

GDB Integration for Pascal-P6

Implementation Assessment

Prepared for the Pascal-P6 Project

February 2026

Scope

1. DWARF Type Descriptions for Variable Inspection
2. Call Frame Information for Stack Backtraces

1. Current State of Debug Information in pgen

1.1 What Already Works

The pgen native code generator (AMD64/GCC) already emits GAS (GNU Assembler) directives that provide source-level line mapping:

```
.file    "hello.pas"
.file    1 "/home/.../sample_programs/hello.pas"
.loc 1 4 1      # source line 4
.loc 1 5 1      # source line 5
```

These .file and .loc directives allow GDB to map instruction addresses to Pascaline source lines. This means GDB can already: display source code at breakpoints, set breakpoints by line number, and step through source lines.

Additionally, the compiler emits function labels with ELF type annotations:

```
.globl  sieve
.type   sieve, @function
```

This allows GDB to identify function entry points and set breakpoints by function name.

1.2 What Is Missing

Two categories of DWARF debug information are absent from pgen output:

- **Data type descriptions:** No DWARF type DIEs (Debug Information Entries) are emitted. GDB cannot inspect variables, parameters, or locals because it has no knowledge of their types, sizes, or locations. Variables like sieve.flags, sieve.count appear as assembler equates (e.g., "sieve.count = globals_start+8288") but this information is not exposed to GDB.
- **Call Frame Information (CFI):** No .cfi_* directives are emitted. GDB cannot perform stack backtraces because it does not know how to unwind the custom frame layout used by pgen. The pgen frame structure uses enterq with additional frame metadata words (current ep, stack bottom, previous ep) that differ from the standard C calling convention.

1.3 Current Frame Layout

Understanding pgen's frame layout is essential for generating correct CFI. The current procedure prologue generates:

```
# Caller executes: call proc.3      (pushes return address)
# In callee prologue:
pushq  $0          # current ep      [CFA-16]
pushq  $0          # stack bottom    [CFA-24]
pushq  $0          # previous ep     [CFA-32]
enterq $0,$1       # push rbp, rbp=rsp [CFA-40]
# ... allocate locals by pushing zeros ...
movq   %rsp,16(%rbp) # set stack bottom
andq   $~0xf,%rsp   # align stack
pushq  %rbx         # save callee-saved regs
pushq  %r12
pushq  %r13
pushq  %r14
pushq  %r15
pushq  %r15         # alignment push
```

The corresponding epilogue:

```
popq  %r15        # undo alignment push
popq  %r15        # restore callee-saved regs
popq  %r14
popq  %r13
```

```
popq    %r12
popq    %rbx
leave          # restore rbp, rsp
addq    $24,%rsp      # remove 3 frame metadata words
popq    %rcx          # get return address
addq    $N,%rsp       # remove caller parameters
pushq   %rcx          # replace return address
ret           # return
```

Stack frame layout (addresses growing downward):

[return address]	CFA-8	(pushed by call)
[current ep]	CFA-16	(frame metadata)
[stack bottom]	CFA-24	(frame metadata)
[previous ep]	CFA-32	(frame metadata)
[saved rbp]	CFA-40	(rbp points here)
[locals...]	rbp-8 and below	
[alignment padding]		
[saved rbx]		variable offset from rbp
[saved r12..r15, r15]		variable offset from rbp

CFA = rbp + 40

Return address = rbp + 32 = CFA - 8

2. DWARF Type Descriptions

2.1 Overview

DWARF debug information encodes data types as a tree of Debug Information Entries (DIEs) in the .debug_info section. Each DIE has a tag (e.g., DW_TAG_base_type, DW_TAG_array_type) and a set of attributes (name, size, encoding, etc.). Variables reference their type DIEs, and include location descriptions that tell GDB where to find the data at runtime.

The approach is to have pgen emit DWARF directives in the assembly (.s) output. GAS and the linker handle the binary encoding. This is the standard approach used by GCC, Clang, and other compilers targeting GAS.

2.2 Pascaline Types to DWARF Mapping

Pascaline Type	DWARF Tag	Notes
integer	DW_TAG_base_type	DW_ATE_signed, 8 bytes
real	DW_TAG_base_type	DW_ATE_float, 8 bytes (double)
char	DW_TAG_base_type	DW_ATE_unsigned_char, 1 byte
boolean	DW_TAG_base_type	DW_ATE_boolean, 1 byte
subrange	DW_TAG_subrange_type	DW_AT_lower/upper_bound
enumerated	DW_TAG_enumeration_type	DW_TAG_enumerator children
array	DW_TAG_array_type	DW_TAG_subrange_type for each dim
record	DW_TAG_structure_type	DW_TAG_member for each field
variant record	DW_TAG_structure_type	DW_TAG_variant_part + discriminant
pointer (^T)	DW_TAG_pointer_type	DW_AT_type references base type
set	DW_TAG_set_type	Limited GDB support; may need opaque
file	DW_TAG_base_type	Opaque handle; show as integer
string	DW_TAG_array_type	Array of char with special handling
class	DW_TAG_structure_type	With DW_TAG_subprogram methods

2.3 Variable Location Descriptions

Each variable needs a DWARF location description telling GDB where to find it at runtime. pgen uses three storage classes:

- **Global variables:** Stored at fixed offsets from globals_start in the .bss section. Currently emitted as assembler equates (e.g., "sieve.count = globals_start+8288"). Location expression: DW_OP_addr with the absolute address. These are straightforward.
- **Local variables:** Stored in the stack frame at negative offsets from %rbp. The compiler already tracks these offsets internally. Location expression: DW_OP_fbreg with the offset from the frame base. The frame base would be defined as %rbp.
- **Parameters:** Passed on the stack above the return address (caller-pushed) or in registers per the AMD64 ABI for some system calls. For Pascaline procedures, parameters are at positive offsets from the frame. Location: DW_OP_fbreg with positive offset, or register locations.

2.4 Implementation in pgen

The work involves modifying pgen to emit DWARF sections in the assembly output:

- **Abbreviation table (.debug_abbrev):** Define the set of DIE templates used (compilation unit, base types, arrays, records, variables, subprograms, etc.). This is a fixed table that can be emitted once.
- **Debug info (.debug_info):** Emit DIEs for each type, variable, and subprogram. The compiler already has the symbol table with all type information, variable offsets, and procedure/function signatures. The work is traversing this symbol table and emitting the corresponding DWARF entries.
- **String table (.debug_str):** Emit identifier names referenced by DIEs. This avoids duplicating strings in .debug_info.

The DWARF emission code would be a new module in pgen that is called after code generation for each compilation unit. It walks the symbol table and emits the appropriate GAS directives.

2.5 Complexity by Type

Not all types are equally difficult to represent in DWARF:

- **Easy (1-2 days each):** Base types (integer, real, char, boolean), pointer types, subrange types, simple arrays. These have direct DWARF equivalents.
- **Moderate (3-5 days each):** Records (including nested records), enumerated types, multi-dimensional arrays, strings. These require careful offset calculation and child DIE emission.
- **Hard (1-2 weeks each):** Variant records (DWARF variant parts with discriminants), classes (structure types with method children and inheritance via DW_AT_specification), container arrays (Pascaline's dynamically-sized arrays need special location expressions), set types (limited GDB support).

2.6 Effort Estimate

DWARF emission framework + abbreviation table: 1-2 weeks. Base types and simple variables: 1-2 weeks. Structured types (records, arrays, enums): 2-3 weeks. Advanced types (variants, classes, containers): 2-4 weeks. Testing and validation: 1-2 weeks.

Total: 7-13 weeks

3. Call Frame Information (CFI) for Backtraces

3.1 Overview

GDB uses DWARF Call Frame Information (CFI) to unwind the stack and produce backtraces. CFI directives describe, at each point in the code, how to find the return address and how to restore callee-saved registers. GAS provides `.cfi_*` directives for this purpose.

Without CFI, GDB cannot determine the call chain. It may attempt heuristic unwinding using `%rbp` as a frame pointer, but pgen's non-standard frame layout (with 3 extra metadata words between the return address and saved `%rbp`) will cause this to fail.

3.2 Required CFI Directives

For each procedure/function, pgen needs to emit CFI directives that track the changing stack layout through the prologue and epilogue. Example for the current frame layout:

```
proc.3:
.cfi_startproc
# After call: CFA = rsp+8, return addr at CFA-8
.cfi_def_cfa rsp, 8
.cfi_offset rip, -8
pushq $0          # current ep
.cfi_adjust_cfa_offset 8
pushq $0          # stack bottom
.cfi_adjust_cfa_offset 8
pushq $0          # previous ep
.cfi_adjust_cfa_offset 8
enterq $0,$1      # push rbp, rbp=rsp
.cfi_adjust_cfa_offset 8
.cfi_offset rbp, -40
.cfi_def_cfa rbp, 40    # switch to rbp-based CFA
# ... allocate locals ...
pushq %rbx
.cfi_offset rbx, <offset>
pushq %r12
.cfi_offset r12, <offset>
# ... etc for r13, r14, r15 ...
# ... procedure body ...
# epilogue:
.cfi_restore rbx
.cfi_restore r12
# ... etc ...
leave
.cfi_def_cfa rsp, 32    # CFA now rsp-relative
addq $24,%rsp
.cfi_def_cfa rsp, 8
ret
.cfi_endproc
```

3.3 Challenges

- **Non-standard frame layout:** The 3 metadata words (current ep, stack bottom, previous ep) between the return address and saved `%rbp` mean $CFA = rbp + 40$ instead of the standard $rbp + 16$. This is the core difference from a C-style frame and must be correctly encoded.
- **Dynamic local allocation:** Locals are allocated by pushing zeros in a loop until `rsp` reaches the target. The exact number of pushes varies. CFI needs to account for this, typically by using `rbp`-relative CFA after the `enterq` instruction, making the CFA independent of `rsp` changes.

- **Stack alignment:** The "andq \$~0xf,%rsp" alignment instruction makes rsp unpredictable. This is another reason to use rbp-relative CFA throughout the function body.
- **Callee-saved register locations:** The saved registers (rbx, r12-r15) are at variable offsets from rbp because they are pushed after the locals and alignment. Their locations must either be computed relative to the aligned rsp or tracked with .cfi_offset directives as each push occurs.
- **Custom epilogue:** The epilogue pops the return address into rcx, adjusts rsp, pushes it back, then rets. CFI must track each step to maintain unwindability.

3.4 Implementation Approach

Two options:

- **Option A - Emit CFI for current frame layout:** Add .cfi_* directives to match the existing prologue/epilogue. This preserves the current code generation and adds CFI as an overlay. The non-standard layout makes the CFI more complex but is entirely feasible. Estimated: 2-3 weeks.
- **Option B - Simplify frame layout first:** Restructure pgen to use a more standard frame layout (e.g., move the 3 metadata words to negative rbp offsets alongside locals, so CFA = rbp + 16 as GDB expects). Then add standard CFI. This simplifies the CFI but requires changing the code generator and runtime. Estimated: 3-5 weeks total, but results in simpler long-term maintenance.

3.5 Effort Estimate

Option A (CFI for current layout): 2-3 weeks. Option B (simplify frame + CFI): 3-5 weeks. Testing across all sample programs and the compiler itself: 1-2 weeks.

Total: 3-5 weeks (Option A) or 4-7 weeks (Option B)

4. Summary

Component	Effort	Complexity	Dependencies
DWARF framework	1-2 weeks	Moderate	DWARF spec knowledge
Base type DIEs	1-2 weeks	Low	Framework
Structured types	2-3 weeks	Moderate	Framework
Advanced types	2-4 weeks	High	Framework + types
Variable locations	1-2 weeks	Moderate	Frame layout knowledge
CFI directives	2-3 weeks	Moderate	Frame layout knowledge
Testing	2-3 weeks	Moderate	All above

Total estimated effort: 11-19 weeks (3-5 months)

4.1 Recommended Implementation Order

- **Phase 1 (weeks 1-3):** CFI directives for backtraces. This is the most immediately useful feature -- developers can see the call stack when a crash occurs. It is also independent of the type system work.
- **Phase 2 (weeks 4-7):** DWARF framework, base types, and global variable descriptions. After this phase, GDB can inspect global integer, real, char, and boolean variables.
- **Phase 3 (weeks 8-12):** Structured types (records, arrays, enums) and local/parameter variable descriptions. This enables full variable inspection for most programs.
- **Phase 4 (weeks 13-19):** Advanced types (variant records, classes, containers). Full type coverage for the complete Pascaline language.

4.2 AI-Assisted Estimate

With an AI coding assistant, the DWARF emission code (which is largely mechanical translation from symbol table entries to DWARF encoding) compresses well. The CFI work is less compressible because it requires careful reasoning about the specific frame layout.

Component	Without AI	With AI	Reason
DWARF framework	1-2 weeks	3-5 days	Boilerplate encoding
Type DIEs	5-9 weeks	3-5 weeks	Mechanical mapping
Variable locations	1-2 weeks	3-5 days	Pattern-based
CFI directives	2-3 weeks	1-2 weeks	Frame-specific reasoning
Testing	2-3 weeks	1-2 weeks	AI generates test cases

Revised total with AI assistance: 7-12 weeks (2-3 months)

4.3 Prerequisites

- DWARF specification (version 4 or 5) -- freely available at dwarfstd.org
- GAS assembler directive reference for .debug_* sections and .cfi_* directives
- Understanding of pgen code generation internals (frame layout, register allocation)
- Understanding of pcom symbol table structure (types, variables, scopes)

- GDB source-level debugging to validate output

4.4 Risk Factors

- **DWARF complexity:** DWARF is a large specification. Implementing the full type system correctly, especially variant records and classes, requires careful attention to detail. Incremental implementation and testing is essential.
- **GDB Pascal support:** GDB has some Pascal language support but it is primarily oriented toward GPC/Free Pascal. Pascaline-specific types (container arrays, classes, properties) may require compromises or approximations in the DWARF representation.
- **Frame layout changes:** If Option B (simplified frame) is chosen for CFI, it requires coordinated changes to the code generator and runtime system (psystem). This carries regression risk.