UNIVERSITY OF WATERLOO

# GNU Debugger support for $\mu$C++ and C$\forall$

by

Thi My Linh Tran

in the

Faculty of Mathematics and Faculty of Engineering

Software Engineering Department

August 2018

# *Preface*

The goal of this work is to add GNU Debugger support for the language $\mu$C++ and C$\forall$. To achieve this goal for $\mu$C++, new extensions are written to provide additional support for high-level constructs that are unknown to the GNU Debugger. In addition to the work done for $\mu$C++, many hooks are added in the GNU Debugger to enable the addition of a C$\forall$ demangler.

This report assumes the reader has basic knowledge of compiler construction. Background knowledge about how GNU Debugger works and specific features of $\mu$C++ and C$\forall$ are provided in the report.

# *Acknowledgements*

I would like to thank Professor Peter Buhr and Thierry Delisle for their guidance throughout the development of this project. The GNU Debugger's internal manual [1] is used as a guide for the development of adding a demangler for C∀, and C∀'s main page [2] for background knowledge and examples.

# Contents

# Listings

# Chapter 1

# Introduction

Computer programming languages provide humans a means to instruct computers to perform a particular task. New programming languages are invented to simplify the task, or provide additional features, enhance performance, and improve developer productivity.

A crucial companion tool to a programming language is a debugger. A debugger is a productivity tool to aid developers in testing and finding bugs in a program. By definition, a debugger executes any program written in one of a supported set of languages and allows developers to stop, monitor and examine state in the program for further investigation.

Specifically, this report talks about how to add GNU Debugger (GDB) support for the concurrent programming-languages $\mu$C++ and C$\forall$. Both languages provide an $M$:$N$ concurrency model, where $M$ user-level threads execute on $N$ kernel-level threads. Often debuggers either do not know about concurrency or only provide a simple understanding of concurrency provided by the operating system (kernel threads). For C$\forall$, new hooks are also added to allow GDB to understand that C$\forall$ is a new source language that requires invocation of a demangler for variable and function names.

Because $\mu$C++ is a translator, all the code written in $\mu$C++ is eventually compiled down to C++ code. This transformation gives $\mu$C++ an advantage with GDB because GDB already understands C++. However, since $\mu$C++ introduced new objects and high-level execution constructs into the language, GDB does not understand these objects or the runtime environment. One objective of this project is to write new GDB extensions to understand

concurrency among tasks, a new high-level execution construct that is discussed more in Chapter 2.

Additionally, if a programming language provides overloading functionality, which allows functions or variables in the same scope with the same identifier, then each of these entities must be assigned a unique name, otherwise, there are name collisions.

However, uniquely numbering (naming) overloads is impossible with separate compilation, because the compiler does not have access to all translation units. Therefore, it is necessary to adopt a scheme related to the overloading mechanism for unique naming. For example, if a language uses the number and types of function parameters to disambiguate function names, than the number and parameter types are encoded into the name:

```
void f( int i, double d, char c );  // f_3_i_d_c
void f( int i, char c );            // f_2_i_c
```

Here, the mangled names for `f` contain the number of parameters and a code for each parameter type. For a complex type-system, the type codes become correspondingly complex, e.g., a generic structure. These names are now unique across all translation units.

Unfortunately, a debugger only has access to the mangled names in a compiled translation units, versus the unmangled names in the program. Therefore, the debugger can only look up mangled names versus original program names, which makes debugging extremely difficult for programmers. To solve this program, the language must provide the debugger with a "demangled" so it can convert mangled names back to program names, and correspondingly retrieve the type of the name. [3]

C∀, a new language being developed at the University of Waterloo, has overloading, so names resolved by the debugger are mangled names. Therefore, another objective of this project is to add a C∀ demangler to GDB.

# Chapter 2

# $\mu$C++

## 2.1 Introduction

$\mu$C++ [4] extends the C++ programming language with new mechanisms to facilitate control flow, and adds new objects to enable lightweight concurrency on uniprocessor and parallel execution on multiprocessor computers. Concurrency has tasks that can context switch, which is a control transfer from one execution state to another that is different from a function call. Tasks are selected to run on a processor from a ready queue of available tasks, and tasks may need to wait for an occurrence of an event.

## 2.2 Tasks

A task behaves like a class object, but it maintains its own thread and execution state, which is the state information required to allow independent execution. A task provides mutual exclusion by default for calls to its public members. Public members allow communication among tasks.

## 2.3 $\mu$C++ Runtime Structure

$\mu$C++ introduces two new runtime entities for controlling concurrent execution:

- Cluster
- Virtual processor

### 2.3.1 Cluster

A cluster is a group of tasks and virtual processors (discussed next) that execute tasks. The objective of a cluster is to control the amount of possible parallelism among tasks, which is only feasible if there are multiple hardware processors (cores).

At the start of a $\mu$C++ program, two clusters are created. One is the system cluster and the other is the user cluster. The system cluster has a processor that only performs management work such as error detection and correction from user clusters, if an execution in a user cluster results in errors, and cleans up at shutdown. The user cluster manages and executes user tasks on its processors. The benefits of clusters are maximizing utilization of processors and minimizing runtime through a scheduler that is appropriate for a particular workload. Tasks and virtual processors may be migrated among clusters.

### 2.3.2 Virtual Processor

A virtual processor is a software emulation of a processor that executes threads. Kernel threads are used to implement a virtual processor, which are scheduled for execution on a hardware processor by the underlying operating system. The operating system distributes kernel threads across a number of processors assuming that the program runs on a multi-processor architecture. The usage of kernel threads enables parallel execution in $\mu$C++.

# Chapter 3

# GNU Debugger

## 3.1 Introduction

The GNU Project Debugger is a program that allows examination of what is going on inside another program while it executes, or examines the state of a program after it crashed [5].

## 3.2 Debug Information

In order to allow effective inspection of program state, the debugger requires debugging information for a program. Debugging information is a collection of data generated by a compiler and/or an assembler program. This information is optional as it is only required for compilation, and hence, it is normally not present during program execution when debugging occurs. When requested, debugging information is stored in an object file, and it describes information such as the type of each variable or function and the correspondence between source line numbers and addresses in the executable code [6]. Debugging information is requested via the `-g` flag during the compilation stage of the program.

The debugging information must be written out in a canonical format for debuggers to read. DWARF is one of the supported debugging data formats, and its architecture is independent and applicable to any language, operating system, or processor [7]. This format uses a data

structure called DIE to represent each variable, type, function, etc. A DIE is a pair: tag and its attribute [8].

## 3.3 Stack-Frame Information

A stack frame, or frame for short, is a collection of all data associated with one function call. A frame consists of parameters received from the function call, local variables declared in that function, and the address where the function returns. The frame-pointer register stores the address of a frame, during execution of a call. A call stack can have many frames [9].

## 3.4 Extending GDB

GDB provides three mechanisms for extending itself. The first is composition of GDB commands, the second is using the Python GDB API, and the third is defining new aliases for existing commands.

## 3.5 Symbol Handling

Symbols are a key part of GDB's operation. Symbols can be variables, functions and types. GDB has three kinds of symbol tables:

- Full symbol-tables (symtabs): These contain the main information about symbols and addresses
- Partial symbol-tables (psymtabs): These contain enough information to know when to read the corresponding part of the full symbol-table.
- Minimal symbol-tables (msymtabs): These contain information extracted from non-debugging symbols.

Debugging information for a large program can be very large, and reading all of these symbols can be a performance bottleneck in GDB, affecting the user experience. The solution is to lazily construct partial symbol-tables consisting of only selected symbols, and then eagerly expand them to full symbol-tables when necessary. The psymtabs is constructed by doing a quick pass over the executable file's debugging information.

## 3.6 Name Demangling in GDB

The library `libiberty` provides many functions and features that can be divided into three groups:

- Supplemental functions: additional functions that may be missing in the underlying operating system.
- Replacement functions: simple and unified equivalent functions for commonly used standard functions.
- Extensions: additional functions beyond the standard.

In particular, this library provides the C++ demangler that is used in GDB and by $\mu$C++. A new demangler can also be added in this library, which is what Rust did, and what is necessary for C$\forall$.

# Chapter 4

# C∀

## 4.1 Introduction

Similar to C++, C is a popular programming language especially in systems programming. For example, the Windows NT and Linux kernel are written in C, and they are the foundation of many higher level and popular projects. Therefore, it is unlikely that the programming language C is going to disappear any time soon.

However, C has inherent problems in syntax, semantics and many more [10]. Even though C++ is meant to fix these problems, C++ has many irreversible legacy design choices, and newer versions of C++ require significantly more effort to convert C-based projects into C++.

To solve this problem, the programming language C∀ is being created at the University of Waterloo. The goal of the language is to extend C with modern language features that many new languages have, such as Rust and Go. Hence, the C∀ extension provides a backward-compatible version of C, while fixing existing problems known in C and modernizing the language at the same time.

## 4.2 Overloading

Overloading is when a compiler permits a name to have multiple meanings. All programming languages allow simple overloading of operators on basic types such as the `+` operator (add) on integer and floating-point types. Most programming languages extend overloading of operators to user-defined types and/or general function overloading. C∀ also supports overloading of variables and the literals `0/1`.

### 4.2.1 Variable

Variables in the same block are allowed to have the same name but different types. An assignment to a new variable uses that variable's type to infer the required type, and that type is used to select a variable containing the appropriate type.

```
1 short int MAX = SHRT_MAX;
2 int MAX = INT_MAX;
3 double MAX = DBL_MAX;
4
5 // select variable MAX based on its left-hand type
6 short int s = MAX;       // s = SHRT_MAX
7 int s = MAX;             // s = INT_MAX
8 double s = MAX;          // s = DBL_MAX
```

Listing 4.1: Overloading variables in C∀

The listing 4.1 shows that when variable overloading exists in the same scope, the variable is selected based on the left side of initialization/assignment and operands of the right side of the expression. For instance, the first assignment to variable `s` at line 6, which is type short int, selects the MAX with the same type.

### 4.2.2 Function

Functions in the same block can be overloaded depending on the number and type of parameters and returns.

```
1 void f(void);                 // (1)
2 void f(char);                 // (2)
3 char f(void);                 // (3)
4 [int,double] f();             // (4)
5
6 f();                          // pick (1)
7 f('a');                       // pick (2)
8 char s = f('a');              // pick (3)
9 [int, double] s = f();        // pick (4)
```

LISTING 4.2: Overloading routines in C∀

The listing 4.2 shows that when many functions are overloaded in the same scope, a function is selected based on the combination of its return type and its arguments. For instance, from line 1-4, four different types of a function called `f` are declared. For the call `f('a')`, the function selected is the one on line 2, if the call voids the result. However, if the call assigns to a char, then the routine on line 3 is selected. This example can be seen on lines 7-8.

### 4.2.3 Operator

An operator name is denoted with `?` for the operand and any standard C operator. Operator names within the same block can be overloaded depending on the number and type of parameters and returns. However, operators `&&`, `||`, `?:` cannot be overloaded because short-circuit semantics cannot be preserved.

```
1 int ++?(int op);              // unary prefix increment
2 int ?++(int op);              // unary postfix increment
3 int ?+?(int op1, int op2); // unary postfix increment
4
5 struct S { double x, double y }
6
7 // overload operator plus-assignment
```

```
8  S ?+?(S a, S b) {
9      return (S) {a.x + b.x, a.y + b.y};
10 }
11
12 S a, b, c;
13 a + b + c;
```

Listing 4.3: Overloading operators in C∀

The listing 4.3 shows that operator overloading is permitted similar to C++. However, the difference is that the operator name is denoted with `?` instead, and operator selection uses the return type.

# Chapter 5

# Extending GDB for $\mu$C++

## 5.1 Introduction

A sequential program has a single call stack. A debugger knows about this call stack and provides commands to walk up/down the call frames to examine the values of local variables, as well as global variables. A concurrent program has multiple call stacks (for coroutines/tasks), so a debugger must be extended to locate these stacks for examination, similar to a sequential stack. For example, when a concurrent program deadlocks, looking at the task's call stack can locate the resource and the blocking cycle that resulted in the deadlock. Hence, it is very useful to display the call stack of each task to know where it is executing and what values it is currently computing. Because each programming language's concurrency is different, GDB has to be specifically extended for $\mu$C++.

## 5.2 Design Constraints

As mentioned in Chapter 3, there are several ways to extend GDB. However, there are a few design constraints on the selected mechanism. All the functions implemented should maintain similar functionality to existing GDB commands. In addition to functional requirements, usability and flexibility are requirements for this project. These final requirements

enable developers to be productive quickly and do more with the extensions. The extensions created for μC++ are simple to use and versatile.

The following new GDB command are all implemented through the Python API for GDB. Python is a scripting language with built-in data structures and functions that enables the development of more complex operations and saves time on development.

## 5.3 μC++ source-code example

Listing 5.1 shows a μC++ program that implicitly creates two clusters, system and user, which implicitly have a processor (kernel thread) and processor task. The program explicitly creates three additional processors and ten tasks on the user cluster.

## 5.4 Existing User-defined GDB Commands

Listing 5.2 shows the GDB output at the base case of the recursion for one of the tasks created in the μC++ program in listing 5.1. The task is stopped at line 8. The backtrace shows the three calls to function `a`, started in the task's `main`. The top two frames are administrative frames from μC++. The values of the argument and local variables are printed.

### 5.4.1 Listing all clusters in a μC++ program

Listing 5.3 shows the new command `clusters` to list all program clusters along with their associated address. The output shows the two μC++ implicitly created clusters.

### 5.4.2 Listing all processors in a cluster

Listing 5.4 shows the new command `processors`, which requires a cluster argument to show all the processors in that cluster. In particular, this example shows that there are four processors in the `userCluster`, with their associated address, PID, preemption and spin.

```
1  _Task T {
2      std::string name;
3
4      void a(int param) {
5          if ( param != 0 ) a( param - 1 );
6          int x = 3;
7          std::string y = "example";
8      }
9      void main() {
10         a(3);
11     }
12   public:
13     T(const int tid) {
14         name = "T" + std::to_string(tid);
15         setName(tid.c_str());
16     }
17 };
18
19 int main() {
20     uProcessor procs[3];          // extra processors
21     const int numTasks = 10;
22     T* tasks[numTasks];           // extra tasks
23
24     // allocate tasks with different names
25     for (int id = 0; id < numTasks; id += 1) {
26         tasks[i] = new T(id);
27     }
28     // deallocate tasks
29     for (int id = 0; id < numTasks; id += 1) {
30         delete tasks[i];
31     }
32 }
```

LISTING 5.1: μC++ source code used for GDB commands

### 5.4.3 Listing all tasks in all clusters

Listing 5.5 shows the new command `task` without an argument to list all the tasks in a μC++ program at this point in the execution snapshot.

```
(gdb) backtrace
#0  T::a (this=0x907880, param=0) at test.cc:8
#1  0x000000000041d207 in T::a(this=0x907880, param=1) at test.cc:5
#2  0x000000000041d207 in T::a(this=0x907880, param=2) at test.cc:5
#3  0x000000000041d207 in T::a(this=0x907880, param=3) at test.cc:5
#4  0x000000000041d2c0 in T::main(this=0x907880) at test.cc:10
#5  0x000000000042633f in ...::invokeTask(uBaseTask&) at ...
#6  0x0000000000000000 in ?? ()
(gdb) info args
this = 0x907880
param = 0
(gdb) info locals
x = 3
y = "example"
```

LISTING 5.2: Call stack of function `a` in the μC++ program from listing 5.1

```
(gdb) clusters
        Name             Address
    systemCluster        0x655280
    userCluster          0x7c4280
```

LISTING 5.3: clusters command

```
(gdb) processors userCluster
    Address     PID          Preemption       Spin
    0x7c6bf0     8396928      10               1000
    0x8c3b60     9453696      10               1000
    0x8c3d20     9977984      10               1000
    0x8c3ee0     10506368     10               1000
```

LISTING 5.4: processors command

### 5.4.4 Listing all tasks in a cluster

Listing 5.6 shows the new command `task` with a cluster argument to list only the names of the tasks on that cluster. In this version of the command `task`, the associated address for each task and its state is displayed.

```
(gdb) task
        Cluster Name         Address
        systemCluster        0x655280
    ID       Task Name            Address          State
    0        uProcessorTask      0x6c39b0        uBaseTask::Blocked
    1        uSystemTask         0x783ef0        uBaseTask::Blocked
        Cluster Name         Address
        userCluster          0x7c4280
    ID       Task Name            Address          State
    0        uProcessorTask      0x806e70        uBaseTask::Blocked
    ...
    6        T0                  0xa49830        uBaseTask::Ready
    7        T1                  0xa89bb0        uBaseTask::Running
    8        T2                  0xac9f30        uBaseTask::Ready
    ...
    15       T9                  0xc8b7b0        uBaseTask::Ready
```

LISTING 5.5: task command for displaying all tasks for all clusters

```
(gdb) task systemCluster
    ID           Name             Address          State
    0        uProcessorTask      0x654200        uBaseTask::Blocked
    1        uSystemTask         0x7c3280        uBaseTask::Blocked
```

LISTING 5.6: task command for displaying all tasks in a cluster

## 5.5 Changing Stacks

The next extension for displaying information is writing new commands that allow stepping from one μC++ task to another. Two switching approaches are provided: structured and free form. The structured form remembers the task tour in a LIFO way, while the free form does not remember the task tour. This semantics means at least two commands are needed for the structured form and the free form. Taking the idea of overloading in programming languages, the structured commands for switching from the current task to a different task overload the command `task` with two arguments, and for switching back to the previous task is a new command `poptask`.

### 5.5.1 Task Switching

Both overloaded `task` or `poptask` commands allow two ways to designate a task: its absolute address or relative id within a cluster. For instance, to switch from any task to task `T1` seen in listing 5.5, the first command in listing 5.7 uses an absolute address (0xa49830) and the second command uses a relative id (6) with respect to userCluster. Core functionality of these approaches is essentially the same.

```
task 0xa49830
task userCluster 6
```

LISTING 5.7: task command option

### 5.5.2 Switching Implementation

To implement the task tour, it is necessary to store the context information for every context switching. This requirement means the `task` command needs to store this information every time it is invoked.

Figure 5.1 shows a task points to a structure containing a `uContext_t` data structure, storing the stack and frame pointer, and the stack pointer. Listing 5.8 shows these pointers are copied into an instance of the Python tuple `StackInfo` for every level of task switching. This tuple also stores information about the program counter that is calculated from the address of the `uSwitch` Assembly function because a task always stops in `uSwitch` when its state is blocked. Similarly, switching commands retrieve this context information but from the task that a user wants to switch to, and sets the equivalent registers to the new values.

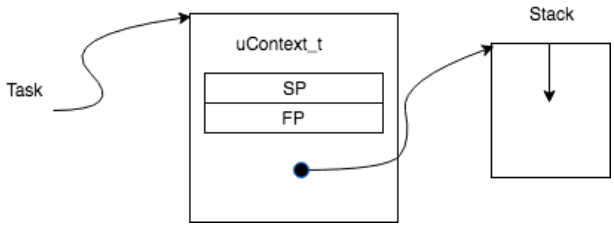


FIGURE 5.1: Machine context (uMachContext) for each task

```
1  # get GDB type of uContext_t*
2  uContext_t_ptr_type =
3     gdb.lookup_type('UPP::uMachContext::uContext_t').pointer
4
5  # retrieve the context object from a task
6  # and cast it to the type uContext_t*
7  task_context = task['context'].cast(uContext_t_ptr_type)
8
9  # the offset where sp would be starting from uSwitch function
10 sp_address_offset = 48
11 # lookup the value of stack pointer (sp), frame pointer (fp),
12 # program counter (pc)
13 xsp = task_context['SP'] + sp_address_offset
14 xfp = task_context['FP']
15 if not gdb.lookup_symbol('uSwitch'):
16     print('uSwitch symbol is not available')
17     return
18
19 # This value is calculated here because we always here when
20 # the task is in blocked state
21 xpc = get_addr(gdb.parse_and_eval('uSwitch').address + 28, 'PC')
22 # must switch back to frame-0 to set 'pc' register with
23 # the value of xpc
24 gdb.execute('select-frame 0')
25
26 # retrieve register values and push sp, fp, pc into
27 # a global stack
28 global STACK
29 sp = gdb.parse_and_eval('$sp')
30 fp = gdb.parse_and_eval('$fp')
31 pc = gdb.parse_and_eval('$pc')
32 stack_info = StackInfo(sp = sp, fp = fp, pc = pc)
33 STACK.append(stack_info)
34
35 # update registers for a new task
36 gdb.execute('set $rsp={}'.format(xsp))
37 gdb.execute('set $rbp={}'.format(xfp))
38 gdb.execute('set $pc={}'.format(xpc))
```

LISTING 5.8: Abridged `push_task` source code

For free-form touring using the `task` command, the value of the stack pointer is copied to/from the `rsp` register, the value of the frame pointer is copied to/from the `rbp` register, and finally the value of the program counter is copied to/from the `pc` register.

For structured touring, the implementation manages a stack of toured tasks. The `pushtask`

```
1  (gdb) info threads
2    Id   Target Id         Frame
3  * 1    Thread 0x7ffff7fe9780 (LWP 704) "utils" 0x000000000042f2ef in
4          uCluster::processorPause (this=0x73f7a8) at
5          /usr/local/u++-7.0.0/src/kernel/uCluster.cc:120
6    2    Thread 0x802080 (LWP 708) "utils" T::a (this=0xa49830) at utils.cpp:7
7    3    Thread 0x904080 (LWP 709) "utils" T::a (this=0xbcad30) at utils.cpp:7
8    4    Thread 0x984080 (LWP 710) "utils" T::a (this=0xb8a9b0) at utils.cpp:7
9    5    Thread 0xa05080 (LWP 711) "utils" UPP::uSigHandlerModule::sigAlrmHandler
10         (sig=10, sfp=0xb864b0, cxt=0xb86380) at
11         /usr/local/u++-7.0.0/src/kernel/uSignal.cc:218
12 (gdb) thread 2
13 (gdb) backtrace
14 #0 T::a (this=0xb0a2b0, param=5) at utils.cpp:8
15 #1  0x000000000041cb4c in T::main (this=0xb0a2b0) at
16 utils.cpp:11
17 #2  0x0000000000425bbc in UPP::uMachContext::invokeTask (This=...)
18     at /usr/local/u++-7.0.0/src/kernel/uMachContext.cc:140
19 #3  0x0000000000000000 in ?? ()
20 (gdb) task userCluster 6
21 (gdb) backtrace
22 #0  uSwitch () at /usr/local/u++-7.0.0/src/kernel/uSwitch-x86_64.S:64
23 #1  0x00000000004288ec in uBaseCoroutine::taskCxtSw (this=0x8c3b78)
24     at /usr/local/u++-7.0.0/src/kernel/uBaseCoroutine.cc:146
25 ...
26 #7 T::a (this=0xa49830) at utils.cpp:8
27 #8  0x000000000041cb43 in T::main (this=0xa49830) at utils.cpp:11
28 #9  0x0000000000425bbc in UPP::uMachContext::invokeTask (This=...)
29     at /usr/local/u++-7.0.0/src/kernel/uMachContext.cc:140
30 #10 0x0000000000000000 in ?? ()
31 (gdb) poptask
32 (gdb) backtrace
33 #0 T::a (this=0xb0a2b0, param=5) at utils.cpp:8
34 #1  0x000000000041cb4c in T::main (this=0xb0a2b0) at utils.cpp:11
35 #2  0x0000000000425bbc in UPP::uMachContext::invokeTask (This=...)
36     at /usr/local/u++-7.0.0/src/kernel/uMachContext.cc:140
37 #3  0x0000000000000000 in ?? ()
```

LISTING 5.9: `poptask` command

command sets the appropriate registers to move to the specified task, and remembers the previous task registers. The `poptask` restores the previous stack register. Popping an empty stack prints a warning. In particular, listing 5.9 shows the backtrace after touring to task 6 of `userCluster`, whose address can be seen on line 23. However, after calling the command `poptask`, the new backtrace shows the address of the previous task, which is `0xb0a2b0` and seen on lines 11 and 30.

```
continue
next
nexti
step
stepi
finish
advance
jump
signal
until
reverse-next
reverse-step
reverse-stepi
reverse-continue
reverse-finish
```

LISTING 5.10: Built-in GDB commands that allow continuation of a program

### 5.5.3 Continuing Implementation

When a breakpoint or error is encountered, all concurrent execution stops. The state of the program can now be examined and changed; after which the program can be continued. Continuation must always occur from the top of the stack (current call) for each task, and at the specific task where GDB stopped execution.

However, during a task tour, the new GDB commands change the notion of the task where execution stopped to make it possible to walk other stacks. Hence, it is a requirement for continuation that the task walk always return to frame-0 of the original stopped task before any program continuation [11].

To provide for this requirement, the original stop task is implicitly remembered, and there is a new `reset` command that *must* be explicitly executed before any continuation to restore the locate state. To prevent errors from forgetting to call the `reset` command, additional hooks are added to the existing built-in GDB continuation commands (see 5.10) to implicitly call `reset`. The list of these commands results from GDB documentation [12] and similar work done for KOS [13].

## 5.6 Result

The current implementation successfully allows users to display a snapshot of μC++ execution with respect to clusters, processors, and tasks. With this information it is possible to tour the call stacks of the tasks to see execution locations and data values. Additionally, users are allowed to continue the execution where the program last pauses assuming that the program has not crashed. The continuation of execution is done by automatically reversing the task walk from any existing GDB commands such as `continue`, or a user can manually reverse the task walk using the command `poptask` and then continue.

# Chapter 6

# C∀ Demangler

## 6.1 Introduction

While C∀ is a translator for additional features that C does not support, all the extensions compiled down to C code. As a result, the executable file marks the DWARF tag `DW_AT_language` with the fixed hexadecimal value for the C language. Because it is possible to have one frame in C code and another frame in Assembly code, GDB encodes a language flag for each frame. C∀ adds to this list, as it is essential to know when a stack frame contains mangled names versus C and assembler unmangled names. Thus, GDB must be told C∀ is a distinct source-language.

## 6.2 Design Constraints

Most GDB targets use the DWARF format. The GDB DWARF reader initializes all the appropriate information read from the DIE structures in object or executable files, as mentioned in Chapter 3. However, GDB currently does not understand the new DWARF language-code assigned to the language C∀, so the DWARF reader must be updated to recognize C∀.

Additionally, when a user enters a name into GDB, GDB needs to lookup if the name exists in the program. However, different languages may have different hierarchical structure

for dynamic scope, so an implementation for nonlocal symbol lookup is required, so an appropriate name lookup routine must be added.

## 6.3 Implementation

Most of the implementation work discussed below is from reading GDB's internals wiki page and understanding how other languages are supported in GDB [1].

A new entry is added to GDB's list of language definition in `gdb/defs.h`. Hence, a new instance of type `struct language_def` must be created to add a language definition for C∀ . This instance is the entry point for new functions that are only applicable to C∀. These new functions are invoked by GDB during debugging if there are operations that are applicable specifically to C∀. For instance, C∀ can implement its own symbol lookup function for non-local variables because C∀ can have a different scope hierarchy. The final step for registering C∀ in GDB, as a new source language, is adding the instance of type `struct language_def` into the list of language definitions, which is found in `gdb/language.h`. This setup is shown in listing 6.1.

The next step is updating the DWARF reader, so the reader can translate the DWARF code to an enum value defined above. However, this assumes that the language has an assigned language code. The language code is a hexadecimal literal value assigned to a particular language, which is maintained by GCC. For C∀, the hexadecimal value `0x0025` is added to `include/dwarf2.h` to denote C∀, which is shown in listing 6.2.

```
1 // In include/dwarf2.h
2 enum dwarf_source_language { // Source language names and codes.
3     DW_LANG_C89 = 0x0001 ,
4     ...
5     DW_LANG_CForAll = 0x0025 ,
6 }
```

Listing 6.2: DWARF language code for C∀

Once the demangler implementation goes into the `libiberty` directory along with other demanglers, the demangler can be called by updating the language definition defined in

```
1  // In gdb/language.h
2  extern const struct language_defn cforall_language_defn;
3
4  // In gdb/language.c
5  static const struct language_defn *languages[] = {
6      &unknown_language_defn,
7      &auto_language_defn,
8      &c_language_defn,
9      ...
10     &cforall_language_defn,
11     ...
12  }
13
14 // In gdb/cforall-lang.c
15 extern const struct language_defn cforall_language_defn = {
16     "cforall",                      /* Language name */
17     "CForAll",                      /* Natural name */
18     language_cforall,
19     range_check_off,
20     case_sensitive_on,
21     ...
22     cp_lookup_symbol_nonlocal,      /* lookup_symbol_nonlocal */
23     ...
24     cforall_demangle,               /* Language specific demangler */
25     cforall_sniff_from_mangled_name,
26     ..
27 }
```

Listing 6.1: Language definition declaration for C∀

listing 6.1 with the entry point of the C∀ demangler, and adding a check if the current demangling style is `CFORALL_DEMANGLING` as seen in listing 6.3. GDB then automatically invokes this C∀ demangler when GDB detects the source language is C∀. In addition to the automatic invocation of the demangler, GDB provides an option to manually set which demangling style to use in the command line interface. This option can be turned on for C∀ in GDB by adding a new enum value for C∀ in the list of demangling styles along with setting the appropriate mask for this style in `include/demangle.h`. After doing this step, users can now choose if they want to use the C∀ demangler in GDB by calling `set demangle-style <language>`, where the language name is defined by the preprocessor macro `CFORALL_DEMANGLING_STYLE_STRING` in listing 6.4.

However, the setup for the C∀ demangler above does not demangle mangled symbols during

```
// In libiberty/cplus-dem.c
const struct demangler_engine libiberty_demanglers[] = {
    {
        NO_DEMANGLING_STYLE_STRING,
        no_demangling,
        "Demangling disabled"
    },
    ...
    {
        CFORALL_DEMANGLING_STYLE_STRING,
        cforall_demangling,
        "Cforall style demangling"
    },
}
...
char * cplus_demangle(const char *mangled, int options) {
    ...
    /* The V3 ABI demangling is implemented elsewhere.  */
    if (GNU_V3_DEMANGLING || RUST_DEMANGLING || AUTO_DEMANGLING) { ... }
    ...
    if (CFORALL_DEMANGLING) {
        ret = cforall_demangle (mangled, options);
        if (ret) { return ret; }
    }
}
```

Listing 6.3: libiberty setup for the C∀ demangler

```
// In gdb/demangle.h
#define DMGL_CFORALL (1 << 18)
...
/* If none are set, use 'current_demangling_style' as the default. */
#define DMGL_STYLE_MASK
(DMGL_AUTO|DMGL_GNU|DMGL_LUCID|DMGL_ARM|DMGL_HP|DMGL_EDG|DMGL_GNU_V3
|DMGL_JAVA|DMGL_GNAT|DMGL_DLANG|DMGL_RUST|DMGL_CFORALL)
...
extern enum demangling_styles {
    no_demangling = -1,
    unknown_demangling = 0,
    ...
    cforall_demangling = DMGL_CFORALL
} current_demangling_style;
...
#define CFORALL_DEMANGLING_STYLE_STRING  "cforall"
...
#define CFORALL_DEMANGLING (((int)CURRENT_DEMANGLING_STYLE)&DMGL_CFORALL)
```

Listing 6.4: Setup C∀ demangler style

symbol-table lookup while the program is in progress. Therefore, additional work needs to be done in `gdb/symtab.c`. Prior to looking up the symbol, GDB attempts to demangle the name of the symbol, which can either be a mangled or unmangled name, to see if it can detect the language, and select the appropriate demangler to demangle the symbol. This work enables invocation of the C∀ demangler during symbol lookup.

```
1  // In gdb/symtab.c
2  const char * demangle_for_lookup ( const char *name, enum language lang,
3                                     demangle_result_storage &storage ) {
4      /* When using C++, D, or Go, demangle the name before doing a
5         lookup to use the binary search. */
6      if (lang == language_cplus) {
7          char *demangled_name = gdb_demangle(name, DMGL_ANSI|DMGL_PARAMS);
8          if (demangled_name != NULL)
9              return storage.set_malloc_ptr (demangled_name);
10     }
11     ...
12     else if (lang == language_cforall) {
13         char *demangled_name = cforall_demangle (name, 0);
14         if (demangled_name != NULL)
15             return storage.set_malloc_ptr (demangled_name);
16     }
17     ...
18 }
```

Listing 6.5: C∀ demangler setup for symbol lookup

## 6.4 Result

The addition of hooks throughout GDB enables the invocation of the new C∀ demangler during symbol lookup and during the usage of `binutils` tools such as `objdump` and `nm`. Additionally, these `binutils` tools also understand C∀ because of the addition of the C∀ language code. However, as the language develops, symbol lookup for non-local variables must be implemented to produce the correct output.

# Chapter 7

# Conclusion

New GDB extensions are created to support information display and touring among $\mu$C++ user tasks, and new hooks are added to the GNU Debugger to support a C$\forall$ demangler. The GDB extensions are implemented using the Python API, and the hooks to add debugging support for C$\forall$ are implemented using $\mu$C++ . For the GDB extensions, writing Python extensions is easier and more robust. Furthermore, GDB provides sufficient hooks to make it possible for a new language to leverage existing infrastructure and codebase to add debugging support. The result provides significantly new capabilities for examining and debugging both $\mu$C++ and C$\forall$ programs.

# Bibliography


[1] *GDB Internals Manual*, 2018. URL https://sourceware.org/gdb/wiki/Internals.

[2] C∀ Language, 2017. URL https://cforall.uwaterloo.ca.

[3] Name Mangling, 2018. URL https://en.wikipedia.org/wiki/Name_mangling.

[4] Peter A. Buhr. $\mu$C++ Annotated Reference Manual. Technical report, School of Computer Science, University of Waterloo, January 2006. URL https://plg.uwaterloo.ca/~usystem/pub/uSystem/uC++.pdf.

[5] *GDB: The GNU Project Debugger*, 2018. URL https://www.gnu.org/software/gdb/.

[6] *Compiling for Debugging*, 2018. URL https://sourceware.org/gdb/onlinedocs/gdb/Compilation.html#Compilation.

[7] DWARF Debugging Standard, 2007. URL http://dwarfstd.org/.

[8] DWARF, 2017. URL https://en.wikipedia.org/wiki/DWARF.

[9] Examining the Stack, 2002. URL ftp://ftp.gnu.org/old-gnu/Manuals/gdb/html_chapter/gdb_7.html.

[10] Andrew Koenig. Pitfalls of C, 2001. URL http://www.math.pku.edu.cn/teachers/qiuzy/c/reading/pitfall.htm.

[11] Selecting A Frame, 2002. URL ftp://ftp.gnu.org/old-gnu/Manuals/gdb/html_chapter/gdb_7.html#SEC44.

[12] Continuing and Stepping, 2018. URL https://sourceware.org/gdb/onlinedocs/gdb/Continuing-and-Stepping.html.

[13] KOS, 2018. URL https://cs.uwaterloo.ca/~mkarsten/kos.html.