Reprinted from the

Proceedings of the GCC Developers' Summit

July 18th–20th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Ian Lance Taylor, *Google*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Multi-Stage Construction of a Global Static Analyzer

(in GLOBALGCC project)

Basile Starynkevitch
CEA LIST
(Software Reliability Lab.)
basile@starynkevitch.net or basile.starynkevitch@cea.fr

Abstract

We describe ongoing work about global static analysis for GCC4 within the GlobalGCC European project, funded thru the ITEA Programme.

The aim of this work is to provide global (whole program) static analysis, notably based upon abstract interpretation and knowledge based techniques, within the GCC compiler, targeted for analysis of medium sized C, Fortran or C++ programs. This will facilitate the adoption of GCC in the area of safety-critical software development, by providing features found in a few expensive commercial tools (PolySpace, AbsInt) or research prototypes (Astree). In this perspective, the emphasis is on the quality of analysis, at the expense of much bigger compilation times, without sacrificing scalability. Such analysis can be used for several purposes: statically compute some interesting properties of the program at most control points (possibly reporting them the user); provide clever, contextual, warnings about possible hazards in the user program (null pointer dereferences, zero divide, conversion loss, out of bound array access, ...) while avoiding too much false alarms; enable additional optimisations, like conditional contextual constant folding, C++ method call devirtualization, an other contextual optimizations.

The compiler's rich program manipulation infrastructure facilitates the development of these advanced analysis capabilities.

To facilitate the development high-level semantical analyses, a domain specific language has been developped, and is translated (thru C) into dynamically loaded code. It uses the Parma Polyhedra Library (also used in the GRAPHITE project) for relational analysis on scalars and gives more expressivity to develop analaysis algorithms. It permits multi-staged generation of the specific analysis tailored to the analyzed source code. Presenting this work at the 2007 GCC Summit will allow us to stress the importance of all outputs of the compiler, not only object-code, and to expose the complementary contribution of static analyses and dynamic/instrumentation approaches like mudflap.

Warning

This paper describes some *work in progress*.[1] A more up-to-date report, and a snapshot of the source code, should be available on the author's web page `http://starynkevitch.net/Basile/` or on `http://ggcc.info/`.

1 Interest and Issues of Global Static Analysis

The current GCC compiler[2] is mostly used to transform a source code file into some object form, containing suitably represented processor instructions. For this very common use, performance of the compiler and of the generated code are expected (but are sometimes in tension, requiring carefully tuned trade-offs).

However, GCC also provides an interesting infrastructure and internal code representations, usable for other purposes. In particular, static code analysis (deep inspection and processing of an analyzed source program, without paying much attention to machine code, or to its execution) is also possible.

[1] Within the GlobalGCC project, ITEA [Information Technology for European Advancement] programme, partly funded by MINEFI (French Ministry of Economy, Finance, and Industry) and other public authorities. Like in every GCC contribution, code copyright has been transfered to FSF.

[2] E.g., the trunk branch of SVN rev. 124285.

1.1 Static analysis overview

Static analysis tools are already used by some software industries, notably in safety-critical (aerospace, automotive, nuclear, medical, ...) applications. Success of commercial (but expensive) tools (like Absint, Polyspace, ...)[3] show that some niche market exists for these techniques. Research prototypes like Astrée [6, 7] or TWO [9] suggest that it should be worthwhile to incorporate some of their ideas into GCC. Most such tools are significantly slower than ordinary compilers, because they usually process a whole program source, using sophisticated (but expensive) representations, but are still used profitably. Safer dialects of C have been proposed [10, 13].

Static analysers are not a panacea: they compute *approximate* properties[4] on the source code (such as properties or relations on the values of the program variables during any of its execution), but they should always terminate their analysis, even for buggy analyzed programs.

Of course, many algorithms used in traditional optimized compilation can be viewed as a particular form of static analysis (usually restricted to a single block, function or compilation unit). And `gcc 4` also permit (in a limited way) whole program compilation thru the `-fwhole-program` compile flag.[5] The *Link Time Optimization* effort within GCC also targets whole program optimization, by encoding some GIMPLE related internal representation in DWARF debugging format inside object files.

1.2 The GLOBALGCC project

Because of this interest in static analysis and in the free GNU Compiler Collection, a consortium of european industial corporations and research labs proposed the GLOBALGCC project[6] which aims to extend the GCC compiler (on Unix or Linux hosts) using global static analysis techniques, lead by Mandriva. The static analysis will work on the GIMPLE and GIMPLE/SSA[7] [15] internal representation[s] of GCC (hence re-using all existing GCC front-ends). It is global, because a whole source program (of several compilation units) should be processed. Such static analysis techniques should enable:

1. Global *program-wide optimisation*, because the properties inferred at a given call site may be propagated to avoid useless computations in the program; for example if on a given call site and calling context the static analyser determined that some pointer is not null, this information can be used to optimise further on, particularize inlinings, etc.

2. *Hazard detections*, that are warnings (for the developer using GLOBALGCC) about possible *threats* like: if a function `f()` was called by `g()` called by `h(x)` with `x>0` then at `foo.c:456` there is a possible zero-divide fault. The challenge is to reduce the number of (stricto sensu unavoidable) false positive alarms.

3. Later, *coding rules validation* can be considered. This means defining a formalism to express some coding rules, and use the statically analyzed properties to partly validate these rules.

It is expected that such static analysis techniques should be computationally expensive (more than ten times slower than a traditional, `-O3` optimized, compilation build of the analyzed program). The `mygcc` project[8] provides a complementary, but very useful, alternative: simple, quick, but useful static analysis based upon sophisticated pattern matching [18]. The static analysis considered here put emphasis on quality of analysis, at the cost of much bigger compilation (i.e., analysis) time.

1.3 Abstract interpreters

Abstract interpretation (pioneered by P. and R. Cousot) [5, 4] provides a conceptual framework for designing such static analyzers. The guiding idea is to abstract on program variables' values with lattices (e.g., considering intervals instead of numbers or their relations, e.g.,

[3] See `http://www.absint.com/` and `http://www.polyspace.com/` and `http://www.mathworks.com/`.

[4] Always computing exact properties would solve the halting problem, which is impossible.

[5] A Google code search reveals that this flag is almost never used.

[6] See `http://ggcc.info/` and `http://gcc.gnu.org/ml/gcc/2006-10/msg00676.html`

[7] GIMPLE is the middle-end, source-language and target-system neutral, internal representations of (normalized) trees within GCC4; GIMPLE/SSA is its Single Static Assigment form.

[8] See `http://mygcc.free.fr/`

linear inequalities) and to "interpret" on every control flow of the program. At some control points, abstract values are over-approximated. Such narrowings (and widenings)[9] ensure that the analysis always terminate (hence avoiding long loops in the analyzer, at the expense of precision): the computed abstract values can be $\top$ (top, representing any value), $\bot$ (bottom, impossible or unreachable) or other elements of the lattice.

Abstract interpreters[10] have rather complex algorithmic behavior (like concrete interpreters): it is not easy to predict if a given abstract value will be further used, or what is the exact time or space complexity of the analysis. This contrasts with most of current GCC optimization passes, which mostly update or rewrite GIMPLE trees, and whose other data has a life internal to the pass.

Lattices for simple (e.g., numerical scalar) variables in simple (e.g., imperative like, without calls) mini-languages routinely exist. Two free lattice libraries have been considered: APRON[11] and the Parma Polyhedra Library (PPL)[12] [2, 1], which was preferred (it is also used in the Graphite branch of GCC). More complex analyzers and lattices should be built above such primitives lattices to abstract other control structures and other data, in particular pointers to heap data structures [12] or arrays [17]. The result of an abstract interpretation is conceptually, at every control point of the analyzed program, an abstract value of program variables; for simple integer programs with variables $v_1 \dots v_n$, it could be an interval $v_k \in I_k = [a_k; b_k]$ for each variable v_k, or a set of linear inequalities (i.e. polyhedra) $\sum_k c_{i,k} v_k \leq l_i$ between variables, etc. For real programs, pointers or data structures are also abstracted by shapes or graphs.

The result ρ of an abstract interpretation depends upon the analyzed program π, the initial conditions α and of course the lattice: $\rho = \phi(\pi, \alpha)$. ρ is a (big) decoration of the syntax tree. Since such abstract interpretations are costly, and because the analyzed program is fixed for a given interpretation, it could be worthwhile to specialize (part of) the analysis for the given program. More pedantically, it may be interesting to partially evaluate $\phi(\pi, \bullet)$. Pragmatically, development of abstract interpreters should take profit of *multi-staged or meta-programming techniques*, i.e., dynamic generation of specialized code during analysis, possibly with introspection [14] to guide widening. The intuition is to generate specialized code which does the analysis of the only particular program which is analyzed.

However, care should be taken to avoid dynamically generating an analyzing code much bigger than the analyzed source program. Practically, a domain specific lispy like language capable of runtime code generation, is deemed useful.

2 A Multi-stagable run-time infrastructure

A run-time infrastructure has been developped (above existing GCC code) to take into account the specific needs of abstract interpreters for static analysis, as considered in §1.3 above. It is tentatively called *basilys* (base for abstract interpretative analysis).

2.1 Compiler Probe facility

Since our static analysis are expected to run much longer than a traditionnal compilation, and because it should produce a lot of intermediate results (abstract values at many control points) which are useful both to the expert user of the analyzer and to its developers. To avoid just generating huge dump files (only usable after analysis ended), a compiler probe facility has been proposed[13] and could be useful to other GCC developers. It works only on some Unix host systems (e.g. Linux[14]), by (optionally—at configuration and at compilation time) running a separate process (the probe, with a GTK based sample graphical implementation in `contrib/simple-probe.c`) which communicate with the GCC process on asynchronous channels (pipes) using textual protocols (requests from probe to compiler, replies from compiler to probe). On the GCC compiler side, frequent calls (dozens per second) to `comprobe_check("reason-msg")` are expected. This is a macro which expands to the test of `comprobe_interrupted`, an almost always zeroed volatile variable. Should a message come from the probe, the `comprobe_interrupted` flag becomes set, and then

[9] We view narrowing and widening operations as necessary heuristics to ensure that analysis terminate rather quickly, even when the analyzed program loops... Simple reflexive or introspective techniques could be useful here.

[10] i.e., abstract interpretation based static analyzers

[11] `http://apron.cri.ensmp.fr/` with LGPL license, wrapping other libraries.

[12] `http://www.cs.unipr.it/ppl/` with GPL license, self-contained.

[13] See patch `http://gcc.gnu.org/ml/gcc-patches/2007-01/msg01278.html`

[14] It needs `SIGIO`, `F_SETOWN`, `O_NONBLOCK`, and `select`.

`comprobe_handle_probe` is called and handles the incoming requests, sometimes by sending appropriate reply messages to the probe. Above that dirty trick, information points (in the compiled source program) are managed and display routines are callable from them on request, to show only information pertinent to (or near of) a given control point.

Therefore, the compiler probe enables giving feedback to the user during our static analysis (i.e., compilation). It is implemented in more than 3KLOC (thousand of source code lines).

2.2 Dynamic run-time

Given the complex usage pattern of abstract values[15] automatic memory management techiques are required (almost every abstract interpreter implementation we know of uses garbage collection techniques [11]).

The current GCC compiler provide a limited form of garbage collection (frowned upon by some developers). The GGC garbage collector is precise, of mark and sweep kind, and deals only with explicitly declared[16] data structures and pointer (global or static) variables, but does not manage local pointers on the compiler call stack, which is lost unless saved in globals; it should be explicitly called. This is acceptable for GGC purpose of managing rarely dying data (mostly GIMPLE trees), shared across several passes. GGC collection needs to scan all the heap, and works better when most of the data remains alive.

Abstract interpreters are a different kind of beast: they allocate a lot of data (the abstract values) and most of it is temporary and quickly fade away (but is difficult to delete explicitly). For such scenarii, other GC schemes are better suited, such as generational copying collectors,[17] detailed below.

Hence, a copying generational garbage collector has been implemented for our abstract interpreters. It is copying generational for young data, but mark and sweep for old data. It works by allocating (with a quick current pointer incrementation) inside a birth zone (typically 4Mwords), without any additional space overhead. When this birth zone is full, a minor copying collection occurs: it scans all the local pointer variables on the stack (and some globals) and copy the live data[18] into the GGC managed heap. Then, the entire birth zone can be freed at once (without spending time on each individual dead value) and suitably quickly reallocated. When a suitable threshold (e.g., 64Mwords) of cumulated allocations occurred in the birth region, a full (or major) garbage collection is triggered: all the local pointers are saved into some GGC data, and the GGC mark and sweep collector is called. Updated pointers (i.e., a write barrier) from new to old are managed on a store list (on the other end of birth region) with a small caching hashtable (for frequently touched pointers).

Generational garbage collectors are uncommon in (portable) C libraries, because they require (for GC-ed data and pointers) a specific, cumbersome, coding style:

- allocation of objects is usually fast (pointer incrementation and test) but may trigger a garbage collection.
- every local pointer should be explicitly known. Our local pointers are all inside a call frame structure declared nearly as `struct frame_st {unsigned nbvar; struct closure_st *clos; struct frame_st *prev; void *varp[nbvar];} curfra;` where `nbvar` is the number of local pointers variables (stored in `varp`), `clos` points to the current closure, and `prev` chains to the previous frame.
- each function should have an explicit prologue and epilogue to manage the singly linked list[19] of such frames, which should be initially cleared.
- no nested function calls are permitted: α = `foo(`β`, bar(`γ`));` should become τ = `bar(`γ`);` α = `foo(`β`,` τ`);`, concretely like `curfra.varp[6] = foo(curfra.varp[3]); curfra.varp[1] = foo(curfra.varp[2], curfra.varp[6]);`
- every update inside such a value (excluding initialization) should be notified (write barrier), and can trigger a minor garbage collection, which can move every local pointer.

[15] It would be very difficult for the developer of an abstract interpreter to know when to free an abstract value.

[16] Thru the `GTY` marker used by the `gengtype` generator.

[17] Like in most efficient implementations of functional programming languages—Ocaml, Haskell, ...

[18] Copying GCs are also rumored to improve data cache locality.

[19] Inspecting the list of call frames provide a simple way of reflexive introspection, notably thru the `clos` fields.

- special care has to be taken for values which have to be individually destroyed (e.g., when containing PPL pointers). Explicit young and old lists of such special values are maintained, the `GTY(mark_hook)` marker is used, and they are explicitly destroyed in our garbage collector.
- a union of all our garbage collected values should be known, and each value should be discriminable inside.

Above constraints are easier to follow in generated code than in human-written one.

Concretely, our *basilys* (low-level) memory values starts each with a discriminating pointer, and are one of:

- objects (see below), used for discriminants, high-level abstract values, analyzer "source" code, etc.
- single or multiple boxes (of basilys pointers),
- boxed GCC stuff, like trees, edges, basic blocks, etc. For example `struct basilystree_st GTY(()) { basilysobject_ptr_t discr; tree val; };`
- (immutable) strings and (updatable) string buffers
- analyser's closures and routines
- pairs and triples (for lists)
- boxed (long) integers
- hash tables (or object maps) whose keys are objects, and values are *basilys* pointers
- hash tables with tree (resp. edge, basic blocks, ...) keys (tree maps, ...); these are used to associate abstract values to tree control points.
- special (destroyable) values for PPL, etc.

Our object values contain a discriminating class, an hashcode, a number, a length, and the object's variable (i.e. instance slots) array:

```
struct basilysobject_st GTY(()) {
  basilysobject_ptr_t obj_class;
  unsigned obj_hash;
  unsigned short obj_num, obj_len;
#define object_magic obj_num
  basilys_ptr_t*
    GTY((length("%h.obj_len")))
      obj_vartab;};};
```

Each value starts with a discriminant (`discr` or `obj_class` in objects). This is a pointer to an object, whose `obj_num` is used as a discriminating magic number, in particular for GGC marking (on full collections). Hence the union of our values is declared as:

```
typedef union basilys_un* basilys_ptr_t;
union basilys_un
  GTY ((desc("%0.u_discr->object_magic"))) {
    basilysobject_ptr_t
      GTY((skip)) u_discr;
    struct basilysobject_st
      GTY((tag("OBMAG_OBJECT"))) u_object;
    struct basilysbox_st
      GTY((tag("OBMAG_BOX"))) u_box;
    struct basilystree_st
      GTY((tag("OBMAG_TREE"))) u_tree;
  /* .... etc. */
};
```

It is expected that these values (runtime types) are sufficient building bricks for most analyzers. Adding new such values is quite easy (i.e. for other GCC data like loops).

We have also considered using an existing runtime (e.g. Ocaml or MetaOcaml,[20] Python, Guile, Ruby, SBCL, ...) but this is not practical, because the current GCC interface (GIMPLE tree based) is quite low level, needs to be adressed in C (thru numerous macros), avoiding the overhead of generic runtime machineries[21]; the practical way to efficiently interface all of GCC internals is indeed to generate specialized code, tightly dependent upon GCC data structures. Hence using a foreign runtime would create a significant impedance mismatch.

Dynamic runtime code generation is possible by generating (during static analysis) a C source file, compiling it as a shared object, and dynamically loading it -thru the `libtool` dynamic loader (a portable wrapper around `dlopen`). Such shared objects are never released (no `dlclose`).

2.3 Basilys objects and closures

Our runtime offers lisp-y closures, which contain a routine value and the closed values. A closure is applied

[20] See `http://metaocaml.org/`. It would have been very sexy if we could in particular concisely write Meta-Ocaml-like patterns for matching GCC `trees`, but runtime considerations make that impossible.

[21] It would be very inefficient to access the son of a GIMPLE tree by some complex routine call.

to a sequence of arguments which can be *basilys* pointers or (plain unboxed) scalars (e.g. `long` integers, GCC `trees`, ...). This application produces a primary result pointer, and secondary results[22] (either pointers or unboxed scalars). The routine value contains the C code pointer and any additional value the code depends upon.[23] The called C routine gets as C arguments the closure, the first two arguments, and the other arguments and results (as arrays of unions) with a descriptive string.

Basilys objects are organized à la ObjVlisp [3], with a single inheritance, meta-class based organization providing mono-dispatched methods and quick *is instance of* and *is subclass of* tests. Classes, slots, selectors, fields are themselves objects. Discriminants of non-object values (like boxed integers, single or multiple boxes, boxed trees, ...) are also objects, and can dispatch messages. Therefore, messages can be sent to any basilys value. Every discriminant or class contains a dictionnary of methods (or nil) and a sending closure (or nil). Message dispatch of selector σ and receiver ρ with additional arguments $\alpha_1 \ldots$ is done as follow:

1. get the discriminant (or class) δ of ρ
2. get the method dictionnary μ in δ
3. find (if any) the closure κ associated to σ in μ, if it is a closure, go to step 7
4. get the sending closure ν in δ, if any, otherwise go to step 6
5. apply the sending closure ν to (ρ, σ), this should give a closure into κ (otherwise error)[24], and go to 7
6. without closure κ or sending closure ν, if the receiver ρ is an object, get the parent class in δ and put it in δ an repeat step 2. If the receiver ρ is not an object (but a non object value, like a boxed tree, etc.) we are stuck in error.
7. a closure κ has been found for the method send; we apply it to $(\rho, \alpha_1 \ldots)$ hence getting the result of the send

[22] secondary results, like in Common Lisp, may be ignored, but are useful, e.g. to return some abstract value with an additional item indicating its completeness.

[23] E.g. if the code contains the hash code of some objects, these objects should be kept in the routine.

[24] It is a hook to implement more dynamic situations, e.g. handling error like unknown selectors, or just generating a particular code on demand.

Our core classes include `ROOT` the topmost class of all objects, `PROPED` subclass of (noted $<$) `ROOT` with a single slot for arbitrary properties (like in Javascript), `NAMED` < `PROPED` for named objects, `DISCR` < `NAMED` for discriminants, `CLASS` < `DISCR` for classes, `FIELD` < `NAMED` for fields, and many others (like `SEXP` for basilys source expression).

The basilys runtime is implemented in more than 6KLOC, and permits the development of a small lispy like compiled domain specific language.

3 A compiled domain-specific language for analyzers

As suggested before, a domain-specific language is useful to express more briefly sophisticated analysing algorithms, with an internal representation suited for meta-programming. Lisp [16] like languages fit the bill.[25] Hence, the compiled domain-specific Basilys language is a Lisp$_1$-like langage, somehow similar to Scheme (with the important restriction that tail-recursion is not supported, because it is non-trivial to compile it to portable C for our runtime.). The topmost internal representations are organized with s-expressions, instances of `SEXP` (containing an optional source location, an operator, a list of arguments).

Most of the data handled by Basilys are basilys values (i.e. pointers) described in §2.2 and §2.3, but handling of non-values (like unboxed raw integers, or raw trees) is also required.

3.1 low level syntax and informal semantics

Assuming some familiarity with Lisp or Scheme, we illustrate our language by giving a few examples. Conceptually, like every Lisp, it is an evaluation based language of expressions. In practice, it is compiled (by generating C code). The examples suppose that `two` and `three` are each bound to a boxed integer containing 2 and 3, stored in the current frame as `curfra.varp[ITWO]` and `curfra.varp[ITHREE]` where `ITWO` and `ITHREE` are in reality some generated indexes, and that `u` is bound to an unboxed machine long integer `1`.

[25] A multi-staged statically typed language like MetaOcaml would be even nicer, but too difficult to realize, because it would require defining a formal typing system for Gcc internal representations and implementing its type inference.

Constants, like `1`, `"a string"`, or (existing) named objects like `#NAMED` denote values[26] and evaluate to themselves. `#NIL` is the nil pointer value (also false).

Some keywords, starting with a colon, are useful in particular for type indications like `:int` (machine long unboxed integer) `:tree` (raw GCC `tree`-s), etc.

`(+ two 3)` evaluates to a boxed integer 5, but compiles to unboxing `two` and boxing the result, something like

```
long t22
 = basilys_get_int(
      curfra.varp[ITWO]);
long t23 = ((t22) + (3));
curfra.varp[IRES] =
 basilysgc_new_int(GLOB_DISCR_INT,
      t23);
```

We need basic Lisp control flow primitives, e.g. `(progn` e_1 ... e_n`)` for sequential (side-effecting) evaluations, `(if` c t e`)` for conditionals, `(forever` ϕ e_1 ... e_n`)` for never-ending loops labelled by name ϕ (repeatedly evaluating in sequence the e_i for side effects) which can be exited by `(exit` ϕ e'_1 ... e'_m`)` inside (evaluate the e'_j in sequence and jump out of the loop labelled ϕ in the same function with the value of e'_m), etc.

An assignment `(setq` v e`)` sets variable v (either local in the current frame, or closed, in the current closure) to value of e.

Binding let constructs are like in Scheme or Lisp: `(let ((`v_1 e_1`) (`v_2 e_2`)` $\cdots$ `(`v_n e_n`)) (`b_1 ... b_k`)` locally binds each v_i to value of e_i en sequentially evaluates the b_j. However, non-pointers variables are useful, so we admit typed bindings `(`τ_i v_i e_n`)` where τ_i is one of `:int` `:tree` etc. For example `(let ((:int x (+ u 2))) (* x 3))` don't do boxing or unboxing of integers. An `flet` construct (Lisp syntax, semantically similar to Scheme's `letrec`) permit local definition of (perhaps corecursive) local functions.

Formal arguments lambda lists are as usual : `(lambda (`v_1 ... v_n`)` e_1 ... e_k`)` defines an anonymous function with n formal arguments v_i whose body is the sequence of e_j. We also admit typed formals `(`τ_i v_i`)`

[26] Hence `1` refers to a boxed integer basilys value.

Secondary results of a multi-valued application[27] are bound with `(multicall` *variables applied-fun* `(` *arguments* ... `)` *bodies* ...`)`; results are returned with `return`. For example

```
(multicall (f1 f2)
  (lambda (f x)
    (return (f x) (f (f x))))
  (g k)
  (list f1 f2))
```

returns the list made by applying once and twice `g` to some `k`.

We need to be able to express that our `+` is a primitive taking two plain (long) integer arguments `x` and `y` and gives an integer, and give its expansion as a C code chunk: with

```
(define-primitive +
  ((:int x) (:int y))
  :int
  "((" x ")+(" y "))")
```

Progressively, all required GCC notations (e.g. defined in GCC `tree.h` file), and all (accessing, mutating, side-effecting) operations on basilys values should be likewise defined as primitives.

We also need macros, like in Lisp. A macro is expanded into some s-expr which is in turn evaluated (actually compiled).

3.2 Modules and their compilations

A module is a sequence of binding definitions and initializations. A binding exist at compile time and at run time. Binding definitions are mostly `defun` to define a function, as `(defun f (x) (+ x 23))` or `defvar` to define a variable, `defclass` to define a class, `definstance` to define an instance. Only publicly exported definitions and macros (thru a `export` or `export-macro` directive) are visible. Macros are just functions (called by the macro expansion process) exported as macros and useful only in other modules.

A module is compiled into a single C file which is then (compiled to a shared object and) dynamically loaded.

[27] Like in Common Lisp `multiple-value-bind` ... and in Scheme `call-with-values` ...

This C file contains a set of static functions for routines, and a single initialization function which build the runtime part of the module's exported bindings.

Naive compilation of such a Lispy language to C is pretty standard technology,[28] implemented by a sequence of usual transformations like:

1. macro expansion[29]
2. normalization, e.g. expanding `(f (g x) y 1)` into some internal equivalent of `(let ((`ϕ `(g x))) (f` ϕ `y 1))` where ϕ is fresh
3. constructing the set of closed variables
4. allocating slots in the call frame
5. expanding to simple C code chunks
6. etc.

Once a module has been translated to C, it can be compiled to a shared object which is then loaded with `lt_dlopenext` (and won't be unloaded).

It should be emphasized that dynamic code generation, (runtime) compilation to C, and a generational collector all work well together: Meta-programming requires some dynamic code generation, a generational collector is uneasy to use without automatic generation of C code using it, and dynamic code generation is portably possible thru C. Of course, all this has a cost overhead, but given that the analysis we are considering are costly, such an overhead is acceptable. And higher level languages also inspired GCC [8].

4 future work: implementing static analyzers

The above sections describe an infrastructure that we feel is useful to implement sophisticated static analyzers. Future work will include the following steps.

1. implementing a powerful GCC `tree` pattern matcher, essentially thru a big `match-gcc-tree` Basilys macro getting the GCC `tree` to match and a set of (suitably defined) patterns.
2. implementing a simple static analyser for mostly numerical functions, using the existing lattices of PPL.
3. implementing more complex lattices above for real data structures, using meta-programming when appropriate.
4. implementing some modular static analysis; while the analyzers above work on a single compilation unit or a set of source files compiled together with `-fwhole-program`, it is necessary to scale to bigger programs to store, in some persistent way (maybe augmented LTO?) partial results of analysis to reuse it.

[28] A prototype translator has been written in $>$ 3KLOC of CommonLisp, only used to bootstrap the Basilys compiler.

[29] Actually, even core languages feature like `if` or `let` are implemented as builtin macros expanding to some first internal representations.

5 acknowledgements

This work is done under the GlobalGCC (GGCC) project, labelled by ITEA Programme, numbered IP05012, partly funded by French Ministery of Economy, Finance and Industry (MINEFI), convention 06.2.93.0159.

Thanks to other GlobalGCC partners, in particular Albert Cohen and Sebastian Pop.

References

[1] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1–2):28–56, 2005.

[2] R. Bagnara, P.M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006. Available at `http://www.cs.unipr.it/Publications/`.

[3] Pierre Cointe. Metaclasses are first class: The ObjVlisp model. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 156–162, New York, NY, USA, 1987. ACM Press.

[4] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[5] P. Cousot and R. Cousot. *Basic Concepts of Abstract Interpretation*, pages 359–366. Kluwer Academic Publishers, 2004.

[6] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In M. Sagiv, editor, *Proceedings of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, Edinburgh, Scotland, April 2–10 2005. © Springer.

[7] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In M. Okada and I. Satoh, editors, *Eleventh Annual Asian Computing Science Conference (ASIAN'06)*, pages 1–24, Tokyo, Japan, LNCS, December 6–8 2006. Springer, Berlin. (to appear).

[8] Zdenek Dvorak. Declarative world inspiration. In *GCC Developers Summit*, pages 25–36, 2004.

[9] Dominique Guilbaud, Eric Goubault, Anne Pacalet, Basile Starynkévitch, and Franck Védrine. A simple abstract interpreter for threat detection and test case generation. In *WAPATV'01, with ICSE'01*, Toronto, 2001. `http://www.di.ens.fr/~goubault/papers/icse01.ps.gz`.

[10] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, Jam es Cheney, and Yanling Wang. Cyclone: a safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[11] Richard Jones and Rafael Lins. *Garbage Collection (algorithms for automatic dynamic memory management)*. Number ISBN 0-471-94148-4. Wiley, 1996.

[12] Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. *SIGPLAN Not.*, 41(7):54–63, 2006.

[13] George C. Necula, Jeremy Condit, Matthew HArren, and Scott McPeak a nd Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM. Trans. Programming Languages and Systems*, 27(3):477–526, May 2005.

[14] Jacques Pitrat. Implementation of a reflective system. *Future Gener. Comput. Syst.*, 12(2-3):235–242, 1996.

[15] Sebastian Pop. *The SSA Representation Framework: Semantics, Analyses and GCC Implementation.* PhD thesis, Ecole des Mines de Paris, december 2006. `http://www.cri.ensmp.fr/classement/doc/A-381.pdf`.

[16] Christian Queinnec. *Lisp in small pieces*. Cambridge University Press, New York, NY, USA, 1996. Translator-Kathleen Callaway.

[17] Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded C programs. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 231–242, New York, NY, USA, 2004. ACM Press.

[18] Nic Volanschi. A portable compiler-integrated approach to permanent checking. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 103–112, Washington, DC, USA, 2006. IEEE Computer Society.