Design and Compilation of a C-like front-end language for GPRM

Ross Meikleham

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 2015

Abstract

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: ____________________ Signature: ____________________

Contents

Chapter 1

Introduction

1.1 GPRM

1.1.1 What is the GPRM

The Glasgow Parallel Reduction Machine is a virtual machine framework for multi-core programming using a task-based approach. It allows the programmer to structure their programs as a seperation of task-code (written as C++ classes) and communication code.

Communication code is currently written in a language called GPIR (Glasgow Parallel Intermediate Representation). GPIR code controls how tasks communicate with one another and whether groups of tasks can be evaluated sequentially or in parallel. GPIR code is compiled down further to GPRM byte-code which is evaluated by the GPRM virtual machine.

The GPRM uses task nodes which consists of a task kernel and a task manager.

Task code is represeted as a task kernel. A task kernel is a self contained unit, typically represented as a C++ class. To create a task kernel, the C++ class needs to be in the *GPRM::Kernel::namespace*.

Communication code is represented as a task manager. A task manager ”co-ordinates” communication between one or more task kernels, and is represented as a function which can be called from a C++ program.[14]

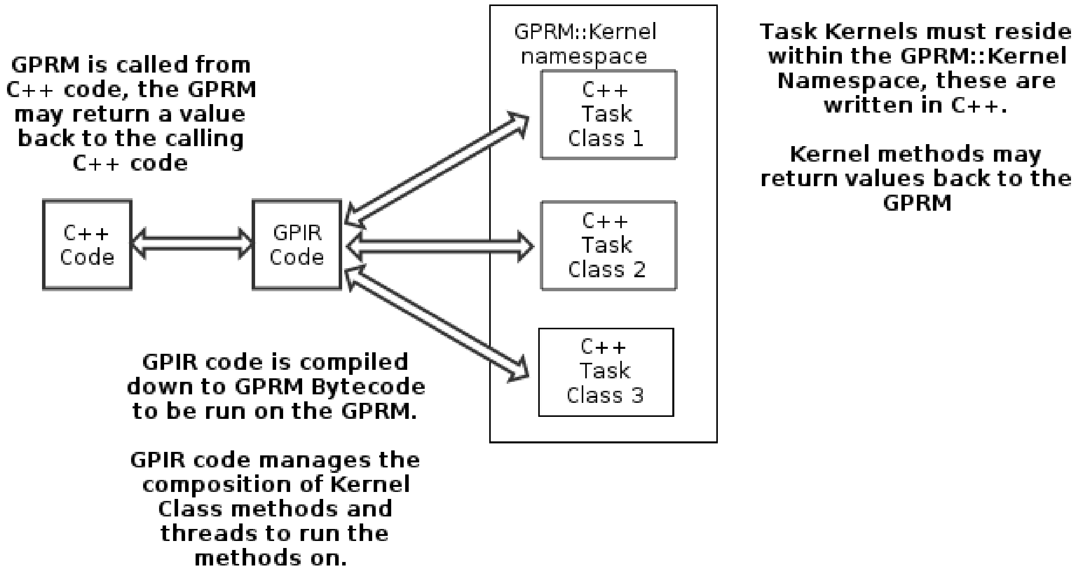


Figure 1.1: A simple overview of the GPRM framework

1.1.2 The GPIR language

The GPIR language is a purely functional S-expression based language that is evaluated in parallel by default with optional sequential evaluation semantics.

Tasks in GPIR are postifxed with a thread number to indicate to the GPRM runtime which thread the task should run on. For example a simple GPIR program which adds numbers in parallel:

```
(begin
    +[0] (+[0] '3 '2) (+[1] '4 '10)
)
```

The two nested additions are performed in parallel, with the first being mapped onto thread 0, and the second being mapped onto thread 1. When they've both been evaluated, then the outer addition with add the results on thread 0.

Quoting

Like in Scheme and Lisp, quoting deffers the evaluation. This is useful for performing sequential evaluation.

```
(seq
    '(obj.m1[0] '1)
    '(obj.m2[0] '2)
)
```

Due to the parallel evaluation of GPIR, if the expressions in the seq block wern't quoted then they would get evaluated in parallel instead of being deffered to be evaluated by the seq function.

Also literal values don't need to be evaluated, they should be deferred and passed to tasks which is the reason the numbers are quoted in these examples.

The GPIR language has other keywords and features but these are the ones that are important for understanding the examples and design choices made for this project.

1.2 Project Aims

The GPIR language isn't really suitable for programming in. For one it requires the programmer to manually manage which thread each task is allocated to. The language is also inconsistant with the C++ language used for the other parts of the framework (Calling code and Class Kernels). A language closer to C/C++ is more more consistent with the entire framework.

The aim of this project is to design a C-like language which can be evaluated in parallel by default, and build a compiler for it. The compiler should be able to compile this new language down to GPIR code. The new language will be called " Glasgow Parallel C" or "GPC" for short.

The language should be easy to pick up and write programs in for anyone familiar with the C/C++ languages. To achieve this the language should be as close to C/C++ as possible.

The language should also abstract away the details of allocating tasks to threads, this job should be part of the compiler.

1.3 Current C/C++ Parallel Programming Models

By researching available C/C++ parallel programming frameworks/language extensions we can determine possible features and design choices that may suitable for the GPC language.

1.3.1 Cilk Plus

Cilk Plus is a general purpose programming language based on C++. It extends the C++ language with features such as parallel for loops and spawning functions in parallel using a "fork-join" model to achieve task-parallelism.

One of the main princeples of the Cilk language is that abstraction is important and that the programmer should use provided constructs to expose the parallelism in their application, allowing the programmer to be free to focus on what the code is allowed to execute in parallel and not worry about the underlying details of manually managing threads. The runtime should then have the responsibility of scheduling the threads, and dividing work between processors.[2].

Cilk Plus introduces 3 new keywords ontop of the C++ language[4]:

- *cilk_for* - Parallelizing for loops, uses the exact same syntax as the standard C++ for-loop with some restrictions.

- *cilk_spawn* - Indicate that a given function can run in parallel with the remainder of the calling function.
- *cilk_sync* - Wait for all spawned calls to finish.

Cilk Plus applications have ”serial semantics”[4], this means that the results of an application run in parallel with Cilk Plus would be exactly the same if it were run serially (*cilk_spawn* becoming a function call, removing *cilk_sync* statements, and replacing *cilk_for* with ordinary for loops).

Cilk Plus makes use of pragmas to indicate to the compiler that a for loop contains data parallelism.[2]

Cilk Plus also introduces a new operator **[:]** to select array sections[1]. This operator allows for ”high level” operaions to be performed on arrays, and can help the compiler vectorize parts of code.

The Cilk Plus runtime makes use of ”task stealing” for dynamic load balancing[2]. This means that if one thread is idle the scheduler can reassign work assigned to be completed by a busier thread. The outcome of this is that the programmer doesn’t have to worry about the specifics of which threads to map tasks to, and it can be left to the rutime itself.

1.3.2 Open MP

OpenMP (Open Multi-Processing) is a language extension available for C, C++ and Fortran which allows for shared memory multi-processing. This is achieved by the use of compiler directives (more specifically in C/C++ this is done through the preprocessor using pragmas) and the OpenMP API.[8].

OpenMP’s use of pragmas for parallel programming means that parallel code keeps sequential semantics and any compiler which doesn’t implement OpenMP extensions can still compile the code by ignoring the pragmas. The results of executing the program serially without the pragmas should be exactly the same as when the sections are parallelized. Another benefit is that sections of serial programs can be parallelized by adding pragmas and existing code doesn’t need to be modified.

For example, given a simple for loop below:

```
for(int i = 0; i < 10; i++) {
    arr[i] =  do_calculation(i);
}
```

We can parallelize this code by simply adding a pragma above the for loop:

```
#pragma omp parallel for
for(int i = 0; i < 10; i++) {
    arr[i] =  do_calculation(i);
}
```

OpenMP also supports task parallelism as of version 3.0[9]. The specification for OpenMP does not specify how the scheduler should work, and no specific implementations of OpenMP appear to have implemented a task stealing scheduler.

A downside to using pragmas is that accidentally missing certain directive keywords may cause undersirable program behaviour such as unecessary parallelization[12], and no warnings will be given by the compiler.

1.3.3 Intel TBB

Intell TBB (Intell Thread Building Blocks) is a portable C++ template library for task parallelism. It contains a range of concurrent algorithms, containers, and it's own task scheduler to achieve this[7].

Operations are treated as tasks, and the task scheduler has the job of dynamically allocating these tasks to individual cores which abstracts the specific details of allocating threads from the Programmer. Like CilkPlus, Intell TBB's task scheduler also implements task stealing for dynamic load balancing[6].

Intell TBB relies on generic programming which allows for writing parallel algorithms based on requirements on types. Parallel code can be written once and can work with lots of different types without having to rewrite the algorithm for each type [5].

Chapter 2

Design

The goal is to create a C-like language which then compiles down to GPIR code. The language should be as C-like as possible ideally it should be as much of a complete subset of C++ as possible.

C++ is a statically typed, imperative language which is sequentially evaluated by default. GPIR is a dynamically typed functional language which is evaluated in parallel by default. These two languages use two completely different paradigms, so the language design has to ensure that GPC is like C++ and can be compiled to GPIR without too much trouble.

2.1 GPC Language Design Decisions

2.1.1 Parallel Evaluation

GPIR is parallel evaluated by default, with a ”seq” function which evaluates the given arguments in sequential order. Mapping GPC code into GPIR code which can be evaluated in parallel should be possible, and allows GPC to take advantage of parallel evaluation by default. GPC should also have a method of sequentially evaluation of multiple statements.

2.1.2 Serial Semantics

Having serial semantics is helpful to the programmer as it makes it easy to reason about the behaviour of parallel code. This is due to the ability to run the entire parallel code in a single thread as if it were sequential code.

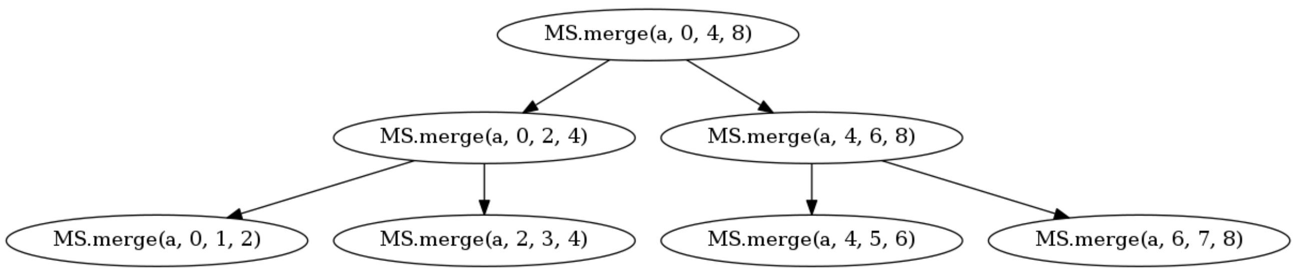
2.1.3 Purely Functional

In the GPRM jumping to labels is an expensive operation, to avoid this function calls will need to be inlined, and loops unrolled as much as possible. The execution path will also need to be known during compile time to achieve this.

Another one of the major benefits of being able to compute the execution path at runtime is that recursive functions can be more efficiently parralelised by the compiler. We'll use a Mergesort example to illustrate this.

```

//Use Sort class from Kernel
GPRM::Kernel::Sort MS;

int size = 8

void merge_sort(int *a, int low, int high) {
    if (low + 1 < high) {
        mid = (low + high) / 2;

        seq {
            par {
                merge_sort(a, low, mid);
                merge_sort(a, mid, high);
            }
            MS.merge(a, low, mid, high);
        }

)

void GPC_merge_sort(int *a) {
    merge_sort(a, 0, size);
}
```

The merge kernel calls be be reduced down to a tree:



From the bottom up, the merge operations in the row can be performed in parallel, knowing the execution path at compile time allows for the compiler to map these to seperate threads.

However if the language is Turing-complete deciding the execution path of a given program is essentially the Halting Problem. It has been proven that the solution to this problem is undecidable[11]

To avoid this problem, the language needs to be restricted to a point where it can still be useful and the execution path decided at compile time.

If the language is purely functional with no side effects then this avoids the problem, however a program with no side effects is useless. There is one major feature that must ne in the language that invokes side effects which is method calls for kernel objects.

Enforcing that anything returned from a kernel object method call's type then has to be in the GPRM::Kernel namespace, and anything passed into a kernel object method call must be in the GPRM::Kernel namespace. This allows passing impure values between kernel methods but restricts them being used for conditional statements.

Essentially we still have side effects but they're restricted to only being in Kernel space and not in the GPRM. The execution of GPC code is then purely functional and the execution path of code can be determined at compile time without oo much difficulty.

2.2 GPC Language Features

This section explains the features of the language with respect to the design decisions in section **??**.

2.2.1 Syntax

The syntax aims to be as close to C/C++ as possible. Statements must end with a semicolon. All variables must be statically typed. Blocks are surrounded in braces. Case sensitivity will be enforced.

2.2.2 New Operators

Two new operators not currently present in C++ "seq" and "par" are introduced. These are placed before a block of statements to determine whether each individual statement within the block are to be evaluated in sequential or parallel. By default a block of statements are evaluated in parallel, but the par keyword makes it more explicit especially if there's lots of nested seq/par blocks.

2.2.3 Objects

Objects can be declared in the top level scope, and must be in the GPRM::Kernel namespace. For example declaring a test object from a class called Test in the GPRM Kernel namespace:

```
GPRM::Kernel::Test test;
```

Standard C++ method calling syntax can be applied to objects.

```
test.m1();
```

2.2.4 Compatibility with C++

Making the language completely compatible with C++ in that all GPC code can be compiled with a C++11 compatible compiler allows programers that are familiar to C++ write GPC code with little difficulty. As long as they are aware of the restrictions that GPC has compared to C++. However the new keywords "seq" and "par" are not in the language.

Currently preprocessor directives aren't supported in GPC so the compiler can just ignore lines starting with "#", and define "seq" and "par" as no-ops. For example this snippet of code compiles with both the GPC compiler and C++ compilers:

```
#include "GPRM/Kernel/Test.h"
#define seq
#define par
```

```

GPRM::Kernel::Test test;

int entry_fn() {
    seq {
        int a = test.m1();
        int b = test.m2(a);
        par {
            test.m3(b);
            test.m3(b + 1);
        }
        return 0;
    }
}
```

When this code is compiled with a C++ compiler it will removed the seq and par keywords and what will be left is plain blocks. Essentially this should generate a serial version of the program which should generate the exact same results as the version compiled by GPC and run on the GPRM. This supports the design decision of serial semantics.

This is useful for implementing the compiler as it allows for testing that the code generated by GPC is generating the correct results by comparing it to the C++ version. It also allows for easier porting of programs already written in C++ to GPC. Programers can use tools already made for C++ to debug their serial code before attempting to run it in parallel on the GPRM.

2.2.5 Types

C++ types such as string, char, bool, int, and double are included. Pointers, and ”multilevel” pointer types are included (e.g. int**, char*). However pointers are restricted in that you cannot take an address of any variable, adding and subtracting integers from pointers is allowed. Usually pointers are passed into the GPRM from the C++ caller to represent an Array.

return values from kernel method calls are implicitly placed in the GPRM::Kernel namespace.

For example:

```
int x = obj.method(5);
```

This is implicitly cast to:

```
GPRM::Kernel::int x = obj.method(5);
```

If a binary expression involved a Kernel value and a ”pure” value then the ”pure” value is implicitly upcast before the operation takes place. For example:

```
bool y = obj.m1(10) == 5;
```

is implicitly cast to:

```
GPRM::Kernel::bool y = ob.m1(10) == 5;
```

This feature stops impurity in the GPC code execution, as the type system should be able to stop Kernel types being used in conditional statements.

For example this is not allowed:

```

seq {
    int x = obj.method(5);
    if (x == 10) {
        //Do Stuff
    }
}
```

If statements must take a ”pure” boolean type as its conditional. x == 10 is a GPRM::Kernel::bool type, so this raises a type error.

2.2.6 Operations

Most basic binary arithmetic operations are included i.e. (+, -, *, /, %, ==, !=, &&, || , <<, >>, &, |, ^) as well as unary operations (−, ˜, !).

(+=, ++, –, -=) are not included, due to the single assignment rule. Although an exception is made for the ”afterthought” of the for loop construct, in which the integer loop variable can be incremented with += or decremented by -=.

2.2.7 Functions

Function syntax is exactly the same as C.

2.2.8 Single assignment

Variables in GPC can only be assigned once per scope.

for example the following isn't allowed:

```
int i = 0;
int i = i + 1;
```

Since ”i” is already in scope, it cannot be redefined in the same scope.

The following code is allowed:

```
int i = 0;
{
   int i = i;
}
```

Since ”i” inside the block is declared in a new scope.

Also variables must be assigned when they are declared, for example the following is not allowed:

```
int x;
x = 5;
```

2.2.9 For-Loops

For loops have the same syntax as C/C++ for loops with some restrictions:

- There is a single loop variable which must be declared inside the loop, and must be an integer.
- The conditional expression must result in a ”pure” boolean type, a *GPRM::Kernel::bool* type is not allowed.
- The ”afterthought” of the for loop must consist of the loop variable with either the ”+=” operator or ”-=” operator with a ”pure” integer value on the right hand side. This is the only place these binary operations are allowed to occur in a GPC program.

These restrictions and properties are in place to keep the functional ”purity” of GPC code, allowing complete compatibility with the C++ language, and keeps the property of GPC programs having serial semantics. At compile time the loop is fully unrolled, these restrictions also allow for detection at compile time whether or not the loop is infinite.

It’s worth noting that these restrictions are similar to the restrictions on the *cilk_for* loop. The *cilk_for* loop must declare a single initial loop variable, the conditional expression must compare the loop variable with a constant ”termination expression”, and the ”afterthought” must either increment or decrement the loop variable by some amount. [3]

An example of a for loop is as follows:

```
for(int i = 0; i < 5; i+=1) {
    obj.m1(i);
}
```

During compilation this loop is fully unrolled and is equivalent to the following:

```
par {
    obj.m1(0);
    obj.m1(1);
    obj.m1(2);
    obj.m1(3);
    obj.m1(4);
}
```

2.2.10 Top Level Statement Restrictions

Top Level statements are restricted to being either object declarations, function definitions, or constant variable assignments. This is partly to be like C++ and also to remove any ambiguity on how top level statements are evaluated.

2.2.11 Entry Function

Since the GPIR function is called by C++ code, it's not preferrable to name the function ”main”. Also the C++ code may be calling more than one GPIR function during its lifetime so a static name is also not preferable. The GPIR code entry function has the same name as the GPC source file it is in. For example the entry function for ”test.gpc” would be called ”test”. This method of determining an entry function is not ideal, but is easy to change in the future if needed.

Chapter 3

Implementation

3.1 Implementation Language Choice

The compiler will be implemented using the Haskell programming language.

During complication the need to traverse trees usually occurs quite often. Functional languages like Haskellare suited to this task due to features such as pattern matching and tail-end recursive optimisation which make traversing trees efficient and simple to implement/

The Glasgow Haskell Compiler is available for most platforms, most importantly for Windows, OSX and Linux on x86 architectures. This allows the compiler to be portable across these platforms provided the libraries used to build the compiler are portable.

Haskell has algebaric datatypes, these makes Abstract Syntax Trees simpler to implement. For example the AST for a very simple expression can be representedas follows:

```

    data Expression =
          Add Expression Expression
        | Negate Expression
        | Const Integer
```

The equivalent in an Imperative/OOP language (in this case Java) would be the following:

```

abstract Class Expression {}

class Add extends Expression {
    public Expression left;
    public Expression right;
    public Add(Expression l, Expression r) {
        left = l;
        right = r;
    }
}

class Negate extends Expression {
    public Expression expr;
```

```
    public Negate(Expression e) {
        expr = e;
    }
}

class Const extends Expression {
    public int value;
    public Const(int v) {
        value = v;
    }
}
```

The Haskell version is much clearer on the structure of the tree, and takes much less code to implement.

It is also to easily extend Haskell datatypes to include custom annotations. For example storing the source code position for parts of expressions may be useful for reporting errors to the user.

Due to the pure function nature of Haskell, parts of complilation can easily be performed in parallel. For example during type checking, each GPC function can be type checked in parallel and this can even be subdivided further into blocks within functions. For this project speed of compilation is not a major concern, but in the future if compliation ever needs to be faster then this option is always available.

Haskell also has powerful libraries for parsing source code such as Parsec which is a parser combinator library. Parsec allows Combinator parsers to be written in the Haskell language itself avoiding the complexity of integration of different tools and languages[13].

3.2 Tools and Testing

3.2.1 Cabal

Cabal (Common Architecture for Building Applications and Libraries) is a system for building and packaging Haskell libraries and programs.[10]. This system can manage the project library dependencies, and automatically download and install missing dependencies. It can also build amd install the compiler on the system, and run unit tests.

3.2.2 Testing

For Unit testing the HUnit library will be used, this can be integrated with Cabal to easily run all the required unit tests.

One form of testing used for this project is testing each individual component (e.g. The Parser or the Type-Checker). Each component in the compiler can easily be uncoupled from one another due to the linear nature of compliation.

Another form of testing is writing GPC source files which should compile, and GPC files which should fail compilation at a certain stage. The compiler is then invoked during testing on all of these files to check whether all the source files which should compile do infact compile with no errors, and all the source files which should raise an error do not compile.

An upside to this method is that testing this way is flexible in that they aren't coupled with the implementation internals of the compiler. The only way that these test would need to be changed was if the design of the language itself would need to be changed.

The downsides to this method is that without manually checking the errors raised by the tests that should fail the source may generate an error which is unrelated to the error that is being tested. This is why some testing of each internal component is done alongside this method.

3.2.3 Code Coverage

The Haskell HPC (Haskell Program Coverage) library allows for recording code coverage over different modules during testing. This can be integrated with cabal unit testing to automatically generate these results. The usefulness of this allows for checking which sections of code still need to be tested and assists in writing further unit tests.

3.3 Compiler Structure

Compilation is split up into multiple stages or ”passes”, it is possible to compile in one pass but seperating each specific section of compilation allows modularity and decoupling.

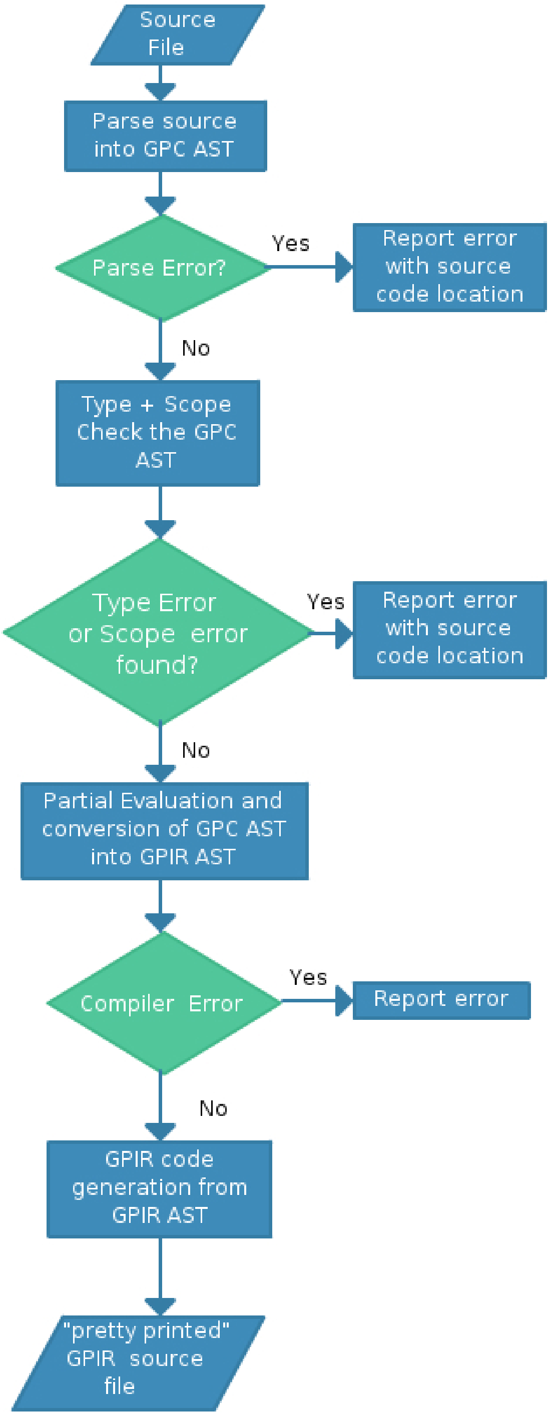


Figure 3.1: Flowchart illustrating the stages of the GPC compiler.

3.4 Parser/Lexer

The Parsec library combines Parsing and Lexing into one stage.

Given the GPC source file the intention is to parse the file into an AST to hopefully eventually be transformed into GPIR source code. It's also useful to store the original source position to provide error information during a further compilation stage, to achieve this the source position info is read from Parsec into an Annotated AST as the tree is being built up.

Parsec does most of the work during this stage, including providing error messages for ”expected” values to be found in source positions and source position information. Most of the work implementing this stage is building the parser combinator functions and composing them together to be able to build the AST.

3.5 Type and Scope checking

During type checking the types of identifiers from the current scope being used in expressions need to be known. Since the scope needs to be kept track of while type checking it makes reasonable sense to check for scope errors in the same stage.

The goal of this stage is to ensure that the static typing of the source file is enforced (e.g. attempting to assign a bool value to a variable of type int should not happen) and, prevent ”logic” errors at compile time (e.g. adding 2 bool values together). Scope checking is also important as identifiers being used within the program need to be binded to an expression of some sort, and the ”single assignment” rule in the GPC language needs to be enforced.

There are two seperate ”types” of scope in a GPC program. The first is the top level scope and the other is at function level scope. Any scope ”further” down from function level scope is itself a function level scope. During this stage the top level scope will be type and scope checked. Afterwards each individual function will be type and scope checked.

For checking over top level statements the following Haskell record is used to keep track of identifier types, objects, and functions encountered:

```
type VarTable = M.Map (Ident SrcPos) (Type SrcPos)
type FunTable = M.Map (Ident SrcPos) (Type SrcPos, [Type SrcPos])
type ObjectTable = M.Map (Ident SrcPos) (Objects SrcPos)

data MainBlock = MainBlock {
    _tlFuncDefs      :: FunTable, -- ^ Function Definitions
    _tlVarTypes      :: VarTable,  -- ^ Top Level Constant variable types
    _objects         :: ObjectTable  -- ^ Table of current Kernel objects declared
} deriving (Show)
```

It needs to be checked that no duplicate functions exist and no duplicate top level variables exist, Once all top level statements have been checked, all functions in scope stored, all top level variables stored, and all variables stored. Each individual function can be checked.

A slightly different stucture is needed to type check functions.

At anytime it is needed to be known what variables that are currently in scope, their types, what functions are avilable to call, their argument types, and return types. Also the objects which are available to call methods on, and the current function the type checker is in at any point.

These values can be stored using the following Haskell record:

```

data CodeBlock = CodeBlock {
    _currentFun :: Ident SrcPos, -- ^ Name of Function block is in
    _funcDefs   :: FunTable, -- ^ Function names and return/argument types
```

```
    _prevVars    :: VarTable, -- ^ Identifiers visible in current scope with types
    _curVars     :: VarTable  -- ^ Identifiers declared in current scope
} deriving (Show)
```

When a function is being type check a new CodeBlock instance needs to be created from a MainBlock instance, as some of the top level information is needed. The details of the function that is being entered is stored in ”_currentFun”, details of all available functions stored in ”_funcDefs”, and store all top level variable in ”_prevVars”. ”_curVars” is left as an empty map as the type checking of the function hasn’t begun yet. Top level objects are also stored in ”_curVars” as an ”object” variable type.

Whenever a new scope is encountered by either calling a function, or entering a seq/par block; A new CodeBlock structure is created using the current structure. The new ”_curVars” is set to the empty map as no variables have been encountered yet, the ”_funcDefs” are copied as all functions are on the top level so they don’t change. The ”_currentFun” is copied if entering a block, otherwise if entering a function the function name and source position are copied.

The value of the new ”_prevVars” is a little more complicated to work out. Any key, value pairs in the current ”_curVars” are stored plus any key, value pairs in the current ”_prevVars” in which the key isn’t present in the current set of ”_curVars” keys. This is because the identifiers in ”_curVars” scope are visible over the identifiers with the same name in ”_prevScope”. Haskell’s union operation on maps discards the key, value pairs in the second map for keys that are present in the first map, so this is trivial to implement.

When type checking a function, every statement in the function is type checked as well as the type of the return statement. For every statement, every expression within the statement is type checked. This is implemented by traversing the Statement AST and checking the scopes of identifiers as well as expected types against actual types.

Objects which are declared are not checked to see if they actually exist. Neither are method calls which means the argument types and return types when calling them cannot be determined at compile time. This is due to the fact Objects are written in C++, and the C++ class would need to be checked for methods and types. This is already implemented in the GPRM, so an error will occur further down to compile ”chain” or during runtime.

If a type or scope error is encountered, an error message determining the type of error, and the source position of the error is returned. Otherwise an empty tuple is returned. When type and scope checking the original AST doesn’t need to be modified, only verified that it follows the type and scope rules of the language.

3.6 Interpreting

The goal of this stage is to run through the execution path the GPIR code will take from the entry function, and partially evaluate the code as much as possible while generating the GPIR AST.

Just before Evaluation the AST is transformed slightly into a similar AST with a couple of differences. One is that annotations are not present (since source position information is not needed anymore), and type information is stripped (since the program has been proven to not have any type errors), also objects in the top level scope are stripped. (Since scope checking proved that all methods are called on objects that exist, and whenever a method is called, the name of the object is part of the call).

Initially the values of all top level assignment statements need to be stored as constants before executing, as well as every function.

While interpreting a state is needed to determine actions taken during certain sections, as well as to evaluate expressions. The following Haskell record is used:

```haskell

type ConstVarTable = M.Map Ident Literal
type FunTable = M.Map Ident ([Ident], BlockStmt)
type VarRegTable = M.Map Ident Integer

-- ^ Current State of the Block we are currently in
data CodeGen = CodeGen {
   _funTable    :: FunTable,  -- ^ Store symbol tree for functions
   _constTable  :: ConstVarTable, -- ^ Store constants in scope
   _varId       :: Integer, -- ^ Current variable id for mapping to registers
   _varRegTable :: VarRegTable, -- ^ maps variable identifier
   _threadCount :: Integer, -- ^ Current thread number to map
   _maxThreads  :: Integer,  -- ^ Maximum number of threads
   _seqBlock    :: Bool, -- ^ Whether or not current block is sequential
   _isReturning :: Bool -- ^ Whether the state of the current block is in a return
}
```

3.6.1 Registers

Values which can be fully evaluated at compile time can be subsituted into the GPIR code with their literal value when they are used.

For example:

```
seq {
    int x = 4 * 5;
    obj.m1();
    obj.m2(x + 3);
}
```

```
seq (
     '(obj.m1[0])
     '(obj.m2[0] '23)
)
```

The value of ”x” is able to be calculated to be 20, so whenever ”x” is used in the scope it can simply be replaced with the literal ”20”.

However, some variables will not be able to be fully evaluated at compile time. So results will have to be written to GPIR registers.

For example:

```
seq {
    int x = obj.m1();
    obj.m2(x);
}
```

```
seq (
    '(register.write[0] '1 'obj.m1[0])
    '(obj.m2[0] (register.read[0] '1))
)
```

The value of x can't be known at compile time so it is written into register 1, and then read from the same register when it is needed.

During interpreting, there needs to be no conflict between registers (e.g. if a value is stored in register 1 that will need to be used later, register 1 cannot be written to until that value is out of scope). There's no hard limit on registers so a simple method of just incrementing the register count every time a value needs to be stored is implemented, although this may possibly cause a lot of unecessary memory usage.

"_varRegTable" is used to store the mappings of variable names to register numbers. Conflicts between variables with the same name in different scopes is not a problem, as the register table in the inner block is thrown away once the scope is left.

3.6.2 Sequential And Parallel Block Differences

A sequential block translates to something different in GPIR than a parallel block.

For example, these two snippets compile to different GPIR code despite the blocks containing the same code:

GPC Parallel

```
par {
    obj.m1();
    obj.m2();
}
```

GPC Sequential

```
seq {
    obj.m1();
    obj.m2();
}
```

GPIR Parallel

```
(par obj.m1[0] obj.m2[1])
```

GPIR Sequential

```
(seq `obj.m1[0] `obj.m2[1])
```

Sequential statements need to be "quoted", which is why the record contains the "_seqBlock" flag, so the interpreter can generate the correct GPIR code.

3.6.3 Thread Mapping

Each task in the GPRM must be mapped to a thread. There is a function in the Interpreter called "getThread" which calculates what the next thread number should be to map the task to based on the current block state.

Currently this function uses a simple incremental scheme and rolls around modulo style after the max number of threads have been specified. The compiler attempts to work out the maximum number of cores on the machine if a thread number isn't given, and uses this number to determine the max number of threads.

If a different scheme is needed the contents of the function will need to be changed, a possible future improvement may be to pass a function to the interpreter when passing it the AST to determine the counting scheme. This would allow for multiple different schemes to be chosen and possibly tuned depending on the type of application.

3.6.4 Branching

When encountering an if statement the interpreter must be able to evaluate the condition and generate a true or false value. The interpreter will only evaluate the statement within the if statement if the condition is true. In the case of an if-else statement either the first statement will be evaluated or the else statement depending on the value of the condition.

3.6.5 Returning From Functions

Performing a return is a bit tricky.

Some pre-evaluation is needed on blocks of statements when entering a new block. if a return statement is in the current block of statements (not counting sub-blocks) then any statements after that are ”removed”, and when a return statement is found, it is evaluated as the return expression. What this means is that once the return is met, the interpreter ”returns” back to the code which executed the previous scope.

There are also two different scenarios when performing a return.

If the interpreter is in a function at the top level of the function: Returning is simple in this case, all that is needed to be done is going to the previous scope.

If the interpreter is nested within one or more blocks in a function: Returning in this case involves going up multiple times until the interpreter is out of the scope of the current function.

To deal with these situations a ”boilerplate” function is used. This function is called whenever an inner-scope needs to be interpted. It takes a boolean value which determines if the current block scope is at the ”top” of a function, gets the current state and then interprets the block. Once the interpreter returns from interpreting the inner block, the ”_isReturning” flag is checked on the returned state. If it is set, and the scope of the current block is at the top of a function, then the ”_isReturning” flag is then set for the current block.

When the ”_isReturning” flag is set for a block, the evaluator will not evaluate any further statements in the further block. This allows for propogation of a ”return” up the block states until the current function is exited.

3.6.6 Inlining Functions

3.6.7 For Loop Unrolling

3.6.8 Partial Evaluation

3.7 GPIR Code Generation

Chapter 4

Conclusion

Chapter 5

Future Work

Future Features that could be added:

5.1 Configuration file generation

yml config generation. These files are needed by the GPRM to determine aliases in GPIR code, libraries used, number of threads/nodes. There is enough information at compilation time to generate these, and they are currently generated manually.

5.2 Language Features

Other C++ language features, C++11 lambdas may be useful, while-loops, and possibly some STL support e.g. ability to use STD vector instead of pointers.

5.3 Compiler Optimisations

There are a few optimisations that can be performed by the compiler to reduce the work needed to be performed by the GPRM at runtime.

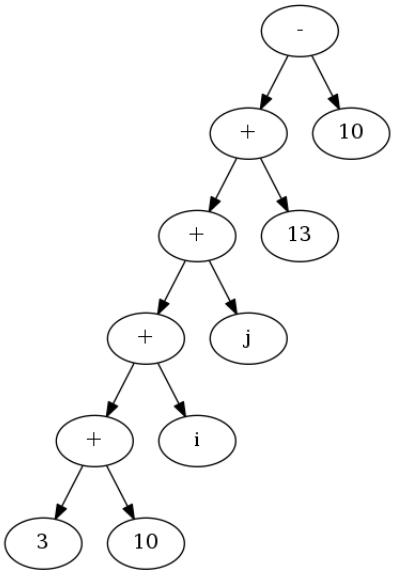
5.3.1 Binary Expression Reduction

Given the following GPC code:

```

GPRM::Kernel::Test test;

void binaryOp() {
    seq {
        int i = test.m1();
        int j = test.m2();
```

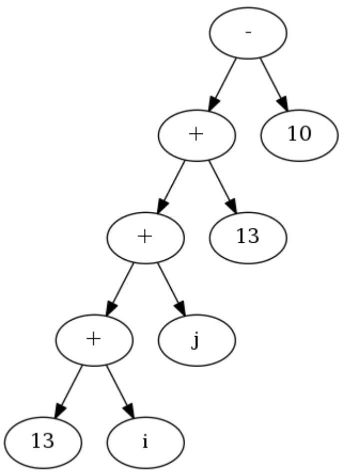
```
8            test.m2(3 + 10 + i + j + 13 - 10);
9        }
10   }
```

In line 8, the expression within the m2 method call results in part of the Abstract Syntax Tree which is represented as a Binary Expression Tree when parsed. Binary operators are contained in the inner nodes with literal values and variables at the leaf nodes.



During the evaluation stage of compilation the tree is evaluated through postorder traversal. Once two leaf nodes are found the binary operation in the parent node is attempted to be applied to the two leaf nodes. If both leaf nodes are values which can be calculated at compile time the expression is evaluated, the two leaf nodes are removed and the parent node is replaced with the calculated value. If one or more leaf nodes are values which can only be known at runtime then the expression is not evaluated and the tree doesn't change.

The tree above once evaluated transforms into the following tree:



We can see this reduction in the generated GPIR code:

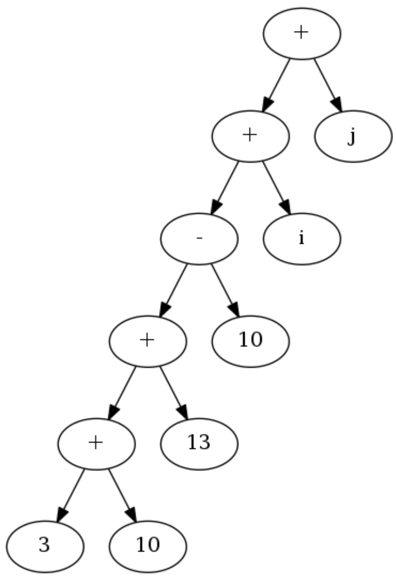
```
;binaryOp.yml

(seq
    '(reg.write[0] '1 test.m1[0])
    '(reg.write[0] '2 test.m2[0])
    '(test.m2[0]
        (-[0]
            (+[0]
                (+[0]
                    (+[0] '13 (reg.read[0] '1))
                    (reg.read[0] '2))
                '13)
            '10)
        )
    )
```

The downsides of this method is that once a value that can't be worked out at compile time is met, it is not possible to evaluate expressions any value further up the tree. In the given example it is clear that it is possible to evaluate the expression further.

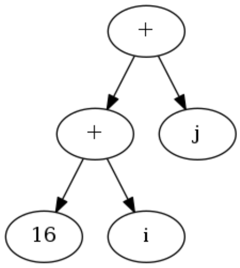
One way to improve the evaluator would be to transform the tree before evaluating it. Using the axioms of the binary operations in the tree (e.g. addition being associative) and the type of values in the leaf nodes,. it should be possible to rearrange the nodes in the tree to create a new tree which represents an expression equivalent to the expression represented by the starting tree.

The specific details of how the tree transformations work and implementation into the compiler is left as future work.

In this example, one "optimal" transformation of the tree would result in the following tree:



The evaluator would then reduce this tree down to the following tree:



This generates much simpiler GPIR code:

```
;binaryOp.yml

(seq
    '(reg.write[0] '1 test.m1[0])
    '(reg.write[0] '2 test.m2[0])
    '(test.m2[0]
        (+[0] (+[0] '16 (reg.read[0] '1)) (reg.read[0] '2)))
```

5.3.2 Pure Function Reduction

When a function is pure (i.e. it contains no method calls on kernel objects, and all arguments given to it when it is called can be evaluated at compile time) the function call should be able to be evaluated down to a single constant value.

The following GPC code is used as an example:

```

GPRM::Kernel::Test test;

// Pure Function
int f() {
    int a = 5;
    int b = 6;
    int c = 10;
    int d = 22;
    return ((a * b) / c) & d;
}

void pureFun() {
    seq {
        int a = f();
        test.m1(a);
    }
}
```

The function ”f” does not call any methods on kernel objects, and has no arguments. Every time ”f” is called it returns the integer value ”2”. Therefore wherever f is called can be replaced with the integer value ”2”. However the compiler generates the following GPIR code for this example:

```
;pureFun.yml
```

```
(seq
    '(reg.write[0] '1 '2)
    '(test.m1[0]
        (reg.read[0] '1)))
```

When "f" is called its value is stored in a register. This is currently how function calls store their return value when evaluated. An improvement to function evaluation to support reduction of "pure functions" would result in generating this simpler GPIR code:

```
;pureFun.yml

(seq
    '(test.m1[0] 2)
)
```

5.3.3 Register Tracking

As explained in section insert section, the register tracking is non existent and every time a new value needs to be stored the register counter is just updated. After a while this could possibly use a lot of memory. A more efficient method would be to track registers being used, once a variable goes out of scope then that register gets "freed".

The "free list" can be implemented in Haskell as a simple linked list containing the numbers of registers that are free. Every time a register is "freed" its number gets added to the tail of the queue. When attempting to assign a variable a register the list is checked to see if it has any free registers on it, if it is then the head of the queue is taken as the register number. Otherwise the total register counter is incremented and the new value is used as the register number.

In a scope keep track of all the register numbers allocated, once out of scope then add every one of those numbers to the "free list", as the values in those register will not need to be used again. This method of register tracking and freeing will use far less total registers than using a new register for each value.

Appendices

Bibliography

[1] Cilk Plus Array Notation. https://www.cilkplus.org/tutorial-array-notation. Accessed: 2015-03-15.

[2] Cilk Plus FAQ. https://www.cilkplus.org/faq. Accessed: 2015-03-15.

[3] Cilk Plus *cilkfor* documentation. https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-ABF330B0-FEDA-43CD-9393-48CD6A43063C.htm. Accessed: 2015-03-15.

[4] Cilk Plus Tutorial. https://www.cilkplus.org/cilk-plus-tutorial/. Accessed: 2015-03-15.

[5] Intel Thread Building Blocks Benefits. https://software.intel.com/en-us/node/506043. Accessed: 2015-03-15.

[6] Intel Thread Building Blocks Scheduling Algorithm. https://www.threadingbuildingblocks.org/docs/help/reference/task_scheduler/scheduling_algorithm.htm. Accessed: 2015-03-15.

[7] Intel Thread Building Blocks Website. https://www.threadingbuildingblocks.org. Accessed: 2015-03-15.

[8] OpenMP. http://openmp.org/wp/. Accessed: 2015-03-15.

[9] openMP Application Program Interface 3.0. http://www.openmp.org/mp-documents/spec30.pdf. Accessed: 2015-03-15.

[10] The Haskell Cabal. https://www.haskell.org/cabal/. Accessed: 2015-03-11.

[11] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem, 1936.

[12] Alexy Kolosov, Evgeniy Ryzhkov, and Andrey Karpov. 32 OpenMP traps for C++ developers. https://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers. Accessed: 2015-03-15.

[13] Daam Leijen. Parsec: Direct Style Monadic Parser Combinators For The Real World, 2001.

[14] Ashkan Tousimojarad and Wim Vanderbauwhede. The Glasgow Parallel Reduction Machine: Programming Shared-memory Many-core Systems using Parallel Task Composition, 2013.