structure and assumptions of each phase.

1 Introduction

As I start to wrap my head around the compiler once again, I'm gathering notes on the internals. Hopefully these will help me (or someone else) maintain the beast later. At the moment, I'm mainly after wrapping my head around the type system again.

2 Structure of the Compiler

The BitC compiler acts almost entirely as a sequence of BitC->BitC transformations. It is a whole-program compiler that operates in several major phases:

2.1 Front-End Processing

The prelude is imported (which ultimately calls `CompileFile` on standard library source code). `CompileFile` is then called on each unit of compilation named on the command line in turn. The effect of `CompileFile` is to parse the input file and then invoke the remaining compile phases on the resulting AST. The parsing pass is recursively re-entrant: an import causes the imported unit to be processed immediately as a consequence of import.

The import/compile mechanism records each result module as processing is started, and checks at import time to see if the desired module has already been loaded. This ensures that each module processed by the compiler is processed only once, and serves as a guard against cyclic import.

The passes applied by the front end are (file `pass.def`):

1. **parse** The input unit of compilation is parsed, and an AST is constructed. `import` forms are processed eagerly and recursively.
2. **defrepr** The semantics of `defrepr` are essentially those of unions, with special handling for the tag check. This pass syntactically rewrites each `defrepr` form into a corresponding union form, marking the resulting AST as originating from a `defrepr` form.

No `defrepr` or `declrepr` forms survive this pass. The interesting processing here concerns the handling of *where* clauses.

[†] Copyright © 2010, Jonathan S. Shapiro.

3. **begin simplification** The front end is moderately aggressive about introducing `begin` forms. This pass removes the ones that are unnecessary, which serves to reduce nesting and processing overheads all the way through the compiler.
4. **Method Declaration Insertion** For any structure or object that contains a field of method type, this pass inserts a corresponding `proclaim` form for the method's definition. This ensures that when the method is actually defined it is type-compatible with the original declaration that lived inside the struct/object. It also ensures that the defining form passes symbol resolution.
5. **Symbol Resolution** A pass is made over the AST constructing a symbol environment and ensuring that all symbol references resolve.
6. **Type Checking** The type inference and checking pass, which is where most of the "interesting" stuff in the current BitC implementation happens.

In the current implementation, the type inference and checking pass checks the compatibility of destination and source in assignments, but does not validate location semantics.

7. **Location Semantics Check** This pass checks location semantics.

The intent of this pass is to ensure that assignments proceed only to valid locations – which is to say, an "l-value". Or to put it another way, the goal was to ensure that intermediate expression results (which are ephemeral) would not be targeted by assignments whose result would immediately be lost, such as:

```
(set! (+ 3 5) 2)
```

Note that there is no error here from the standpoint of type checking or semantics, but the operation is complete nonsense, because the result is immediately lost. Which is perhaps not so bad here, but consider:

```
(set! (array-ref (make-array a 300 3:int32) 1) 5)
```

Where we will go and initialize a complete array, only to discover that the whole thing is unreferenced.

There are several bugs in this pass at the moment. Need to discuss them.

8. **defrepr Consistency Check** This pass checks that the `where` clauses of `defrepr` forms are valid, and that all `defrepr` forms are fully concrete. Note that this pass cannot be done until after type checking, because we need the type checker to have assigned types at all ASTs before we run this.
9. **noalloc Check** This is a vestigial, half-implemented pass that is essentially useless in its current form and should be changed to simply complain when `noalloc` is required, because the check isn't actually being done properly.
10. **Initialization Check** This pass ensures that initialization ordering restrictions are honored. It attempts to ensure that no use-occurrence of an identifier appears before that identifier has been "observably defined". The intent is to impose a lattice ordering on initialization so that the state of globals is unambiguously defined at the first instruction of the `main` procedure.

The current checking pass isn't right.

11. **Instance Lambda Hoisting** Many (perhaps most) type class instances define their corresponding methods using "immediate" lambda forms. The instantiator can't currently handle that. This pass "hoists" the immediate lambdas into global procedures, and replaces the original lambdas with the names of those global procedures.

The current implementation does not correctly hoist constraints.

The internal result of this pass is a `UocInfo` object for each module, which contains references to the parsed and decorated AST, the top-level lexical environment for symbols, the top-level lexical environment for types, and the "instance environment".

If a module passes the front end, it has passed all checks. In principle, no errors resulting from type checking or semantics program should be possible in the mid-end or the whole-program pass. It is, of course, possible that there will be unresolved symbols due to missing source units.

2.2 Backend Directed Per-UoC Processing

Once all ASTs have been processed by the front end, further processing depends on the choice of backend (more precisely: the output target). The prototype compiler currently has several back-ends (files `backend.cxx`, `backend.hxx`):

- **exe** Generates whole-program executables. The executable is produced by compiling the C code generated by the `c` back-end.
This backend operates whole-program.
- **c** Generates C code corresponding to a whole program (primarily for debugging).
This backend operates whole-program.
- **h** Produces a C header file for use by C runtime functions to ensure compatibility with the type declarations created by the BitC compiler.
This backend operates whole-program.
- **bito** Produces a BitC "object" file, which is merely an aggregation of checked input units into a single file, multi-module representation (for now, a `.bito` file is actually source code). This is a disgusting hack, but it allows us to emulate the behavior of static link lines and libraries until we can do run-time specialization.
This backend target does not execute the gather phase or the whole-program phase.
Note that this pass relies on the correctness of the AST pretty-print implementation to re-emit the program originally presented by the user. The BitC pretty printer is not merely a convenience tool.
- **others** Various XML-ish dump formats for debugging purposes. These are mostly obsolete.

The front-end processing is identical for all back ends. The selected back end optionally defines a backend-specific phase that can be run on each unit of compilation after the front end has run.

If a per-UoC phase is defined by the back end, it is first run on the interface modules and then on the source modules.

2.3 Pre-Gather Phase

Once all modules have been processed by the front end, the backend is given an opportunity to run a single pass over the entire input processed to this point. The distinction between this pass and the per-UoC pass defined by the backend is that the per-UoC pass is executed on each module/UoC in turn, and is not designed to retain state from one UoC to the next. The pre-gather backend pass directs its own iteration over the UoCs, and is consequently able to retain state across them.

At the moment, the only backend that defines a pre-gather phase is the `.bito` emitter. This phase also indicates (through a flag in its specification) that it does not actually want the gather phase or the whole program phase to be run.

A cleanup should be considered here, which is to implement a generic "all modules" helper function that in turn calls the per-module function. This would encapsulate some of the appearance of complexity from the command driver without leading to replicated code.

2.4 Instantiation (Gather) Phase

Once all per-UoC processing has been done, we reach the instantiation phase of the compiler. This part of the compiler is in flux between a static compiler and an interactive top level. The current implementation only handles static compilation, but it does so using the same incremental mechanisms that will eventually be used by the interactive top level. The following description applies primarily to the static compiler.

What we are planning to do here is to create a single, self-contained module (and unit of compilation) that contains every reachable procedure, type, and constant from the original program. All procedures in the unified UoC are fully

concretized by the instantiator - no type variables appear. Procedure names in this UoC are mangled to reflect both their specialized type signature and their originating source module. Once this module is built, we will do back-end processing on it, ultimately arriving at a module that we can emit directly to C.

The process of copying things into the "grand UoC" is driven incrementally from the initial entry points. For each entry point:

- The current entry point will be copied to the Grand UoC. Make it the "current definition". If it has already been migrated, declare that we are done.

In order to be suitable, an entry point must already have a fully concrete type. The intuition is that this is something like `main`, where a concretely realized label is going to be invoked as an external entry point to the program.

Note that *because* instantiation proceeds from a fully concrete procedure, any type variables that *cannot* be instantiated (recursively) from the initializing expression/form will never be used, and will be instantiated to unit.

- Walk the body of the current definition, type, or variable definition. For each procedure, type, or variable definition that is referenced by the definition of the current form, resolve any type variables for that use-case, determine what (mangled) name the resulting concrete instance will have, and if no copy of that instance has been performed, instantiate it and copy it to the Grand UoC.

Note that the procedure is incremental, and proceeds one entry point at a time. A small detail glossed over above is that new definitions are instantiated into a transient unit of compilation and merged into the Grand UoC when instantiation has fully completed. When we implement an interactive top level, the Grand UoC will constitute everything instantiated prior to the current top-level form, and the transient UoC will handle the instantiation necessary to evaluate the current top-level expression. Only the new forms are passed through whole-program processing (see below).

2.5 Whole-Program Processing

Once the Grand UoC has been constructed, we perform back-end whole-program processing on it. The objective of this processing is to reduce the program (which is still a legal BitC program) down to something that can be emitted directly as C code. That is: to reconcile the differences between the semantics required by BitC and the semantics expressible in C. Three transforms are performed at the whole-program layer:

1. **Closure Conversion** Inner procedures are lifted, closure records are constructed, and wrapper procedures are introduced.
2. **Tail-Call Marking** The "call" ASTs for procedures that can legally be implemented as tail calls are marked for special handling. This signals to the C code emitter that these calls should be performed using labeled `gotos`. I don't recall at the moment whether the target procedures are also marked.
3. **SSA Transformation** An SSA-like transform is performed. This is an enormously ugly requirement, and we had hoped that it would not be necessary. Unfortunately, GCC expression blocks cannot contain `goto`, so we were forced to introduce an SSA-like transformation.

The SSA pass is the one pass that does *not* perform a strict BitC->BitC transformation. There are cases where it generates a "fake" `let` form for the sake of symbol resolution, whose initializers are missing.

References

- [1] **NOT USED** —: American National Standard for Information Systems, Programming Language C ANSI X3.159-1999, 2000.
- [2] **NOT USED** —: *IEEE Standard for Binary Floating-Point Arithmetic*, 1985, ANSI/IEEE Standard 754-1985.

- [3] **NOT USED** —: *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, 1987, ANSI/IEEE Standard 854-1987.
- [4] **NOT USED** Andrew W. Appel and Zhong Shao. “An Empirical Study of Stack vs. Heap Cost for Languages with Closures.” *Journal of Functional Programming*, **1**(1), January 1993
- [5] **NOT USED** Joel F. Bartlett. *Mostly-Copying Garbage Collection Picks Up Generations and C++*. WRL Technical Note TN-12. Digital Equipment Corporation, 1989.
- [6] **NOT USED** Jacques Garrigue. “Relaxing the Value Restriction.” *Proc. International Symposium on Functional and Logic Programming*. 2004.
- [7] **NOT USED** Anita K. Jones. *Protection in Programmed Systems*, Doctoral Dissertation, Department of Computer Science, Carnegie-Mellon University, June 1973.
- [8] **NOT USED** Mark Jones. “Type Classes With Functional Dependencies.” *Proc. 9th European Symposium on Programming* (ESOP 2000). Berlin, Germany. March 2000. Springer-Verlag Lecture Notes in Computer Science 1782.
- [9] **NOT USED** M. Kaufmann, J. S. Moore. *Computer Aided Reasoning: An Approach*, Kluwer Academic Publishers, 2000.
- [10] **NOT USED** Richard Kelsey, William Clinger, and Jonathan Rees (Ed.) *Revised⁵ Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, **33**(9), pp 26–76, 1998.
- [11] **NOT USED** David MacQueen, “Modules for Standard ML.” *Proc. 1984 ACM Conference on LISP and Functional Programming*, pp. 198–207, 1984.
- [12] **NOT USED** Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised* The MIT Press, May 1997.
- [13] **NOT USED** J. S. Shapiro, J. M. Smith, and D. J. Farber. “EROS, A Fast Capability System” *Proc. 17th ACM Symposium on Operating Systems Principles*. Dec 1999, pp. 170–185. Kiawah Island Resort, SC, USA.
- [14] **NOT USED** Jeffrey Mark Siskind. “*Flow-Directed Lightweight Closure Conversion*” Technical Report 99-109R, NEC Research Institute, Inc., Dec. 1999.
- [15] **NOT USED** Zhong Shao and Andrew W. Appel. “Space-Efficient Closure Representations.” *Proc. LISP and Functional Programming*, pp. 150-161, 1994.
- [16] **NOT USED** N. Wirth and K. Jensen. *Pascal: User Manual and Report*, 3rd Edition, Springer-Verlag, 1988