USENIX Association

Proceedings of the FREENIX Track: 2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004

© 2004 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Creating a Portable Programming Language Using Open Source Software

Andreas Bauer
Institut für Informatik
Technische Universität München
D-85748 Garching b. München, Germany
baueran@in.tum.de

Abstract

On a first glance, the field of compiler construction and programming language design may not seem to have experienced major innovations over the last decade. By now, it is almost common knowledge how a lexer works, how parsing is done, but not many have yet realized how Open Source software — and in particular the GNU Compiler Collection — have silently offered language implementors new and better ways to do their job. Therefore, this paper describes the novel advantages Open Source software provides and, furthermore, it illustrates these with practical examples showing how the presented concepts can be put into practice. Another important contribution of this paper is to give an overview over the existing limitations and the technical problems that can occur when creating an Open Source based programming language implementation.

1 Introduction

The extensive Open Source GNU Compiler Collection (GCC) offers optimized code generation for a large number of different platforms and programming languages, for instance, C, C++, Java, and Ada, to name just a few. Historically, however, GCC was like most other compilers, aimed to support only one programming language, namely C, and for only a limited number of target platforms, i. e. those that would support the GNU system [1].

Due to the openness and the free availability of the source code, GCC was soon retargeted to, back then, even exotic hardware platforms and the differentiation between the "old" GNU-C back end, and the separate front ends became increasingly immanent. In other words, GCC turned into a quasi-platform itself, rather than being just another C compiler.

For programmers implementing a new language, this development can be of tremendous benefit, because it means that they can rely on GCC as being their actual target, so that native code generation is basically transparent to the front end. In a nutshell, it allows the implementors to focus on what is really important for them: language design.

Fig. 1 shows the different compilation stages of the compiler suite: the input can either be a straightforward C program code or, alternatively and more interestingly, interfacing occurs where the dashed line separates the stages. This leaves users with the choice of *a)* interfacing GCC in an "old fashioned" manner by emitting standard C code as well, or *b)* by interfacing the back end directly via the *tree structure*. This data structure is GCC's means to describe abstract syntax trees. Technically, however, a `tree` is merely a GCC-specific pointer type, but the object to which it points may be of a variety of different types; it is used to represent and perform various optimizations on the program (see § 4.3, 5).

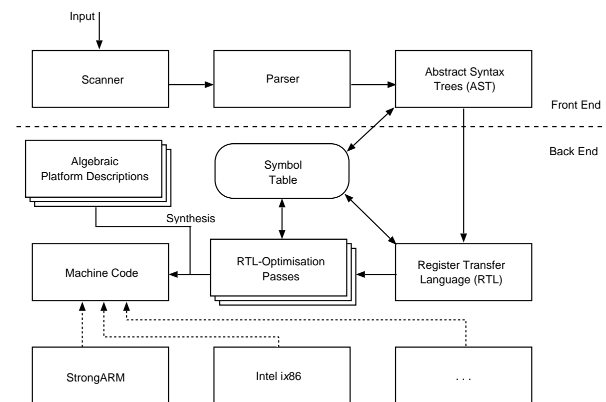

Figure 1: The main compilation stages of the GCC suite.

Although many compilers do indeed translate a program merely to C code, they potentially sacrifice the possibility of generating useful as well as detailed debugging information regarding the input program, and are also likely to miss out on specific intermediate program optimizations performed on the tree structure, e. g. alias analysis. Hence, the focus of this paper rests solely on integrating a well defined GCC front end that employs trees to communicate with the compiler suite's back end.

For the sake of completeness, it should be pointed out though that targeting (low, or high level) GNU-C code does offer advantages compared to, say, emitting

ANSI-compatible C. GNU-C extends ANSI-C with various non-portable constructs that help emitting optimized machine code. Nested functions, or global register variables are just some of the many characteristics custom to GNU-C [1].

An exceptional thing to note about Fig. 1 is how GCC handles the generation of native binary code: by consulting a separate, more or less, algebraic specification of the actual platform, such as `i386-gnu-linux-aout`, the abstract machine code written in the Register Transfer Language (RTL) gets mapped to native machine code according to the physical reality of the respective targets. This "physical reality" is typically determined by the available set of commands, number of processor registers, employed calling conventions, and so forth.

Such a strict differentiation between the various processes and the strong modularization of the GCC suite as it is also reflected in the figure, essentially, makes many of the typical tasks a compiler writer has to go through [2] redundant. For instance, complicated basic block analysis, or register coloring algorithms in the back end do not need to be re-implemented any longer. Instead, by using the compiler suite, all the effort can be put into designing the distinctive and essential features of a new programming language. Back end optimizations are already in place.

Problems, of course, remain. Although the GCC front ends do not need to be concerned about hardware constraints in the first place, mapping from higher level language features into GCC's interfacing `tree` representation can be all, but trivial. As a matter of fact, certain types of programming language front ends have been battling with GCC's code generation strategies for quite some time. (See, for instance, [3].) However, this is not necessarily so, because the program representation in terms of a tree structure is inadequate, but rather due to the fact that GCC treats that intermediate program representation, basically, as if it was a procedural (C-like) program. In a lot of cases this is not a problem, in others, however, very subtle problems arise.

Consequently, this paper not only goes through the notion of targeting GCC as a portable back end (see § 4, 5), but it also hints to practical problems that may arise when applying the presented concepts in a straightforward naïve manner (see § 6).

2 Related Work

Especially in light of commercial software development, GCC is by far not the only portable back end solution, even though it is probably one of the most accessible and useable ones today, due to the free availability of the sources and the active community surrounding it.

2.1 Back Ends

For instance, Chess/Checkers [4] is a very successful, commercial framework that is suited particularly well to build embedded systems software. The Chess module acts as a — more or less — standard C compiler while the subsequent passes of Checkers map the output to a user specified architecture written in a specialized hardware description language.

But also in the Open Source world, further portable back ends do exist, e. g. MLRISC [5] which is a customizable optimizing back end written in Standard ML that has also been successfully retargeted to multiple (mostly RISC) architectures. It deals with the special requirements imposed by the execution model of different high level, typed languages by allowing many components of the system to be customized to fit the source language semantics as well as the runtime system requirements.

A framework which comes close to GCC's ideals is the Little C Compiler (lcc) [6]. Basically, it is a retargetable compiler for standard C and generates native code for the Alpha, Sparc, MIPS R3000, and Intel ix86 as well as for its successors. Directly interfacing with lcc is different to GCC though and, therefore, not discussed in this paper.

Additionally, new programming languages could also use (parts of) the free Zephyr environment [7] which also offers concepts that are similar to those found in the GCC suite: for instance, the Zephyr component VPO is a platform and language independent optimizer that is built upon its own register transfer language. It has already been used with several C front ends, each of which generates a different intermediate language.

Obviously, the choices are manifold, but for the remainder of this paper the focus rests solely on the GCC back end, simply because it is the most widely used of all the presented suites, has a very large and active developer community, and is open and free in the sense of the GPL to allow and, in fact, encourage modifications to it.

2.2 Front Ends

Already a large number of free (and not so free) programming languages make use of a separate, portable back end. The increasingly popular logic language compiler Mercury [8], for instance, offers even multiple interfaces; among them is one for "pure" low level C, GCC trees, Java, and lately even .NET support was added.

The Glasgow Haskell Compiler (GHC) [9] is another prominent compiler for a declarative language, namely Haskell, that achieves portability thanks to the GCC back end. Although GHC offers its own optimizing code generation for Sparc and Intel ix86 processors, it still relies on the GCC suite when a wider range of hardware

targets needs to be addressed. GHC, on the other hand, is also a perfect example where straightforward interfacing fails. The reasons for that are outlined in greater detail in § 6.

Of course, further interesting language implementations based on GCC exist. Aside from C, the standard distribution already supports Java (GJC), Ada (GNAT), Fortran (G77), Treelang, Objective-C, and C++ (G++). However, when considering the integration of a new language these may not turn out to be the best starting points, since each such front end is rather sophisticated in itself, and the essential interface mechanisms to GCC cannot be seen clearly. This paper aims to narrow this documentation gap at least partly.

3 A Toy Expression Language

This section introduces the foundations of a rather simple expression language which will be used as an example throughout the remainder of this text. Let's call the language *toy*, simply because it does almost nothing useful. Toy is inspired by the "pocket calculator language" hoc as it is described in [10] and, similarly, in other text books.

$$
\begin{array}{lcl}
list & ::= & \mathbf{empty} \\
 & | & list \\
 & | & list\ fnbody \\
 & & \\
assignment & ::= & \mathbf{variable} = expr \\
 & & \\
fndecl & ::= & \mathbf{name}\ (\,) \\
fnbody & ::= & fndecl\ :\ \mathbf{begin}\ expr\ \mathbf{end} \\
 & & \\
expr & ::= & \mathbf{number} \\
 & | & \mathbf{variable} \\
 & | & assignment \\
 & | & expr + expr \\
 & | & expr - expr \\
 & | & expr * expr \\
 & | & expr \,/\, expr \\
 & | & (\ expr\ ) \\
 & | & -expr
\end{array}
$$

Figure 2: The abstract syntax for the sample language, toy.

An abstract syntax for our sample language can be seen in Fig. 2. The bold font is used to reference to tokens which are handled separately. Ambiguities in the grammar (usually resulting in shift-reduce conflicts) can later be resolved by assigning the respective yacc precedences for each operator. Note, for the sake of simplicity, loop structures and the like have been omitted from the grammar. However, the sections § 4.3–5 address the processes of including additional as well as more advanced language features.

Basically, toy covers the most elementary operations of arithmetics and allows the user to hold temporary values in variables. Hence, a toy implementation accepts only very basic programs and does not issue any warnings at compile time; that is, a toy program is either correct, or incorrect.

4 Interfacing directly with GCC

Despite a number of already existing front ends for GCC, interfacing its optimizing back end is a process that can be all, but easy to conceive. This is mainly due to the fact that *a)* the interface itself does not remain 100% stable and has evolved in an ad-hoc manner along with each additional front end, and *b)* the entire process as well as the interface itself are not properly documented anywhere [11] (especially if one considers file dependencies and the like as part of the actual interface).

4.1 Essential Files

Most importantly, when writing a new compiler based on GCC the user has to provide a number of essential files such as `Make-lang.in`, or `config-lang.in` which describe the build dependencies as well as the name of the new language, unique suffixes (`.toy`), and, finally, the name of the executable compiler (`cc1toy`). These files are all located in a separate directory inside GCC's main source directory, e. g. `gcc-3.3.2/gcc/toy`.

Thanks to the rather accurately documented Makefiles of already shipped front ends, it is almost obvious how a rudimentary `Make-lang.in` should look like: GCC merely expects a number of build, install, and clean targets in order to integrate the new front end into the overall compilation process. In other words, the shell command '`make toy.clean`' should be functional to clean up merely the toy object files, while '`make toy.dvi`' should produce a printable manual in dvi-format, and so on.

Note, that the executable `gcc` is usually built as the main *compiler driver*, i. e. it recognizes and "drives" the user supplied source code to the according compiler, based on input file suffixes and command line options.

4.2 Methods of Interfacing

When creating a programming language, the user typically has a couple of different choices on how GCC can be integrated to achieve a maximum of portability later on. The most obvious and probably most widely used approach is to create a custom front end which is subsequently tied to the compiler suite's driver. However,

depending on the complexity and size of the new language implementation it is also possible and feasible to link merely the GCC back end to a stand-alone front end whose back end driver would then be a shared library. One language that follows this approach is Mercury [8]; here, the difference for users is mainly noticeable when invoking the compiler as the compile driver is not the `gcc` executable, as usual, but rather the new front end itself.

Either method affects primarily the configuration nitty gritty of the new compiler, because in principle a user can choose whatever language seems suitable for implementation as long as there exists at least a certain degree of compatibility with the C-language's calling convention (or, at least, some sort of an interface to it), essential for linking the parts together in the end.

Additionally, there is also a choice on whether to use GCC's internal data structures for code representation directly, or not. In other words, the front end could build a valid GCC tree structure, e. g. in the most simple form by using semantic actions in the attributed grammar, or it could build up and fill its own intermediate data structures which are subsequently translated into GCC trees and, finally, RTL.

Of course, both approaches are valid, and having to create a custom intermediate code representation in the front end is very tedious work. The advantage of going through the effort, however, is a better possibility to trace bugs in the user submitted code, because the front end would then be able to check additional constraints that are normally hard to tackle in later stages of the compilation process inside the GCC back end.

Also, when making up a GCC-based programming language entirely from scratch users are likely to face difficulties in situations where they have to interface directly with the `tree` structure: once the back end is given a malformed `tree`, it is hard to undo or rebuild parts of it, let alone associating the error with the originally provided piece of code. Therefore, the creation of an intermediate program representation first can be — despite a reasonable amount of extra work – sometimes a good idea if the supported language is rather challenging in terms of representing its core features, e. g. special scoping, nested inner functions and classes, etc.

4.3 The `tree` Representation

The `tree` structure acts as the main interface between a custom front end and GCC's back end. A good overview over all the pre-defined trees is available, for instance, in [1] but details are really only documented in the GCC source files `gcc/tree.def` and `gcc/tree.h`. Consulting and understanding these files is absolutely essential for our task.

A `tree` is really only a pointer type representing a single node of an abstract syntax tree structure. It can be used for a *data type*, a *variable*, an *expression*, or a *statement*. Each node has a "tree code" associated with it which says what kind of object it represents. For instance, the code

- `INTEGER_TYPE` represents a type of integer,
- `ARRAY_TYPE` represents a type of pointer,
- `VAR_DECL` represents a declared variable,
- `INTEGER_CST` represents a constant integer value,
- and `PLUS_EXPR` represents a plus-expression (see Fig. 2).

The structure to which the pointer points to is implemented via a C union type. The individual fields are accessed via *predicates*, i. e. C-type macros, such as `INTEGRAL_TYPE_P` which in turn calls and evaluates the result of `TREE_CODE` to determine whether a given node is of integral type. Although in general, it is feasible to apply any type of tree node to a predicate, there are certain macros that demand for nodes of exactly a certain kind; `TREE_CODE`, however, is not one of them. Common fields are summarized in the structure `tree_common`.

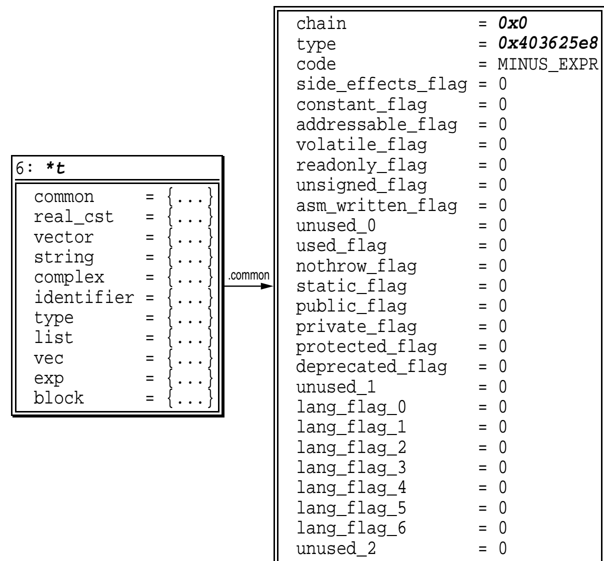

Figure 3: A binary minus expression node, visualized using the Data Display Debugger. *The* `tree_common` *structure is right of the* `tree` *union definition.*

The inside of an expression node is depicted in Fig. 3. Its operands can be accessed with the macro `TREE_OPERAND` as in `TREE_OPERAND(my_expr, 0)`; that is, an expression's operands are actually zero-indexed.

When writing a properly integrated GCC front end, i. e. one that issues trees, it is most useful to take

text.c:

```
int
main ()
{
  int result_0;
  compute (5);
  return result_0;
}

int
compute (int argument_0)
{
  return argument_0 * 2;
}
```

text.c.tu:

```
@1      type_decl         name: @2       type: @3      scpe: @4
                          srcp: <internal>:0            chan: @5
@2      identifier_node   strg: int      lngt: 3
@3      integer_type      name: @1       size: @6      algn: 32
                          prec: 32       min : @7      max : @8
@4      translation_unit_decl srcp: <internal>:0
        :
        :
@1969   identifier_node   strg: main     lngt: 4
@1970   function_type     unql: @1787    size: @20     algn: 64
                          retn: @3
@1971   function_decl     name: @1972    type: @1453   scpe: @4
                          srcp: test.c:9               args: @1973
                          extern
@1972   identifier_node   strg: compute  lngt: 7
@1973   parm_decl         name: @1974    type: @3      scpe: @1971
                          srcp: test.c:8               argt: @3
                          size: @6       algn: 32      used: 1
@1974   identifier_node   strg: argument_0             lngt:
```

Figure 4: On the right hand side is a typical dump for a single translation unit generated by the C front end of GCC. Boxed are the distinctive function definitions and arguments as they appear in the original source code on the left. The @-symbols identify and reference tree nodes as is also indicated by the arrows.

a closer look at the generated "tree code". The C and C++ front ends allow for this already using the `-fdump-translation-unit` switch, amongst others. A typical result of this can be seen in Fig. 4 and, although extensive and complex, the typical properties described above are all present in this short example. Furthermore, the additional "links" drawn in the figure give a rough idea about how well suited the tree structure is to extract control flow information from a translation unit.

5 Mapping

In order to map a toy statement covered by a rule $expr ::= expr + expr$ to a GCC binary expression node, the pre-defined expression `PLUS_EXPR` can be used, or accordingly, `MINUS_EXPR`, and `MULT_EXPR` for the other basic arithmetic operations. (Division is being treated separately due to data type and rounding issues.) Of course, additional expressions exist to hold further data types, single programming language statements, or even entire function definitions as well as general compound statements.

5.1 Generating Trees

For many languages, most of the tree generation occurs during parsing. Consequently, such front ends make use of a yacc-like grammar (see Fig. 2) to be able to employ other Open Source tools like GNU Bison, for instance. Bison uses an LALR(1) algorithm (look-ahead LR) to recognize a context-free language. Some front ends, like the C++ of GCC >= 3.4, however, use and have introduced their own *recursive descent* parsers.

There is no single "best way" of parsing, but it is noteworthy that there exist many programming languages for which a context-free grammar in general is not expressive enough. Again, the purely functional language Haskell provides an example for being one of them [12, § 9.3]. For a language like toy, however, the tools yacc, or GNU Bison are fully adequate. A good practice would be to hook them into `Make-lang.in` and to dynamically (re-)build the parser along with the actual front end. This also applies to lexicographic analysis via flex; flex, the "fast lexer", is usually a very good companion for this exercise, but not scope of this paper. (See [10], or `flex(1)` for further details on flex and yacc.)

To assemble a valid tree structure for a toy expression language, yacc *actions* can be used. (Other parsers have to invent their own.) That is, the implementation of the grammar shown in Fig. 2 gets extended by statements like

```
expr:  number        { ... }
     | variable      { ... }
     | expr '+' expr
       { $$ = build
              (PLUS_EXPR, /* Tree code */
               integer_type_node,
               $1,        /* Operand 1 */
               $3);       /* Operand 2 */ }
     | ...
     ;
```

where `build` is a pre-defined function of `src/tree.c`. It gets used to build a binary expression of a certain tree code with a certain type.

It is important to notice that the largest possible tree structure is always built on a per-unit basis which usually resembles an entire input function. Hence, the parser

typically interrupts at *a)* each function declaration, *b)* each definition, and *c)* sometimes also at each explicit function closing. Since toy functions are defined to be argument-less, the according actions could be implemented as follows:

```
fndecl:  name '(' ')'
         { $$ = build_fndecl ( $1 ); }
fnbody:  fndecl ':' BEGIN expr END
         { build_fnbody ( $1, $4 ); }
```

Basically, `build_fndecl` needs to call the according functions of `src/tree.c`, firstly to create a node representing the parsed function's return as well as argument types (`build_function_type`), and secondly to hold the declaration itself (`build_decl`). `build_fnbody` needs to contain code for building tree nodes that represent the return value as well as for emitting actual RTL. Both routines are schematically depicted in Fig. 5.

Historically, the transformations had to be performed statement by statement in order to facilitate RTL synthesis and to avoid space problems, but this narrow scope turned out too constraining for complex programs. Nowadays, GCC enjoys internal *garbage collection* (GC) to tackle the memory issues, which is almost always recommended to be used in front ends, too (see § 6.3).

5.2 From Trees to RTL

Basically, the `tree` data structure acts as the main interface between a GCC front end and its optimizing back end. However, a front end is not totally isolated from the technicalities of emitting RTL. Typically, it has to

- ensure that there exists a transformation from all the employed tree types to RTL;
- trigger RTL expansion at the end of parsing functions, or general compound statements;
- provide direct tree-to-RTL conversion, should custom tree types be involved that cannot be lowered to existing ones (see also § 5.3).

Although, `build_fndecl` does not directly emit RTL, the call to `src/tree-optimize.c:tree_rest_of_decl_compilation` does, in fact, trigger subsequent low level code generation. The function itself is a flexible wrapper around `src/toplev.c:rest_of_decl_compilation` and can also handle features like nested input functions should it be required.

In `build_fnbody`, the user has to deal with RTL expansion more directly. As a rule of thumb, `expand_*` functions usually accept a `*_STMT`, or `*_DECL` node and emit RTL for it thus, `src/function.c:expand_function_start` starts RTL for a new function and must always be called first.

Omitted are the details of handling arguments and garbage collection, since toy functions are rather basic with only one type, integer, and zero parameters each. Additionally, these aspects are very front end specific and can be added afterwards, once a basic front end compiles.

In principle, the internal garbage collector is accessed via different `ggc_*` calls:

ggc_add_tree_root: This is used to hook into the marking algorithm of GCC and should come first.

ggc_alloc: In order allocate memory, `ggc_alloc` has to be used.

ggc_collect: This function triggers the top-level mark-and-sweep routine. It gets called several times by `src/toplev.c:rest_of_*` functions to free memory.

The file `src/toplev.c` also provides the main entry point for the C and C++ front ends, i. e. a `main` routine, which in turn invokes the various compilation passes. Obviously, the file also handles a lot of the code generation from trees and, therefore, can and should be used directly wherever possible. Reimplementing `src/toplev.c` stuff requires a very thorough understanding of the GCC internals and is, definitely, one of the most difficult parts when building a compiler from scratch.

5.3 Introduction of new Tree Types

The pre-defined data types, tree codes and macros as they are available in GCC versions $3.x$ are — for a basic expression language as it is described in § 3 — sufficiently expressive to be able to create a comprehensive intermediate code representation which then gets mapped to RTL instructions.

The downside, however, is that currently GCC front ends behave somewhat like a C compiler, in a sense that the syntax of the tree structure is strongly biased towards procedural languages. In other words, the front end of a language more sophisticated than toy, or C, probably based on an alien programming paradigm, almost always needs to introduce its own tree definitions to adequately represent statements, functions, and all kinds of additional types (see § 6, 7).

Introducing new kinds of trees happens in the respective `.def` files of each front end, e. g. simply via `DEFTREECODE (PLUS_EXPR, "plus_expr", '2', 2)`: the first operand is the tree *code*, the second is its *type*, the third is its "kind" (i. e. used for constants, declarations, references, binary arithmetic expressions, and so on), and the optional fourth argument is the number of argument slots to allocate if necessary; two in this example.

```
tree
build_fndecl (char* name)
{
  ...
  my_fntype = build_function_type
               (integer_type_node,
                param_type_list);
  ...
  my_fndecl = build_decl
               (FUNCTION_DECL,
                name,
                my_fntype);
  ...
  tree_rest_of_decl_compilation (my_fndecl, 0);
  return my_fndecl;
}
```

```
void
build_fnbody (tree fndecl, tree expr)
{
  expand_function_start (fndecl, 0);
  ...
  expand_return (build
                 (MODIFY_EXPR,
                  void_type_node,
                  DECL_RESULT (fndecl),
                  expr));
  ...
  expand_function_end (...);
}
```

Figure 5: These two functions are responsible for building trees representing an input function's declaration, as well as for holding its definition, type, and to trigger RTL generation for each node, respectively. Naturally, `build_fndecl` *should be called first. Calls to* `expand_*` *denote direct RTL expansion.*

On the one hand side, the extendibility allows for a representation of almost arbitrary programming language features and types, but on the other it also requires additional work to *a)* either lower the extensions to the existing nodes, or *b)* to make up additional `expand_*` functions that match, say, an `*_STMT` node to RTL. The expansion functions mostly reside in `src/stmt.c`, making up custom ones, however, is more challenging than lowering and one of the reasons why RTL is not the interface of choice between the front and the back end of GCC.

6 Problems

Although, GCC offers marvelous possibilities to speed up the development of rather platform independent programming language implementations, it does put its own peculiar constraints on front ends, especially if those resemble non-imperative paradigms.

Most prominently, functional and logic programming languages have a hard time taking full advantage of a "generic" back end like GCC. Many of them offer quite advanced concepts such as first order and higher order functions, automatic garbage collection, and a strong static type system based on polymorphism — clearly a contrast to the C language.

6.1 Higher Order Functions

Many programming languages treat functions as first-class citizens to support higher order functions. A higher order function takes one or many functions as argument, or returns these as its "return value". This is especially useful when using the *continuation passing* style (CPS) with the continuation being a passed-on function that should be executed next, similar to an additional program counter. In other words, the order of the computation is implicitly defined by an additional function argument — the continuation. Note, real CPS functions never return.

CPS gets employed in many functional programming language compilers and interpreters, such as GHC used for Haskell, for instance. But, unlike in Haskell, functions are not first order in C nor in GCC, i. e. functions can not be passed as an argument. Therefore, higher order functions do not exist in C. Function pointers, however, are first-class and the continuation passing style can be approximated by using `void` pointers to functions.

```
-- a) Without a higher order function
squareListNoHof []   = []
squareListNoHof list = ((head list)^2):
                        (squareListNoHof
                         (tail list))

-- b) With a higher order function
squareList list = map (^2) list
```

Figure 6: These two Haskell algorithms compute the square of each value given in a `list`*, in* a) *without the use of higher order functions, and in* b) *in combination with the higher order function* `map` *which takes a function and a list as arguments. Clearly, solution* b) *is more compact, thus easier to comprehend.*

In essence, this means that a lot of explicit casts and indirect function calls are involved. Since function pointers can only refer to functions with either global scope, or local scope to a particular file, this approach itself is not suitable to get the full flexibility of higher order functions which users may be used to from their declarative programming language of choice (see Fig. 6). Higher order functions may also be lexically nested and need *closures* to represent such scope information (see § 6.3).

Consequently, declarative languages like Haskell, ML, or Lisp demand for a compiler which provides its

own mechanisms for handling higher order functions such that the GCC back end must not be overly concerned with that "technicality" anymore. For example, GHC tackles this problem using a twofold approach: firstly, it maintains its own internal stack structure parallel to the architecture's runtime stack, and secondly, by modifying the output of GCC using a crude pattern matching algorithm that removes certain assembly instructions that are responsible for handling function arguments and frames [13]. This, however, is also described in greater detail in § 6.2.

6.2 Tail Calls

In all declarative programming languages a high number of recursive function calls occur of which, typically, many are tail calls. A tail call is a function call in the tail position of the calling function.

Consider the following straightforward Haskell implementation of the greatest common divisor algorithm:

```
gcd :: Int → Int → Int
gcd a b | a == b = a
        | a > b  = gcd (a − b) b
        | a < b  = gcd a (b − a)
```

Typical for the declarative programming style, this code is relatively easy to conceive; it contains two tail recursive calls to calculate a result. In that vein, most functional code bears a high percentage of tail calls which are either recursive, or even mutually recursive with a differing number of function arguments, respectively.

Tail calls can be implemented without requiring more than a single stack frame of the architecture's runtime stack, regardless of the amount of tail calls performed. This is possible, because a tail call is, essentially, the very last instruction of the caller. Thus, the caller has finished computation at this point and its stack frame should be free for "recycling" by the callee [3, 14] which would then take over responsibility to return to the original caller, or issues further tail calls.

According to the UNIX calling convention for C, however, the caller is responsible for cleaning up the callee's function arguments (see [15]). Thus, marshalling arguments subsequently to the actual tail call has to be realized in such a way that the topmost caller either does not remove a wrong number of arguments, or lets the callee discard unused stack slots itself. Fig. 7 shows what could happen, for instance, when a function $f(A1, A2, A3)$ performs a tail call to $f'(A1, A2, A3, A4)$ which expects an additional `int` argument.

Although elegant, tail calls impose a number of problems to the user who seeks to employ GCC as a back end for, say, a purely functional programming language compiler such as as GHC:

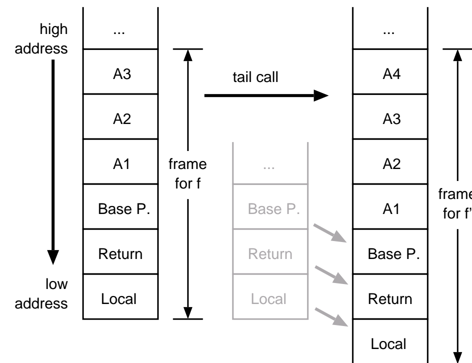

Figure 7: A proper tail call to a function which takes more arguments than the caller requires runtime stack marshalling for reusing the current stack frame, e. g. shifting the return address down (or up, depending on the platform).

1. Using recursion exclusively as a means of computation has not been foreseen by most platform's C calling convention [15]. Typically, the calling conventions do not differentiate between a tail call and a "normal" call. Hence, a stack frame gets always reserved via the `call` command of the according architecture and leads to a stack growth rate of $O(n)$ — linear but malicious.
2. Because of 1. proper tail calls with an overall stack consumption of $O(1)$ are hard to implement in retrospective without sacrificing binary compatibility to existing libraries and systems software.
3. Mapping continuation passing to a sequence of tail calls via C pointers will not work without addressing and solving the problems imposed by 1. and 2., respectively. Albeit, optimizing *indirect* tail calls turns out to be even more challenging than direct tail calls (see [3]).

Despite a possible first impression that the tail call problem might be merely a minor optimization candidate for a compiler zealot with too much time, the problem is of a very fundamental nature today, because generating code which does not comply to the C standard calling convention results in hard-to-resolve issues of binary compatibility to already existing executables as well as in portability problems. However, efforts to tackle the tail call problem, especially with respect to making GCC a better back end for declarative languages have already produced measurable improvements in that area (see § 7, [3]).

As already mentioned in § 6.1, GHC avoids the problem by letting the GCC back end emit unoptimized machine code which is then processed by a Perl script called "The Evil Mangler" [13]. This script alters the functions' *epilogues* and *prologues* such that they do not reserve stack frames for tail calls anymore. This is possible due to GHC's internal stack management

which, essentially, substitutes the platform's runtime stack in terms of function parameter passing; calls are then argument-less.

6.3 Garbage Collection

Rather than using `malloc` and `free` to obtain and reclaim memory, many modern programming languages, such as Java, or C#, offer the concept of garbage collection (GC) to automatically reclaim unused memory segments in the heap. In a nutshell, this is supposed to help avoiding dangerous and hard-to-find bugs like buffer overflows which are a number one source for remote exploits in today's systems software [16].

But also functional languages rely on GC to allow *lazy evaluation* [17] and higher order functions in support for notions such as continuation passing (see § 6.1, 6.2). Again, the problem with GC is that this concept is alien to C, thus mostly to GCC as well; although, this is not entirely true anymore: the GNU compiler for Java (GCJ) produces a library called `libgcj` which implements the Boehm-Weiser conservative garbage collector [18] as it is also employed in the free .NET implementation Mono [19], for instance. Boehm-Weiser realizes a basic mark-sweep algorithm to perform collections.

Closures to express the scope of functions, or objects as they are also needed in functional programming are the crux here. A closure, typically, is a function generated at runtime to capture information about the environment. Hence, it seems self-evident to put these on the architecture's stack. However, as closures usually exceed the lifespan of the items that are being referenced by it at a time (Think of tail call optimization as an example!) this often turns out impossible.

A valid alternative is often to create closures in the heap with the expense of having to garbage collect the space occupied by them. Indeed, this is what many declarative language compilers do nowadays. Java and C# on the other hand rely on garbage collection merely for a convenience reason; both do not support closures per se.

At the moment the Java front end is the only programming language implementation based on GCC that actually uses Boehm's integrated code directly, while all the functional and logic languages either maintain their own memory structures or, more recently, use alternative concepts like that of a "shadow stack" which is supposed to make collecting "garbage" from the platform's C runtime stack easier [20] and by doing all the required work entirely in the front end.

Additionally, even more subtle technical problems may arise, for instance, when pointers are hidden that really need to be visible to the collector, as it can happen when `memcpy` is used to copy these to unaligned memory locations, or by casting pointers to and from integers. While programmers would abstain from such a dangerous practice, it might turn out more difficult avoiding it totally in automatically generated code. A similar issue is related to cyclic data structures that reference to each other directly, or indirectly, like a circularly linked list: in that case the GC algorithm is unable to remove the referenced memory and the application can easily run out of heap space, despite a working GC.

The variety of different approaches to the problem of GC hints to the fact that there is, currently, not a standard solution to realize a 100% reliable automatic memory management as it necessary for a number of modern programming languages, even though the conservative garbage collector seems to be sufficient for some standard applications.

Essentially, all the open issues mentioned in this section leave language implementors two choices should they require GC: 1. they can try to cope with the restrictions the current conservative collector imposes, or 2. they can write their own memory management, in which case using the GCC back end does not provide additional convenience, unfortunately.

7 Conclusion

Unarguably, implementing a new programming language is but a trivial undertaking. However, this paper has shown that Open Source software and in particular the GCC suite are flexible and nowadays also mature enough to take on a whole variety of different (and sometimes tedious) tasks programmers used to tackle manually, e. g. by re-implementing well known algorithms.

Although, the presented tools can be a tremendous help when targeting a wider range of platforms, problems using this approach remain — as is sketched in § 6 — especially for declarative, or strictly object oriented programming languages.

This may be the reason why recent developments in the GCC community are largely sparked by the internals of the Java front end. The Java group not only made clear there is need for sophisticated garbage collection, but it also introduced the foundations for a new intermediate program representation that, in a sense, surpasses the expressiveness of former GCC trees.

The new representation is called GENERIC and, in the future, each front end will be required to lower any kind of program representation to the GENERIC form [21]. Subsequently, the back end will translate GENERIC into a well defined subset called GIMPLE. The "gimplifier" is necessary, because most of the lower level optimization passes are going to be defined on the *static single assignment* (SSA) form [22, 23] which can be distilled from a GIMPLE representation at ease.

SSA has been chosen, mainly because many of the newer compiler optimizations are defined over this form [24, 25, 26]. In fact, some of the new as well as old optimizations are hard to realize on trees, or RTL alone: trees tend to be rather individual for each language front end and are highly context dependent (see § 4.3), whilst RTL is often too close to being an abstract hardware platform, e. g. by associating objects to virtual stack slots rather early, although in later compilation passes these objects may have turned as redundant temporaries.

It is obvious that the SSA based rewrite of GCC is a very ambitious project that primarily affects large parts of its very complex back end. Therefore, results are not likely to ship anytime soon, at least not before GCC 3.5 is released. The current CVS version, however, works reasonably well already and effort is currently being put into porting the existing front ends over to GENERIC as is the case with Fortran 95.

However even at present, GCC is well suited as a portable back end for a variety of different programming languages. The tree representation is flexible as well as extensible (see § 4.3, 5.3) and various optimizations are performed on the existing intermediate program representations.

Unfortunately, the GCC interfaces have always changed in subtle ways and documentation on building and integrating front ends has always been sparse. By showing the most fundamental coherences between a language's grammar, the employed tools (see § 3, 4), the tree representation and the transition over to RTL (see § 5), this paper sketches all steps necessary to build a GCC based compiler, more or less, from scratch.

On the other hand, the paper has also shown where the current problem spots are, in particular, for strictly object oriented languages, or declarative ones (see § 6): although improving, the memory management of the back end and the generated code is often insufficient to deal with features like closures, garbage collection, or higher order functions.

Along with the extremely helpful GCC mailing list (archives), the existing front ends and their respective documentation, it should now be possible for anyone interested to understand what is required to integrate a new front end into the GCC suite.

Alternatively, of course, a user may chose to implement an optimizing and specialized back end himself, however, one must not forget that, despite all trouble spots, GCC contains hundreds (if not more) man-years worth of code optimizations, tweaks, and ported platforms and it seems only natural to make use of these achievements whenever possible, or even better, to help overcome problems and lead projects like the SSA rewrite to a success.

References

[1] GNU Compiler Collection Internals. http://gcc.gnu.org/onlinedocs/gccint/.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison Wesley Higher Education, 1986.

[3] A. Bauer. Compilation of Functional Programming Languages using GCC — Tail Calls. Master's thesis, Institut für Informatik, Technische Universität München, Germany, 2003.

[4] Chess/Checkers. http://www.retarget.com/.

[5] MLRISC. http://cs1.cs.nyu.edu/leunga/www/MLRISC/Doc/html/.

[6] Little C Compiler. http://www.cs.princeton.edu/software/lcc/.

[7] Zephyr/VPO. http://www.cs.virginia.edu/zephyr/.

[8] T. Conway, F. Henderson, and Z. Somogyi. Code generation for Mercury. In *Proceedings of the 1995 International Symposium on Logic Programming*, pages 242–256, Portland, Oregon, 1995.

[9] The Glasgow Haskell Compiler. http://www.haskell.org/ghc/.

[10] B. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice Hall, New Jersey, 1984.

[11] Z. Weinberg. A Maintenance Programmer's View of GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 257–268, May 2003.

[12] The Haskell 98 Report. http://www.haskell.org/onlinereport/.

[13] The Glasgow Haskell Compiler Commentary — The Evil Mangler. http://www.cse.unsw.edu.au/˜chak/haskell/ghc/comm/.

[14] W. D. Clinger. Proper tail recursion and space efficiency. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, 1998.

[15] *System V Application Binary Interface/Intel386 Architecture Processor Supplement*. The Santa Cruz Operation, Inc. (SCO), fourth edition, 1996.

[16] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, pages 177–190, 2001.

[17] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.

[18] H. Boehm. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans˙Boehm/gc/.

[19] The Mono Runtime. http://www.go-mono.com/runtime.html.

[20] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the third international symposium on Memory management*, pages 150–156. ACM Press, 2002.

[21] D. Novillo. Tree SSA — A New Optimization Infrastrcutre for GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–195, May 2003.

[22] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM Press, 1988.

[23] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11. ACM Press, 1988.

[24] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 97–105. ACM Press, 1998.

[25] E. Stoltz, H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment form for explicitly parallel programs: Theory and practice, 1994.

[26] A. Leung and L. George. Static single assignment form for machine code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 204–214, 1999.