Automatic parallelization with flow programming

Christian G. Kalhauge

Kongens Lyngby 2012
IMM-BSc-2012-17

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-BSc-2012-17

Preface

This thesis was prepared at the department of Informatics and Mathematical Modeling at the Technical University of Denmark in fulfillment of the requirements for acquiring an BSc. in Informatics.

The thesis deals with the automatic parallelization of a flow based language.

Lyngby, 28-June-2012

Christian G. Kalhauge

Acknowledgements

I would like to thank my supervisor Christian W. Probst to allow me to work on this big project.

I would also like to thank Luke A. D. Hutchison, MIT CSAIL, for the friendly advices and a giving dialog about flow based languages.

A thank is needed to Tobias Bertelsen, Bjarke Wanstrup, and my dad, for correcting spelling and other errors in the report and helping me making it readable in general.

I like to thank my fellow students for not insisting to me drink with me every night, and my vector groups for doing it any way.

Last I like to thank my family for the food and shelter, doing the final process, and my dog, Kvast, for company.

Contents

Chapter 1

Introduction

We are currently moving in a quickly against the biggest problem in the computer world since the Y2K problem. We are constantly putting more and more processors in a computer but we have not changed the way we program.

The current programming languages makes it hard to program as they gives the programmer the felling that there is a linear workflow. This means that it is hard to program parallel programs, and in the future it will only be the best programers that can use the entire power of the computer.

This is why we need a new way to program. **Flow** is an intermediate language which purpose is to enable automatic parallelization of code, and guarantees that the output program does not contain deadlocks or race conditions.

The project is a excavation of the possibilities of flow based languages. I will touch subjects as automatic parallelization, code testing, memory handling, and static/dynamic CPU scheduling on multiprocessing systems, and how to parse and compile code.

Chapter 2

Analysis and Design

This chapter contains the analysis and design consideration made throughout the project. First I will analyse what the language needs to be able to do. Then I will go through the different stages of the translation of `FlowCode` into **Flow** and how to compile **Flow** into parallel code.

2.1 Do we need a new Programing Language?

2.1.1 Is Parallel Programing Important?

During the last decade we have reached a clock-rate on CPUs, that seams to be the fastest we can reach, with the current technology. The problem lies in that changing the voltage in electrical systems builds up heat. So if we build our CPUs with a higher frequency, they would burn. The solution is to build multiple processors onto one chip, and making them run at a lower frequency, this both saves power and gives lower temperatures on the cores.

We can therefore expect to see an increase in the number of processors in our computers. The problem of the increase in processors is that our current programing languages do not accommodate for the different nature of the new platform. It is not a big problem now, as the number of processors in a home computer is around eighth cores. This means that we do not need to make the programs parallel, as we often run more than one application at one time, which are directly parallelized by the operation system. But this is about to change, as we might soon have CPUs with more than a thousand cores spread around the computer optimized for doing different tasks [JJK$^+$11].

In the future we need programs that allows for fine grained parallelism by running them on more processors. The biggest problem now, is that our current code is no longer processed in a strictly ordered manner, which means that our normal approaches and algorithms become unreliant, and parallelism may even hurt their performance.

Our current imperative languages does handles parallelism very badly as they allow for a lot of parallel errors like race conditions and deadlocks. If we keep on using these languages we will end up with a lot of buggy code. The worst problem with race conditions is that they are not necessarily found at tests, and if they occur they can be extremely hard to reproduce.

Of cause programs written as parallel is at least as good as serial code translated into something that is parallel, but to rewrite old code is often a costly affair. So if the programing language should have a impact on the world it requires that we can translate old code into the new without changing it too much.

We need a new solution than can make another layer between the programer and the hardware to protect the programmers, and enable them work with multiple cores without knowing it, or at least without they need to think about it.

2.1.2 Short description of important terminology

Data-parallelism is when the parallelism arises from the use of different data. If we parallelize a for-loop, we perform the same operation, but as we do it on different data it can be done in parallel.

Task-parallelism is when the parallelism comes from running different tasks on the same data. An example would be to count the number of names in the phone book that starts with "A" while summing all the phone numbers.

2.1.3 What is out there right now?

Before I started the project I started a analysis, of the current market, and what was already out there. So I have look at some of the leading parallel programing languages:

C It is possible to code parallel in C, but it is hard without extensions like OpenMP. The standard way is to use parallel tools like pthreads or the Windows equivalent. The risks of making errors is high, as all checking is done by the programmer, but on the other hand there is a lot of power to do what you want. The problems behind OpenMP, and other parallelization extensions to C is that as they want to keep the power with the programer. This means that the parts of the code that should be run parallel is often manually chosen, and these parts often only focusses on the for-loop, or data-parallelism. C is by default extremely hard to automatically parallelize as it uses pointers, which makes it hard for the compiler to predict the idea of the program.

Java Java is like C, but has a build in functionality for parallel programing. The parallel part focuses on the monitor-pattern which isn't build for speed but for security. That makes sense giving that Java is also used for server programing-language. There is also a lot of different classes that allow for some sort of data-parallelism, but this is also manually chosen.

X10 is a solid parallel programing language, it looks like Java but there is added more functionality to improve the parallel experience. It is possible to do both data and task parallelism, but there is still no automatically parallelization.

StreamIt is a stream based language, almost in the degree of a turing machine. Parallelism is reached through reading and writing to streams and let different functions work on them. This language uses implicit parallelization but is not build as a high performance language but a language to work on streamed data, like when decoding a video stream.

ZPL where the programing language that should substitute C, but was last updated in 2004. ZPL is a data parallelism based programing language which is based on automatic parallelization of arrays.

OpenCL is set of instructions, developed for other languages, that allows the programer to access all the processing units on the computer. This language is exciting as it allows for much better usage of the resources of the computer. The base problem behind OpenCL is that it is difficult to use, and all the parallelism used in the program is manually chosen by the programmer.

Fortran A lot of work has been done on automatically parallelize fortran, but again the main focus of this automatic parallelization has been focusing on loops.

Map-Reduce is a programing model suggested by google. It is based on the idea that some parallel problems are easily distributed across multiple servers. This can be done by mapping the problem out to the servers, and afterwards collecting the data. The collection of data is called the reduce step. The base problem with this model is that it is especially designed to work on servers, and that it focuses on data-parallelism.

2.1.4 What does Flow need to be able to do

After analyzing the languages I found that a very few of them in fact tried to support checks against parallel errors. This has for a long time been left to the programmer. I want to take some power from the programmers and instead give them a platform that produces guaranteed thread safe code.

The idea behind **Flow** is that it should support the current software development community by helping it evolve from the use of obsolete serial programing languages to the full-blown parallel programing languages. This should be done without **Flow** becoming a temporary solution.

To summarize, **Flow** needs to be:

Completely parallelizable , thereby ensuring that when we get more cores in the computer, they are used correctly and efficiently. Compilers using **Flow** should be able to produce code without race conditions, deadlocks or other synchronization problems. I.e. **Flow** should be guaranteed thread safe.

Hardware and operation-system independent meaning that **Flow** should only coded once and then it should be able to run fast on any hardware or operation system. This also ensures forward portability as each platform can compile **Flow** to their own code.

Flow is not a programing language , but an intermediate language. The programing language should be separate from **Flow** . **Flow** should be a guarantee, so if a language can translate into **Flow** , then it can use the parallel **Flow** compilers.

2.2 Flow design

This section contains the how the design considerations of **Flow**

To accommodate the requirements we need a very modular structure. There are three steps, translation, optimization, and compilation. At translation we try to translate the language into **Flow** by finding the control flow of the language. It is not always possible to translate an imperative language to **Flow** as the control flow of the language is not known at compile time. Most declarative languages should be directly translatable.

The compile step is the exciting step, as it is here the parallelization happens. It is my hope that when a hardware producer creates a new processing unit, then in the same way as they need to build a new assembly compiler, they should also present a **Flow** compiler. Operating systems should decide how the compilation happens, which hardware compilers to use and how to distribute the problem. This could both be compiled at runtime, as in OpenCL, or completely compiled as in C.

Figure 2.1 shows the intended stages of compiling.

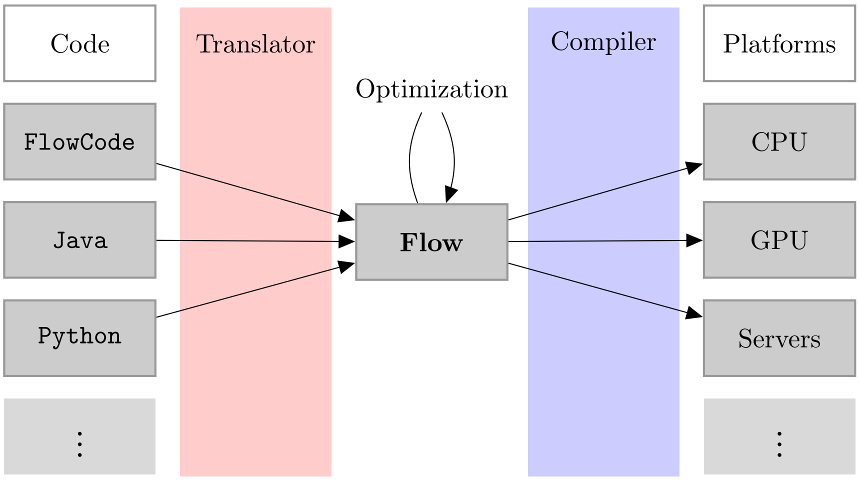


Figure 2.1: The **Flow** compile structure

The Idea is that many syntaxes can be translated to **Flow** , and **Flow** can compile to many platforms. So by hooking up to **Flow** you automatically get the parallelization and the guarantee that your program runs without parallel errors.

The programers should be able to tab into this structure at any time. They should be able to translate fully or use it embedded in an existing programing language. Let us say that we have developed a program in Java, and we want to have a part that uses the full capacity of the computer, then we would translate that part of the code to **Flow** , which in turn could be compiled parallel to the Java Intermediate Language.

It would of cause be better if we could use parallel languages that gave the programmer a feeling of programming in parallel, because the decisions made by the programmer is then reflected directly in the execution, and is not guessed by a compiler. But by allowing all languages to try to translate to **Flow** , it could work as a bridge between the serial coding of today and the parallel coding of tomorrow.

In this project I will make a prototype of **Flow** . To prove the idea I will also create a language `FlowCode` that can be translated to **Flow** and a **Flow** compiler that can compile **Flow** to parallel code.

2.2.1 Explicit dataflow

This sections notion of the directed acyclic graph, and its features is build on work made by Luke Hutchison[Hut11].

But how can we guarantee that a race condition cannot occur? A race condition can occur when the order of the access to a variable is not specified. Figure 2.2 is an example of a typical race condition. As we can see is the access of `var` not controlled by the programmer but by the execution of the processes. The example can, with `var` being 0, produce 3 different results, -3, 0 and 3. And this only gets worse when more variables is in play.

```
Process A                     Process B
reg <- read(var);             reg <- read(var);
reg <- reg + 3;               reg <- reg - 3;
write(var,reg);               write(var,reg);
```

Figure 2.2: A typical race condition situation written in pseudocode

The problem is that when we read from the variable, we don't know which state it is in. Process A could have changed the state of `var`, before process B got a change to read, and visa versa. To prevent this we make an assumption:

Rule 1 *A variable cannot be written to after its instantiation. This makes them immutable and from now on I will call them values.*

Lemma 2.1 *We can efficiently describe any program that follows Rule 1 as a directed acyclic graph, where nodes are the values, and the edges are functions. I call this the data flow graph, as it represents the movement of the data.*

Proof. The graph is directed since functions per definition has a direction, and the graph is without cycles as a function cannot write to a variable after its instantiation.

It is important for the directed acyclic graph, in order to be parallel and guarantee no race condition, that a function does not have a state. If a state is stored in a function it cannot be represented as an edge in the graph.

Rule 2 *A function is stateless and does not depend on previous functions, only the input.*

When the data flow graph is acyclic thens the runtime data flow is known completely by the compiler, that makes the program unambiguous. And when a program is unambiguous then the risk of race conditions and deadlocks is completely removed. Race conditions do not occur, because the order the reads and writes is known, thereby eliminating the race. And the deadlock can not occur because a system only deadlocks if it is in a cyclic lock, which is prevented by the acyclic nature.

Before we continue I will introduce the dependency graph , which is reverse data flow graph, see **Fig. 2.3**. The edges of the dependency graph describes the dependencies of the node. So if some data A depend some other data B, illustrated by a arrow, then B needs to be calculated before A. The dependency graph is also a directed acyclic graph, and as we do not have any cycles we can describe it as a layered graph. We distribute the nodes in the layers, with the rule that all dependencies of a node must be in a layer lower than the node.

Using Rule 2 and the layered dependency graph we can see that all functions producing data to the same layer can be executed in parallel. This can be done as we know that running a function does not alter anything but the data, and that all data is present, when we have reached the layer.

To reach a higher level of parallelism, we could run layers from independent parts of the program separately, or make some sort of lazy interpretation. The point is that we get implicit parallelization of the functions in the same layer.

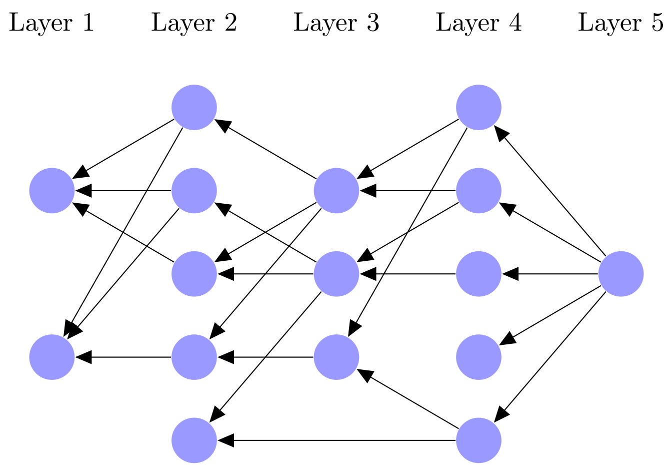


Figure 2.3: A layered dependency graph

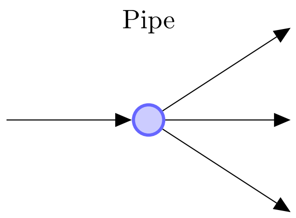
This way of structuring the dependency graph also allows us to make automatic memory allocation. When we reach the layer after of the last reader of the data, then we know that all the functions that need the data has run, and program can therefor safely deallocate the data. We also know exactly when to allocate new memory, without the programmer explicitly stating so.

2.2.2 How it is done in Flow ?

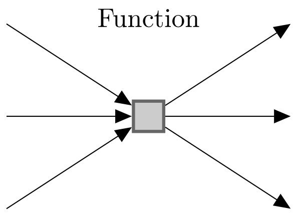
Essentially **Flow** is a layered dependency graph, with some tools added to give the programer control over the parallelization. **Flow** contains of two main components, the pipes and the functions. These are represented as nodes in a graph.

The graphs of **Flow** will be shown as data flow graphs as they are easier to read.

The Pipe is the value holder in the program it transfers the information from one function to an other. A pipe have a in-degree of 1, and a arbitrary out-degree. The node from the end of the in-edge is called the writer, and the nodes from the end of the out-edges are called the readers. The writer and the readers are always functions.



The Function is, as the name suggests, the function of **Flow** . Functions calculate on the data of the pipes they read and writes the result of the calculations to others pipes. A function has arbitrary in- and out-degree, the nodes from the end of the in-edges is called sources and the nodes from the end of the out-nodes is called targets. Targets and Sources are only allowed to be pipes.



This structure is checked at compiler time not to contain cycles, to ensure that it is still a directed acyclic graph.

The functions is a bit different from functions in C-like programing languages, because they can have more return values. This adds an extra level of complexity, but it is needed for the pull function to work, which we will see in **section 2.2.2.1**

A small example about how the data flow of a statment:

$$A = 10 + 2$$

would look like pictured in **Fig. 2.4**. 10 and 2 is given and we then calculate A to 12 by adding them.

Internally is **Flow** build up the same way as most declarative languages evaluates, opposite the data flow, see **Fig. 2.5**. This is actually the dependency graph that I mentioned before. It means that there is a clear evaluation order, which is important when building algorithms, that depends on an incremental build up.

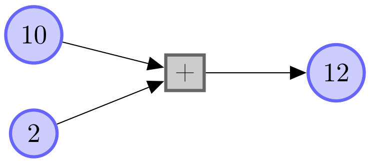


Figure 2.4: Small example of the data flow of $A = 10 + 2$

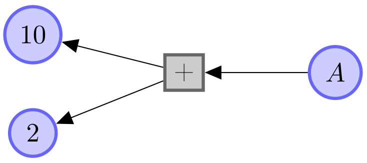


Figure 2.5: The **Flow** dependency graph for the same example

2.2.2.1 Types in Flow

Each pipe does also hold a type. In a production version of **Flow** , a Type should be a transparent object without any information other than its name. The size of the node and its representation should be decided by the compiler, within the design specifications of **Flow** .

Structs A modern intermediate language needs to support some sort of structuring of data, to allow for object oriented programing. This can be a problem to control when using parallel programing, because creating large objects or structs heavily affect the parallelism. We often se programing languages locking entire objects when handled by a thread. This means that a lot of processing time has a risk of getting lost.

To prevent this a `push` and a `pull` function has been added.

The `pull` function, pulls a field from the object, and locks it. The `push` function then pushes the value back into the field, and marks the field as unlocked again. This is never felt at runtime, as the locking and freeing is only done in the compiler to check that a value is never pulled twice without pushing it back in the meantime.

As an example lets look at a Point struct that we want to calculate on. We want the `x` field to be doubled and the `y` field to be halved. If we lock the entire struct we cannot run these calculations in parallel. What we do is to pull `x` and pass the Point over to the `y` calculation function. The `y` calculation function cannot pull the `x` value so we can do the calculations on `x` without fear of a race condition. And on the same time we can calculate on `y` without problems, because locking `x` does not lock `y`.

Signals One of the big problems running parallel code on a computer is that some hardware and software resource access is by nature serial, and that the resource has a state. This means that there is a risk of race conditions. Often these problems are handled by the operating system, but in some cases it needs to be controlled by the programmer.

A good example is when we draw on a screen. If we write to it in a uncontrolled fashion we could risk that some of the background images gets in front if the foreground. In such a case it is important that we can control the access, so that we draw the objects in the right order.

To ensure the correct order of access of the resources, the transparent type `Signal` is introduced. The `Signal` is a **Flow** representation of a resource that isn't directly in the memory system. To make use of the `Signal` the developer make a set of functions that uses the `Signal`. A good example is the print functions that need a Printer signal to print.

Some resources should not be accessed by multiple threads at one time. To prevent that, we can declare these signals as atomic, which means that pipes controlling the signals only can have one reader, except in special cases; see Select 2.2.2.2. When pipe only have one reader then there can not get more of them.

Set, List and Dictionary There is three build in collection types in **Flow** . These are are the `List`, the `Set` and the `Dictionary`. They all have special abilities, that makes them better to what they do in a parallel environment.

Set The `Set` is a unordered collection of data that is especial optimized to get accessed by multiple threads, without giving problems or loosing performance. It is primarily used for collecting calculations in a parallel environment. The set is the most parallelizable of the three types, because the nature of the set empowers the compiler to decide the order of the calculations, when mapping over a set.

List The `List` is an ordered collection of data, but does not allow for random access. It is primarily used to aid problems that need vectors and matrixes, where we can map over the structure.

Dictionary The `Dictionary` allows for random access. It is possible to calculate on the values in a dictionary in parallel. But the `Dictionary` is primarily used to store results while calculating on a list in a parallel environment. A lot of constrains is based on the dictionary, which makes the structure more "serial" than the two other.

2.2.2.2 Special functions

Besides the function and the pipe, I have also introduced some other control flow altering nodes; the select, the map, and the recall node. These nodes are special functions.

Select The select is the logical data flow changing element. It has 3 sources and 1 target. The first input should be a boolean and the next two should be of the same type as the target. If the boolean is true we take the first source and pass it to the target otherwise we parse the second.

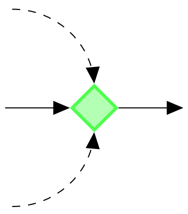


Figure 2.6: The select node

At runtime we can choose not to evaluate the true or false source before we get the boolean, or if can evaluate the true and the false pipe with a smaller priority. It can be a great advantage to, when we have idle processors, that we can start calculating the possible sources before we know whether the boolean is true or false.

Map To loop over an collection is a very serial approach to a problem that is often implicitly parallelizable. **Flow** does contain a notion that describes that all the elements in one or more collections need to be calculated, by a given function. This node is called a `Map` and it runs a function for each element in the collections.

Flow demands that all lists in a `Map` environment can at compile time be guaranteed to have the same number of elements. This ensures that we do not get array out of bounds failiurs, and there is also no problem with deciding the number of iterations needed. In this case it is important to point out that at compile time we do not need to know the exact size of the lists, only that they are of the same size.

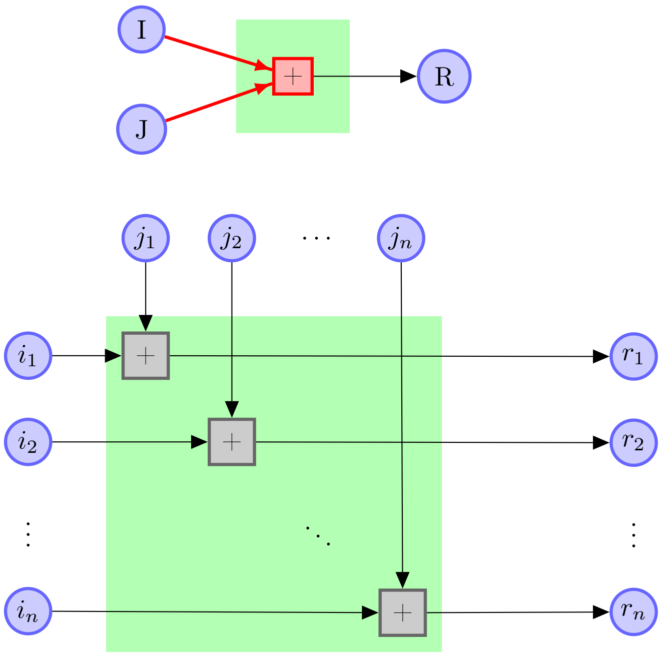


Figure 2.7: Map example for the addition of two lists I and J

In the `Map` the `Set` is seen as a type that all threads working in the Map environment can write to. This is especially important in situations where we want to filter data and only want a part of the data. It is of cause also possible to map over a `Set` but sets cannot map with other collections, because sets are unordered.

It is also possible to use a dictionary in a `Map` function, especially if the dictionary maps values to a Set. The fact that the set is a part of structure makes the the dictionary accessible for all threads working on it.

Fig. 2.8 shows a case of advanced mapping, were we both have lists, sets and dictionaries. The example shows a parallel way to make a statistic over how many times a letter is used in a string. What happens is that we map over the String H, illustrated by "hello", with the function addOneForLetter, that

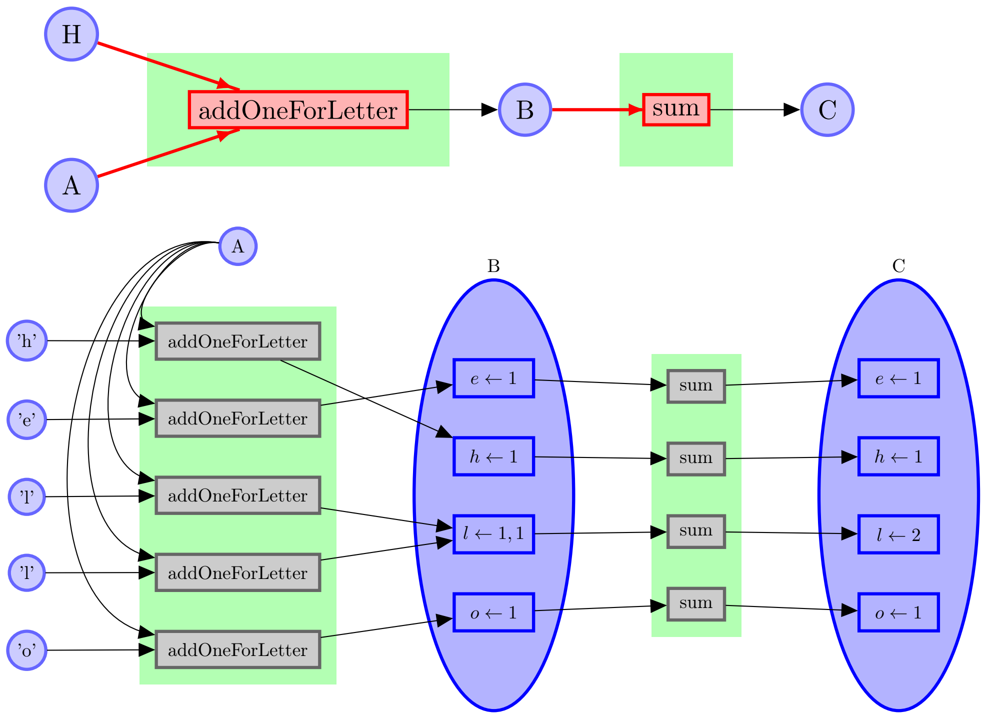


Figure 2.8: Advanced mapping

looks up the letter in a dictionary and then adds 1 to the found set. A is a Dictionary of Sets of Integers. As adding to a set is a parallel operation, we see that the two ones added to the 'l' entry, is put in a single set. The Mapping returns a Dictionary of Sets of Integers we call B. We want to be able to find the occurrences of a letter in the string, when we query the dictionary, not a Set of ones. This means we need to sum the sets. We do this by mapping over B with the sets sum function and this produces a new Dictionary of Integers, that describes the occurrences of the letters.

Recall As **Flow** does not allow for loops at all, a special function has been added to optimize recursion. Of cause it is always possible to reference the function we are currently running, but recursion, if done wrong can take a heavy memory toll, especial if we have something that should mimmic a infinite loop, like a rendering cycle.

The function is called `recall` and it looks exactly as a normal recursive call, but it is guaranteed that no operations is made on the output, thereby it is the last function called in the parent function. The parent function is thereby per definition tail recursive. The big advantage of this system is that it is possible to translate the function to a stack free loop call, which saves memory.

2.3 `FlowCode`

To make a proof of concept and to ease the testing of **Flow** . I have created a syntax that is easily translated to **Flow** . It is called `FlowCode` .

When I read through the different versions of syntaxes and compilers for parallel languages it hit me, that one of the key points was to try to take the programmers serial code and translate it to parallel code. I think that it is a mistake. If we want to make a parallel language, then the programmer should be aware that they are writing parallel code. One of the biggest problems in imperative parallel programing languages is the implicit workflow. The program is all ways run from the top to the bottom. This gives a lot of complications, especially when working with multiple threads and thereby also two pieces of separate implicit workflows. Suddenly we do not know which of the workflows, that should be executed next. It is therefore important that the workflow of `FlowCode` is explicit, and got a parallel feeling to it.

So we got some demands to what `FlowCode` should do:

Explicit Workflow `FlowCode` should have a explicit workflow. The programmer should have a total control over which function is run next time on which data.

Easy Translated `FlowCode` should look as much as **Flow** as posible. Meaning that it should contain notions of a pipe and a function, and functions names are the same as in **Flow** .

Allow Experiments `FlowCode` should implement the **Flow** functionality, and to ease the actual compiling of the **Flow** it should be able to inject C code.

2.3.1 Syntax

I have created a total syntax description in the appendix. But here is the overall idea. The basic construct of **Flow** is the statement. A statement is a continuous string of functions and pipes separated by colons. Statements are terminated with a semicolon.

Pipes	Function
`[a,b,c]`	`afunction`
`[12,0]`	`+`
`["HelloWorld",p]`	`print`

Figure 2.9: The Pipes and Function construct

Figure 2.9 gives some examples of functions and pipes. If some pipes is connected to a function like `[a,b]:function`, then the pipes `a` and `b` are sources of the function, and the function is the reader to `a` and `b`. And the same thing happens when `function:[d,e,f]`, then `d`, `e`, and `f` are targets of the function, and the function is of cause the the writer to `d`, `e` and `f`. If we use the example from before with

$$A = 10 + 2$$

this is coded in `FlowCode` in Figure 2.10.

```
[10,2]:+:[A];
```

Figure 2.10: The equivalent `FlowCode` of $A = 10 + 2$

It is possible to connect pipes with pipes and functions with functions, but this automatically adds an extra pipe section or function between them to uphold the rules of **Flow** , as it is seen in Figure 2.11.

FlowCode	Same as
`[a,b]:[c,d];`	`[a,b]:doNothing:[c,d];`
`[a,b]:+:negative:[c];`	`[a,b]:+:[value]:negative:[c];`

Figure 2.11: The Pipes and Function construct

To make a program we can connect multiple of these flows, but since the language has an explicit workflow the line order is irrelevant. An example of this is showed in Figure 2.12.

The language also holds notions of structs, functions, methods, modules, templates and interfaces, but to know more of this and see more code-examples look

```
Method 1            =        Method 2

[10,2]:+:[A];                [A,B]:-:[C];
[15,3]:+:[B];                [15,3]:+:[B];
[A,B]:-:[C];                 [10,2]:+:[A];
// result C = 6              // result C = 6
```

Figure 2.12: Example of explicit workflow

in the Appendix. But as teaser I will just show an "Hello World" example in `FlowCode` and the corresponding **Flow** graph, see Figure 2.13.

```
module helloworld;
import flow.IO;

def [IO in]:main:[IO out] {
   [in]:pull<stdout>:[p,io];          // remove stdout from IO
   ["Hello World!\n",p]:print:[p_1]; // print "Hello World!"
                                      // in the terminal.
   [p_1,io]:push<stdout>:[out];       // put the stdout back.
};
```

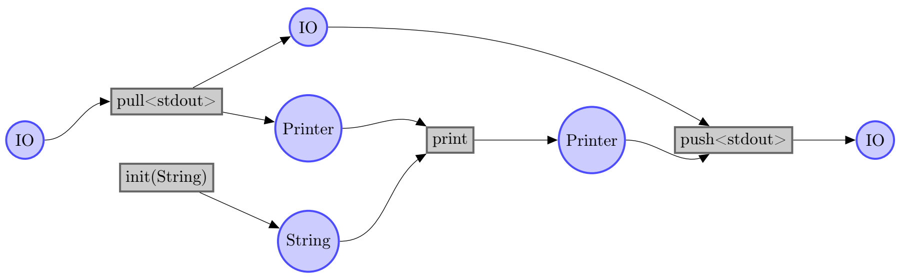


Figure 2.13: Hello World in `FlowCode`

2.4 Compiling Flow

But how can we convert the **Flow** into parallel code? The strict ordering of the layered dependency graph ensures us that if we run all dependencies of a node, before running the node itself, then every thing is run in the correct order.

The general idea is that we can run the nodes dependencies in parallel, since no state is stored in the functions. One of the most unforgiving things about parallelizing code is that it at max can give a linearly, over the number of processors, speed up. This means that understanding the nature of the hardware is important, as every optimization counts.

But this does prove to give some complications as the hardware is not tuned for small problems, but for large continuous problems. I have found these point important.

Synchronization Often in parallel code there is a lot of synchronization instructions, keeping the threads from making errors. This is bad instructions as they don't contribute to the final product. We would like to minimize this as much as possible, meaning that the threads should work on the same data as little as possible.

Coherence The modern CPU is pipelined, which means that there is a startup and cool down period on calculations. So to make two processors work, changing between problems, could give a worse performance than one running it all. To counter this effect we need to group similar instructions together, thereby making the workflow coherent.

2.4.1 Task parallelism

Task parallelism is when the threads is running different tasks at a time. This can be done by running functions from the same layers of the dependency graph.

There are two approaches to Solve the problem of parallelization. Either we can do it statically at compile time or dynamically at runtime.

2.4.1.1 The Statical Solution

The statical solution would mean, that the code is already distributed amongst the processors at compile time. As we already know what the processors need to run, then compiler can make a lot of preprocessing optimizations. Which heighten our coherence and the processor would preform better in this solution. But this approach would give more synchronization time, and in worst case cause a thread to unnecessarily wait on other treads to finish.

2.4.1.2 The Dynamical Solution

This solution tries to minimize the waiting between the threads by making the threads pull jobs from a job queue. If these jobs are to small then this create a problem, as the time used in switching between the jobs will take more time than the running the jobs. To counteract this problem, we could make a serial decomposition of the graph, meaning that if a path does not allow for parallelization, we group it together into one job.

2.4.2 Data parallelism

Data-parallelism is when we do the same function for an collection of data. This can be done by mapping over collections.

2.4.3 Memmory Management

Memmory management is a hard topic as it needs to be done completely automatic when compiled from **Flow** . We want to use as little memory as possible without suffering to much on the performance.

There is different approaches to memory allocation. They are listed from the memory heaviest to the lightest.

All allocated This is the crude version. At compile time we find out how much memory needed to contain all the pipes values without reusing space. This would be easiest to program. But it uses memory for each operation. So the longer the program the larger amount of memory needs to be allocated. In cases where there is some sort of recursion, then the total memory usage can not be determined at compile time.

Max allocated Determined the maximum concurrent accessed memory used. And then allocate the memory at the start of the runtime. Then the compiler ensures that when some memory is guaranteed not to be used the memory space, then a function is allowed to write a new value to it. This can both be done at compile time with a memory cost, and at runtime with a processing cost.

Dynamic allocation This is finest grained version of the three memory solutions. Here is all allocation and deallocation calculations done at run time.

This make this the most memory efficient version, but constant allocations and deallocations do take a toll on the performance.

2.4.3.1 Versioning

One of the biggest challenges is that we need to store a copy of each active version of all the data. This not a problem on primitive types as Integers and Doubles, but when we need to use it on a struct we get a problem. If we change one fields, then, if we do not do anything smart, we need to copy the entire struct.

By using the pull and push we do not have to copy anything, except if the struct is needed by another function. But with the use of a partial persistent data structure we could keep the extra data used at a minimum, and still find the struct at the correct time.

2.4.3.2 Lists, Sets and Dictionaries

Because we do not know the size of all lists, when we compile, we need to allocate lists at runtime. This does also count for Sets and Dictionaries, but they are more dynamically allocated.

Chapter 3

Implementation

In this chapter I will walk thru the implementations of **Flow** , a Translator and a Compiler. All implementations is done in Java. I have chosen Java of two reasons. The first is that it is a hard typed object oriented programming language. That makes the development faster as many errors are caught at compile time. The second reason is that Java is the language that have used most in my study, which means I can draw on knowledge from other projects. The drawbacks is that Java is slower than compiled languages, but as I am only developing a prototype the focus is on rapid development.

3.1 Flow

Flow is just a directed acyclic graph, with some special nodes. And a graph is easily represented in Java by making objects referring to each other. I have chosen to make the graph double linked, but I recommend only to use the references to sources and writers, i.e. query in the direction of the dependency graph.

The **Flow** implementation contains of two declarations; types and functions. The type declaration is just the description of a new type. A function declarations is a small dependency graph describing the functionality of the function. The dependency graph of a function is described as a collection of FlowElements.

3.1.1 FlowElements

There are two key classes: The `FlowNode` and the `FlowPipe`. These are connected to make the graph. The `FlowNode` represents the function in **Flow** and the `FlowPipe` is the representation of the pipe.

`FlowNode` has two arrays containing FlowPipes, one called targets and one called sources. The arrays is accessed by giving a slot id, represented by an integer. This presents some problems when connecting the `FlowPipe` to the `FlowNode`, as if we also need to supply in which slot the `FlowPipe` is connected to the `FlowNode`. Therefore have I added another class called `FlowSlot` that describes at which `FlowNode` and which slot the Pipe is connected.

3.1.1.1 `FlowPipe`

A `FlowPipe` has a `FlowSlot` writer and an collection of `FlowSlot` being the readers. Besides the direct **Flow** oriented values. The FlowPipes do also contain of a `Type` (see **section 3.1.4**) and a `FlowSize` (see **section 3.1.5**). The `Type` can be used for memory management and for strong type checking. `FlowSize` is a class with no attributes that can be used to check whether lists is of the same size.

3.1.1.2 `FlowNode`

`FlowNode` is an abstract class and is implemented by the build-in functions, like DoNothing, Select, Recall, Push, Pull, Map, and then of cause by all the user defined functions. A class diagram picturing the dependencies is shown in **Fig. 3.6**. `FlowNode` in itself does only contain the **Flow** oriented values, but the subclasses implements other relevant values.

`FlowFunctionNode` is the core part of the nodes, and most of the nodes in a complete program will be of this type. `FlowFunctionNode` extends the `FlowNode` with a `Function` field referring to a function declaration. The Allowed types of sources and targets are now decided by the `Function`.

`FlowMapNode` contains a `FlowSize` and a `Function` (see **section 3.1.3**). The `FlowSize` is used to guarantee that the pipes we are mapping over have the right sizes. The `Function` is the internal **Flow** , that is run for each of the map-iterations.

3.1.1.3 Unmodifiable

The original Idea was that the graph should be unmodifiable after creation to ensure that no operations destroyed or altered the graph. This should prevent that running a compiler altered the **Flow** . Sadly, it was to complex to make an entire graph without making changes, especially in the stages where a Map was inserted automatically when translating.

Instead of a completely unmodifiable structure, I added a possibility for the programmer to knowingly changing the structure. This was done by adding a reset operator for most of the set operators that affected the structure. The beauty in this solution, is that we can check whether if a programmer is trying to set a property twice, and then we can throw an exception. But if he knows that he wants to reset writer he can do that without getting the exception.

3.1.2 Visitor

All the classes described here implements the `FlowElement` interface. The reason is that the entire flow implements a visitor pattern that enables us to build tools to **Flow** without altering in the structure of **Flow** . A Visitor allows us to build different compilers using the same structure.

The `FlowVisitor` is an abstract class that can be extended to build tools that works on **Flow** .

Listing 3.1: The FlowVistors abstract methods

```
public abstract class FlowVisitor {
  ...
  public abstract void visit(FlowSelectNode node);
  public abstract void visit(FlowFunctionNode node);
  public abstract void visit(FlowMapNode node);
  public abstract void visit(FlowPipe pipe);
  public abstract void visit(FlowRecallNode node);
  public abstract void visit(FlowPullNode node);
  public abstract void visit(FlowPushNode node);
```

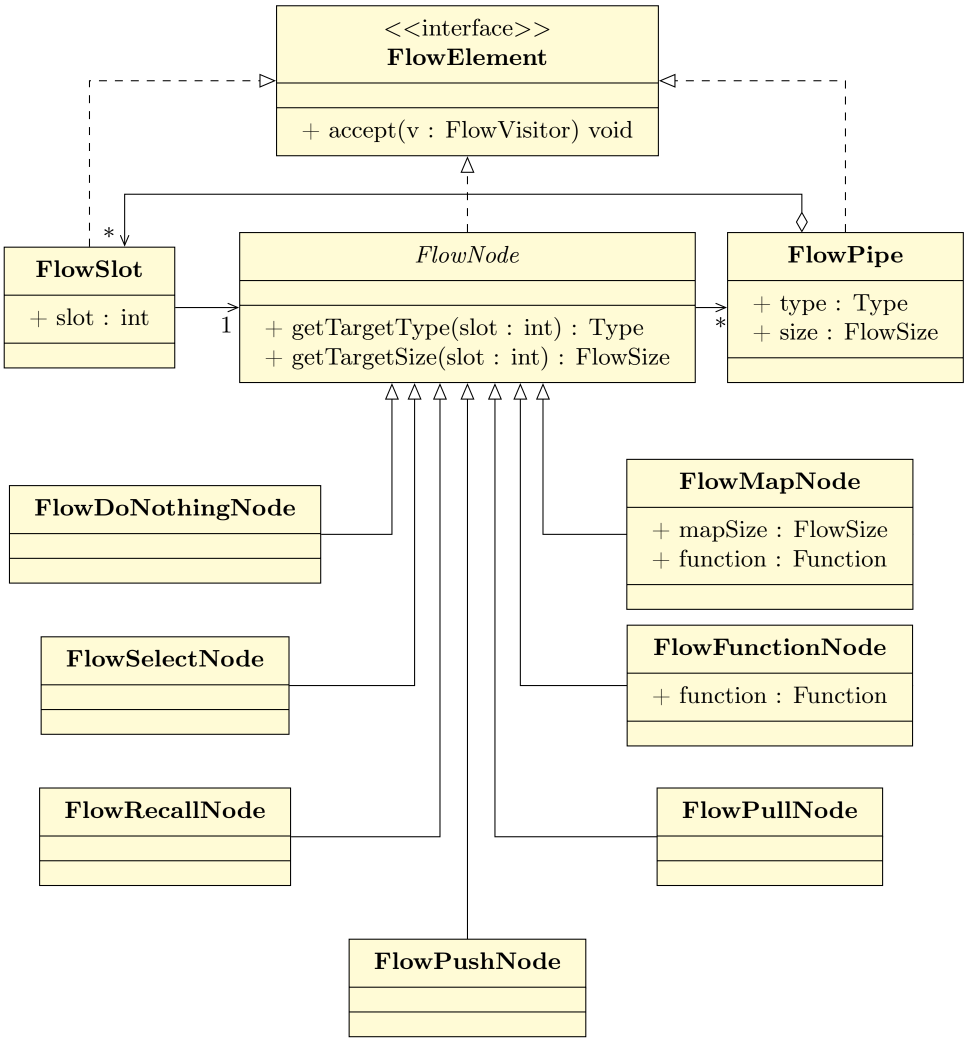


Figure 3.1: Class diagram over FlowElements

```
  public abstract void visit(FlowDoNothingNode node);
}
```

Using a visitor pattern gives many advantages. First of all can we build tools to **Flow** without altering the classes. Second of all is the pattern extremely robust; We get an compilation error if we do not remember to implement the handling of some of the FlowElements, and the visitor is guaranteed to reach all elements, and do it in the same order every time. This makes it easier to make a compiler, because all the transversing is done by the visitor.

Disadvantages with the system is that we can not store data in the structure which means if we want to store associative data, we need to store it in a collection on the side.

The `FlowVisitor` contains some additional features. It make sure that every element is only visited once, this is not something that the programer needs to think about at all. Internally the checking is done by looking up the element in a Set with all the visited elements.

Besides that, the `FlowVisitor` also allows for an easy way to run the program and catch exceptions:

Listing 3.2: The FlowVistors run method

```
public void run(FlowElement element)
    throws FailedFlowVistingException {
  try { element.accept(this); }
  catch(FlowVisitorException e){
    throw new FailedFlowVistingException(e);
  }
}
```

`run` catches all exceptions of type `FlowVistingException`, which is a runtime exception, and transforms it to a Exception that the programer needs to handle. This gives the programer a secure way of throwing exceptions while developing tools for **Flow** .

I have developed some tools that uses this visitor-pattern, to show the idea.

FlowCodeFinalizer is a tool developed simultaneous with the `FlowCode` translator. And it checks that everything is put correctly together. A more detailed description can be found at **section 3.2.2**.

FlowFlattener is a tool that flattens the flow. Often when building programs we would like to build them modular so that we can reuse code we have written. This is sadly not good for performance, because sometimes a function can start the calculations with only half of the arguments.

An Example is a function that returns a random number between two doubles. The function would then take three arguments; two doubles and a random generator. To calculate this we usually we start with getting a random number from the generator between 0.0 and 1.0. We multiply this number by the difference of the min and the max value and then add the min value. The problem is that if we do not flatten the function, we can risk, even though we have the min and max values, that we can not begin calculating the difference before we have the random generator. And we can not get a random value from the generator before we have the min and max value.

This is where the FlowFlattener comes in handy. It takes all functions that is not recursive and chain them on the existing **Flow** .

The FlowFlattener is build in with a ability to check for recursion, because if a function is recursive we cannot flatten it. The idea behind the check is simple, every time we reach a function we would like to flatten, we create a new FlowFlattener that knows it has been called from the current flattener, and the function it's currently trying to flatten. When a suspect function emerges the sub flattener can ask if its equal to the function its working on, if it is not, then it ask its parent whether the function is something that the parent is working on and so does the parent and so on.

If the function is a recursive call the flattening of the function is skipped using an exception callback.

The FlowFlattener returns an complete new Flow separate from the input.

FlowSizeSetter is a small tool, used by the FlowFlattener to sets the sizes of the pipes after completion. This is a separate tool to make programing easier.

3.1.3 Function

The `Function` is a packaging device for **Flow** , so that small parts of a program can be build separately and used from a module. The `Function` is mainly used in the `FlowFunctionNode`, that uses the `Function` to check if it is connected correctly. Therefore a `Function` needs to know its source types and its target types. A special method called `is` can be used to check that the function is called correctly, and a method called `getTargetType` can get the type of the target at a certain slot.

To support the checking of the sizes of Lists in the **Flow** it is possible to query a target size in a function. The method is called `getTargetSize`. A special approach is needed to find the size, because the size cannot be transferred directly from the internal **Flow** of the function. The reason for this lies in how we check the size, see **section 3.1.5**. The Idea is to return the number of the source, that the output has the same size as, and to return -1 if none of them matches.

Here is a walkthrough of some of the function subclasses that can be seen in **Fig. 3.6**.

`FlowFunction` The flow function is the main function in **Flow** , it contains a sub flow that can be run.

`CodeFunction` As this project is only a prototype I have added simple C code injections for basic features like + and -. In a final product this would have been substituted by a dynamic set of classes that the compiler could implement.

`InitFunction` The init functions makes new pipes, so that functions can use them. The reason that they have been given their own classes is because they do not have an internal flow, and that the operations are directly associated with memory allocation which is not something that we would like to inject.

3.1.4 Types

Types is used when testing that the pipes are connected correctly to the functions in the **Flow** . The types class diagram, see **Fig. 3.3**, describes the current definition of the type. The types is one of the parts that I have worked less on, and it is therefor also longer from the final product, than the other features.

First of all the `CodeType` would not exist, as with the functions, the formulation of the build in types should be let up to the compiler within the definitions of **Flow** . The `RenamingType` is a artificial type that I have added to **Flow** to allow `FlowCode` to redefine names easily. The proper way to rename a type, would have been to build it into the checking system.

The staying parts of the types hierarchy would be the `Type`, the `TemplateType`, the `StructType`, and the `ListType`. Besides this a `SetType`, a `DictType`, and a `DefinedType` should be added. The `TemplateType` is just a abstract class that allows for a template.

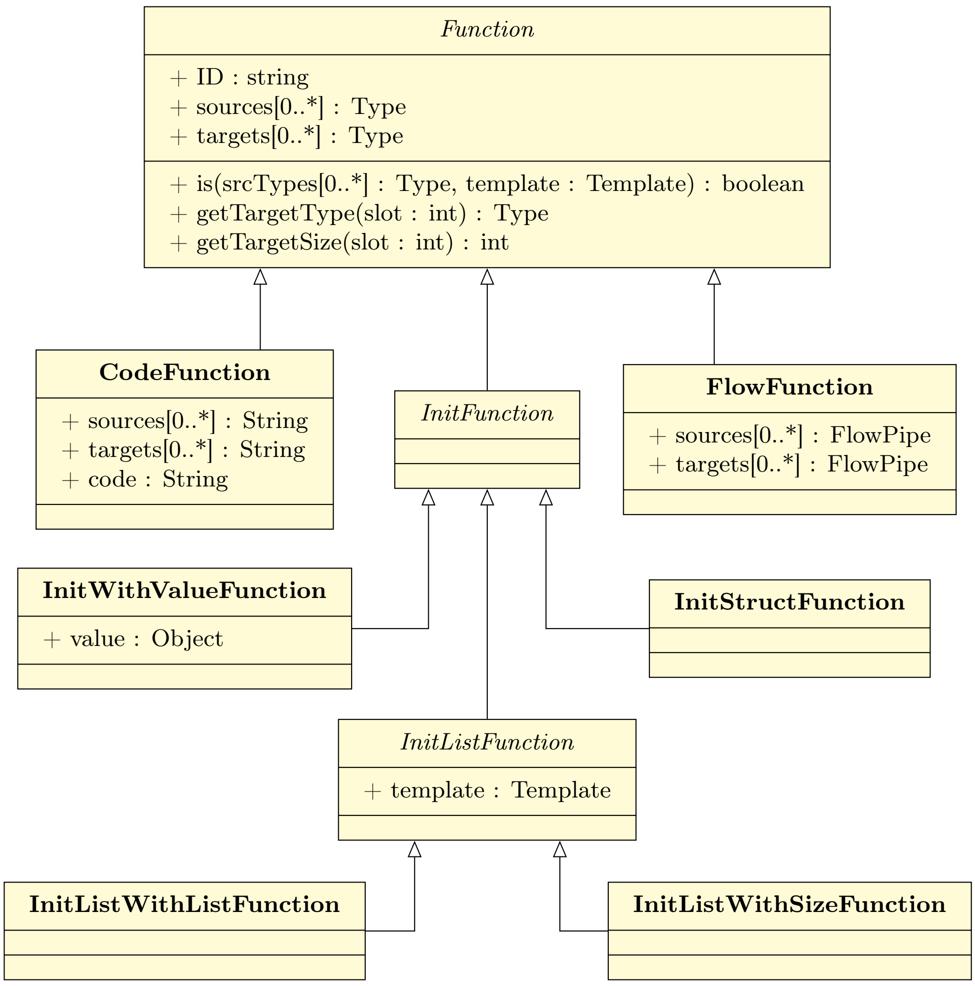


Figure 3.2: Class diagram over Functions

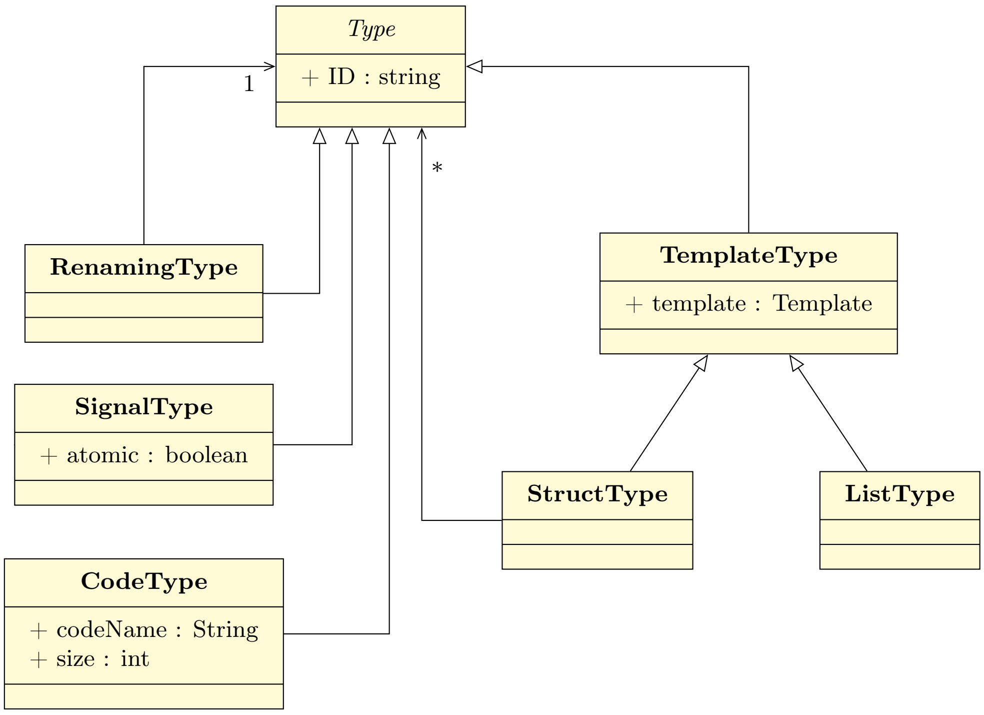


Figure 3.3: Class diagram over Types

The List, Set and Dict types is special types that represent the Mappable types. The Struct type is a bit more interesting, as it holds a structure associating field names with types. The `SignalType` is a build in type that represents a non-memory resource, and can safely be ignored by the Compiler. Lastly is the `DefinedType` which is a type that we claim is a build-in part of **Flow** .

3.1.5 FlowSize

I have talked about the `FlowSize` before, but in this section I will talk a little about the functionality of the class. `FlowSize` have no functionality at all, except that it can compare itself with others objects of type `FlowSize`.

To be able to compare the sizes of single object like a Integer, and the sizes of List and Sets, I have created a Static instance of `FlowSize` called `SINGLE`. The instance is pointed to by all single objects, all collections point to their own instances or an other collections instance. If two collection points to the same instance of `FlowSize`, **Flow** guarantees that they have the same size.

The fact that `FlowSize` comparison are context depended also explains why it is impossible to compare the sizes of a flow directly with the sizes in a function. The function could be called with different sizes and should accommodate all.

Internally the `FlowSize` is set by functions like `InitListWithListFunction` or by mapping operation, and hopefully by a padding function in the future.

3.1.6 Module

The top structure in **Flow** is the Module. It is contains the set of function and type declarations, that are associated with the **Flow** and an ID. This can be used to precompile packages to speed translations.

3.1.7 Code Securing

I have tried to make **Flow** as robust as possible by testing, almost every move done by translator. If something goes wrong the **Flow** will throw a `FlowBuildWrongException`, which is a runtime exception. When building a Translator then the `FlowBuildWrongException` should be handle like a `NullPointer-Exception`, by prevention. This means that a `FlowBuildWrongException` should not be caught, as it indicates a error in the logic of the Translator, not in the translated code.

3.2 `FlowCode` Translator

I have build the Translator using Java, which is the same language used in **Flow** . The translator starts by building a parse tree using ANTLR.

ANTLR is a program that produces code to Java, or another platform, from Parser and Lexer code. As I have already use ANTLR in some of my previous courses it was an obvious choice. ANTLR allows for both extraction of auto generated parse trees and for Java code injection. The code injection enable me to build my own parse tree. I have chosen to use a parse tree that I designed myself, to get better control over the process, and because it was the approach that I used before. But the other approach could possible spared me some developing time, especially because changes in the structure would have been directly implemented, without changing anything but the ANTLER code.

I will not get into the details of the parse tree and the ANTLR code, as this is pretty much strait forward, but cumbersome.

3.2.1 General Translation

I translate `FlowCode` to **Flow** in two stages first we define and then we translate. The first stage defines the elements to the TranslateEnvironment and then the second stage completes the translation of the product.

The stages are important because `FlowCode` does not require the programmer to write the functions in a linear fashion. The `FlowCode` translator uses a two passes.

3.2.1.1 `TranslateEnvironment`

The TranslateEnvironment is a vital part of the translation. It holds all information about the current translation. When defining something on the environment a new declaration is added to the environment which allows other declarations to use it. We therefore have two classes a FunctionDefinition and a TypeDefinitions which helps The TranslateEnvironment to keep track of the Types and Functions defined.

`FunctionDefinition` The function definition holds function declarations of the same name, as overloaded functions in `FlowCode` , i.e. function declarations with the same name but different source types. Because we use automatic mapping, collections of a type is equal to the type when comparing source types.

The Function declarations are stored internally in a list, and to get the a specific declaration out again, the following method can be used.

```
getFunction(srcTypes[0..*] : Type, template : Template ) : Function
```

The method check thru all the functions in the definition and returns the first function that gives a match. The method can certainly be optimized, but it wasn't important for the prototype.

`TypeDefinition` Works a lot like the `FunctionDefinition`, the major difference is that we store type declarations instead of function declarations and that the method used to get the Type is:

```
getType(template : Type) : Type
```

which uses the build-in `hasTemplate` method of the Type, to find the correct type.

But before we can put the function and type declarations in the Definition Objects we need to define them in a way so we can reference them.

To do this a Factory pattern is used. The Factory pattern allows us to create some temporary object that can build the type and function declarations, which is useful as we then can translate the parse tree in stages. A class diagram over the Factories is given in **Fig. 3.4**.

The definition step is simply to create an empty function or type and fill in the necessary informations so that it can be used in a flow. One Problem is that a function needs the types of the arguments to be defined. The define procedure is first to define the types, then the Structs, because they may need a type template, and then lastly we define the function.

This brings us to the next stage, the translation. The translation is done in the same order as the definition. The strict ordering is again needed, as the struct fields are used when we translate the Function.

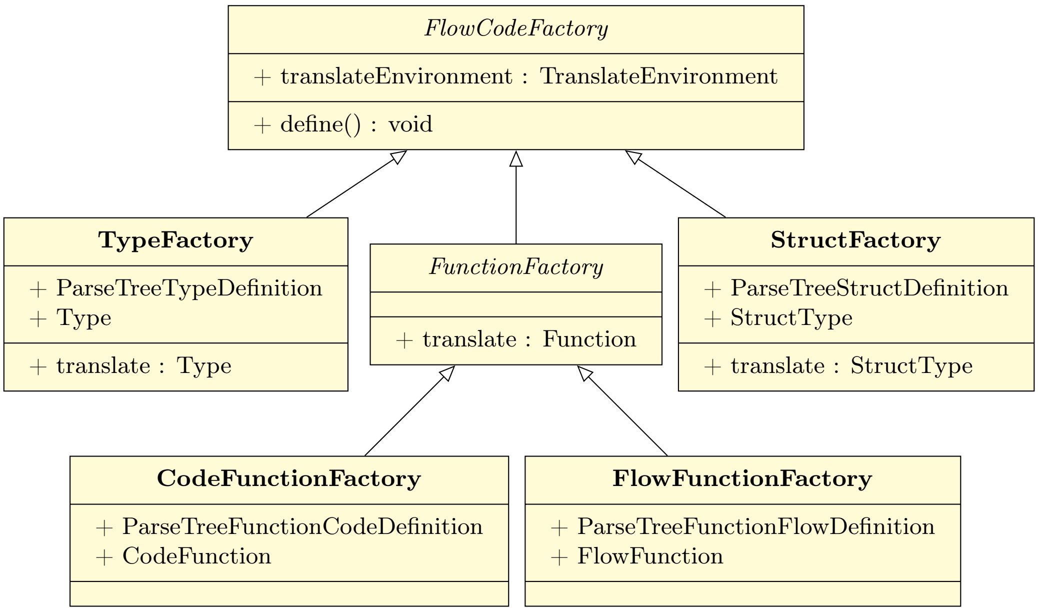


Figure 3.4: Class diagram over Factories

TypeFactory The translation of types is strait forward as they are already translated when defined, except if it is a renamingType, where we need to find its associated type.

StructFactory To translate the Struct we find the associated Types in the TranslateEnvironment, and then assign them in the structure under the field names.

CodeFunctionFactory In the CodeFunctionFactory, we just add the the C code to function.

FlowFunctionFactory This the the most complicated part of the translation, and it is explained in its own section. See **section 3.2.1.2**.

3.2.1.2 FlowFunctionFactory Translation

I will just introduce some terminology: The ParseTrees FlowFunctions are subdivided into statements. A statement is a continuous stream of sections and a section can either be a pipe collection or a function.

The first thing we do to translate the function is to add the argument pipes to a HashMap, called pipes, that associates the names of the pipe to the actual pipe. We later use this to connect the functions with the pipes.

Then we translate the statements one at a time in a function called

```
handelStatement(statements : ParseTreeFlow)
```

We can not assume a special ordering of the statements, which means that the first statement could be the last in the ordering.

`handelStatement` divides the Statement into sections and then calculate them one at at time. In the sections there are an implicit ordering, therefore can we make assumptions according to the last section. The algorithm for calculating a Statement is as seen in **algorithm 1**

Algorithm 1 handelStatement

```
function ← null
pipes ← null
while $|sections| > 0$ do
    $s$ ← next(section)
    if $s \in F$ then                                  ▷ s is a Function section
        if function $\neq$ null then                   ▷ The last section was a function
            pipes ← pipe[function.numberOfTargets]
            function.targets ← pipes
        end if                                         ▷ pipes are now set if possible
        function ← createFunction(s)
        function.sources ← pipes
        pipes ← null
    else if $s \in P$ then                             ▷ s is a Pipes section
        if pipes $\neq$ null then                      ▷ The last section was a pipe
            function ← doNothingFunction
            function.sources ← pipe
        end if
        pipes ← getPipes(s)     ▷ If pipes do not exist in the HashMap, create
        if function $\neq$ null then                   ▷ This section is not the first
            function.targets ← pipes
        end if
        function ← null
    end if
end while
```

The algorithm tries to create the data flow, while knowing the the former section.

The type of the section is decided by whether the pipes and function is null or not. If both the pipes and the function is null then the section we are working on is the first in the statement. If the function is not null the function was the last read, and the same goes for the pipes.

Every time we handle a pipes section we need to either find them in HashMap or create them and put them there. If we do not do this we cannot find the pipes again, if look for them, when we handle the rest of the sections.

The FlowFunctionFactory translation is ended by running the `FlowCodeFinalizer` tool on the flow.

3.2.2 FlowCodeFinalizer

The FlowCodeFinalizer is a tool using the visitor pattern of **Flow** . It inherits the `FlowVisitor` class and as such it is guaranteed to reach the entire construction.

The approach to building this was relatively simple.

- The First thing the tool was build for was to give all Pipes a type and to ensure that all the functions was loaded. In this functionally lies an automatic check that the pipes are connected correctly.
- The next part of this tool is to build maps, as `FlowCode` has implicit mapping this does at the same time check that all lists that uses the same map are of the same size.
- To a certain extend the finalizer checks that the pull/push relationship of a struct has not been violated. But this functionality has not yet been extended to check within functions. Structs are at the time not permitted to enter functions with some of its fields locked.
- Lastly we also check that the system do not use pipes that are the targets of a recall function.

The Type checking is remarkably easy when using the visitor pattern and the functionality of **Flow** . The actual logic for setting the types for all the pipes can be done in a very few lines of code.

Listing 3.3: FlowPipe Handling in FlowCodeFinalizer

```
public void visit(FlowPipe p) {
  try {
    // set the type and size of the FlowPipe
    if (p.getWriter() != null) {
      p.setType(p.getWriter().getTargetType());
      p.setSize(p.getWriter().getTargetSize());
    }
    // Checking that all is set for the cases where there
    // is no implicit writer.
    if (p.getType() == null || p.getSize() == null)
      throw new PipeNeverWritenToExecption(p);
  } catch (CouldNotTranslateException e) {
    throw new FlowVisitorException(p,e);
  }
}
```

The checking is done automatically when we evaluate the functions, because when we try to get a function from the TranslateEnvironment it needs to be declared, if it is not a `FunctionNotDefinedException` will be thrown.

Listing 3.4: FlowFunctionNode Handling in FlowCodeFinalizer

```
public void visit(FlowFunctionNode p) {
  try {
    // Get types from pipes.
    List<Type> types = p.getSourceTypes();

    // Get current function
    Function function = p.getFunction();


    if(function == null) {
      // If currently don't have a function,
      // then try to find a new one.
      function = getEnv().getFunction(
        p.getID(),
        types,
        p.getTemplate() );
      // NOTE : that this function will fail with an
      // exception, if no function pressent
      p.setFunction(function);
    } else if(!function.is(types, p.getTemplate()))
      // We have function but it doesn't fit our pipes.
      throw new WrongArgumentsToFunction(
        function,types,p.getTemplate() );
    ...
  } catch (CouldNotTranslateException e) {
    throw new FlowVisitorException(p,e);
  }
}
```

To build the map is pretty simple. First we check if anything needs mapping, this is done by comparing the output of the FlowCode source checker with the Flow source checker. Because the FlowCode source checker sees List of a type as the type itself when checking and the Flow source checker does not, then if it is correct in FlowCode and not in Flow we need to make it into a map. To make the FunctionNode into a map is relatively easy. First the Map takes over the FunctionNodes function, then we move all the sources pipes to point to the Map instead of the FunctionNodes, and lastly the same is done with the targets. The FunctionNode now completely detached from the **Flow** , should be deallocated by the garbage collector.

Listing 3.5: FlowMapNode Creation while Handling the FlowFunctionNode

```
    // Trying to pass to a map;
    if(!function.is(types, p.getTemplate())){
      // Because the isSameTypes call does not differ between
      // List<A> and A, and is does. We can see that if is
      // doesn't pass we need to map this function.
      FlowMapNode mapnode = new FlowMapNode();
      mapnode.setInternFunction(p.getFunction());
      // Set the Function
      for(int i = 0; i < p.getNumberOfSources(); i++) {
        // Move the source pipes;
        mapnode.setSource(i, p.getSource(i));
        p.getSource(i).removeReader(new FlowSlot(i, p));
        p.getSource(i).addReader(new FlowSlot(i, mapnode));
      }
      for(int i = 0; i < p.getNumberOfTargets(); i++){
        // Move the target pipes;
        mapnode.setTarget(i, p.getTarget(i));
        p.getTarget(i).resetWriter(new FlowSlot(i, mapnode));
      }

      mapnode.accept(this);
    }
}
```

After this we make the mapnode accept the `FlowCodeFinaliser`, where a check is made that all pipes that is mapped is of the same size as the Map.

The tools also checks that the pushing and pulling is done correctly. To keep track of in which state the different structs are, I have added a Struct checker class. The Struct checker has two functions, to store the state of the Structs fields, and to enable easy checking of these field. Each Struct checker class is associated with several pipes, the FlowNodes that do not alter the struct can associate the targets of the FlowNodes with Struct checker of the source. That we pull and push correctly, is checked in the push and pull nodes.

We check that we do not call anything with a locked pipe in the function and the recall node, by the following code snippet:

Listing 3.6: The locking check for recall and struct

```
// Check that none of the pipes is locked;
for(int i = 0; i < p.getNumberOfSources(); i++) {
  if(isRecallLocked(p.getSource(i))
    || isStructLocked(p.getSource(i)) )
    throw new PipeLockedAndCantBeAlteredException(
      p.getSource(i) );
}
```

The `isStructLocked` method, checks that none of the fields in the struct are locked.

The `isRecallLocked` method is part of the last thing that the FlowCodeFinalizer checks. After a recall all targets get locked, regardless of type, this means that they cannot be used by any thing but the select node. Ensuring that the resulting flow is tail recursive. The locking is controlled by a simple `HashMap`. When the true or false source of a select node is locked, the target will be locked too.

3.2.3 Modules

Its a vital part of the syntax to be able to import modules, which allows for better reuse of code. `FlowCode` uses the build in Modules from **Flow** to do the job. This section will put together all the implementations from the previous sections.

The process of translating a module is strait forward, and is done in the `FlowCodeTranslator`.

1. Check that the Module hasn't been loaded before. This works to a certain extend but this feature isn't vital for the prototype.

2. Find the src code in the filesystem from a module name. A module name is like the filesystem names but divided by dots like in Java and Python. The TranslationEnvironment holds the library directories that should be searched in.
3. Run the ANTLR Lexer and Parser on the file.
4. Import and translate necessary modules, this is done recursively with the FlowCodeTranslator.
5. Define types, structs and functions, using the factories.
6. Translate types, structs and functions.
7. Add the types, structs and functions to the module and return it.

3.2.4 Exceptions and Traceability

If there is errors in the code then it is important that the translation fails. The easiest way to do this is to use the exception structure of Java.

All my exceptions inherit the same `CouldNotTranslateException`, which all can take a Message and an `Exception` as parameters. This allows me to build a structure where each layer of the translation throws its own subclass of CouldNotTranslateException. If an exception is throw in one of the low layers the next layer will throw a new exception of its own, convoluting the exception. That way there is full traceability down the layers and when we in the end output the `Exception`, we get a complete and almost readable message.

Listing 3.7: An error message

```
Couldn't translate module test7, problem while translating caused by:
CouldNotTranslateFunctionException – Problem in main, while checking caused by:
FailedFlowVistingException – Exception raised in <Pipe: null argument > caused by:
PipeNeverWritenToExecption – The Pipe never written to <Pipe: null argument >
```

The Error messages notes that the pipe was never written to, while checking the internal **Flow** of the function main, which is a function of the module test7.

3.3 Compilers

The compilers I have provides are both using the visitor pattern, Which makes them stron

3.3.1 Tikz compiler

To show that the `FlowCode` translated correctly, and to illustrate that it is possible to build a compiler to **Flow** , I have created a tool that compiles the **Flow** to Tikz code. The Tikz code is then renderd to something like this:

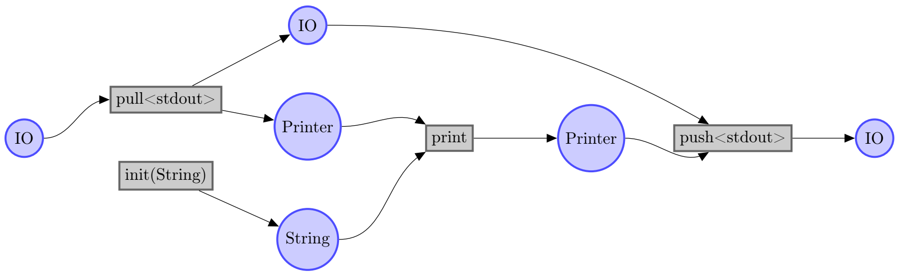


Figure 3.5: Finished product

Tikz is a extension to Latex that makes it easy to make vector graphics. You simply write the Tikz code and embed it in a latex document to make it run. I have created a little piece of example code, that shows a small part of the functionality.

Listing 3.8: Tikz example code

```
\begin{tikzpicture}
  \node (A) [draw=red] at (0,0) {$a_\text{label}$};
  \node (B) [draw=blue] at (2,0) {$b_\text{label}$};
  \path (A) edge (B);
\end{tikzpicture}
```

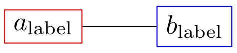


Figure 3.6: Output of small example of tikz

It is of course possible to add different styles and stuff to both the edges and the nodes.

I wanted the drawing to be a layered data flow graph. The Idea behind the compilations was to give all elements a level ID describing when they were met. And then group nodes with the same level IDs together in a column, in the order of the level IDs.

The way I have implemented this is by using a `HashMap`, associating the element to a level ID, and then an `Arraylist` of `List`, containing the pipes of each level. I assign the ID's by a simple algorithm:

Arguments to the function we want to compile get 0 and calls to functions with no arguments get the value 1. Then after this we set the level ID for the next elements to be the value of its writer or the max value of its sources level IDs plus one. The transversal uses the Visitor tool from **Flow** to make sure that all nodes are visited.

The actual compiling to code is simple as we just print the elements as nodes, which we bundle in matrices. This produces automatic centering. The naming of the nodes, is based on layer-<layer number>-<number in layer>. The names is kept in a HashMap, so when we need to connect the nodes we can lookup the names fast.

Listing 3.9: Example of the layering.

```
\matrix (layer1) [layer] at ($(layer0) + (3.0cm,0)$){
  \node (layer-1-0)[func]{pull<stdout>}; \\
  \node (layer-1-1)[func]{init(String)}; \\
};
```

The connections are made after each matrix is appended to the code. We only connect backwards so we do not need to worry about that some of the nodes is not instantiated.

Listing 3.10: Example of the paths.

```
\path (layer-3-0)
  edge [dir,out=150,in=0] (layer-2-1)
  edge [dir,out=-150,in=0] (layer-2-2);
```

To make the edges between the nodes easier to separate, edges will curl. This did give some problems because there is a bug in the Latex complier that presentates itself when the compiler tries to make long curly edges. I have fixed this by only making edges curly if they are connecting bode less that 13 levels away.

3.3.2 Code Compiler

I did not implement a C compiler, because even though that it would definitely proof that **Flow** can be translated into parallel code, it is a very complex and time consuming process.

An important part of this project is, if not to compile **Flow** into parallel code, to make it plausible that it is possible to compile **Flow** into parallel code. Therefore I have developed a Compiler that produces parallel pseudocode. I call this pseudocode "Simple".

The pseudocode is based on some assumptions.

- That we can access all memory from all positions in the code.
- That we have a structure, where we can add and remove snippets of code, called jobs. It also needs a feature that allows one job to give the green light to other jobs. This could be handled thru the OS with the use of threads, or properly more efficient directly in the code.

The pseudocode will not differentiate between types, and generally not work with memory management. The creation of the pseudocode is done the same way that a compiler would create parallel C code. The compiler first subdivides the tree into serial jobs, consisting of actions. An action could be a calling of a method, allocating memory or allowing an other job to run.

A part of the structure is represented in **Fig. 3.7**. And it is not pretty. Therefore not to make it even harder to see I have not included the last three actions. Even thought they are similar important.

ExitAction The exit Action tells the function that the computations is done, this is essential when clean up operations are needed to be run.

MapAction The Map Action takes a function call action, and a memory position of the size of the Map.

RecallAction The Recall Actions contains two lists of parameters, the first is the current input memory positions, and the second is the memory positions of the function. The idea is that before calling the recall the function simply moves the data from its input parameters to the input of the function, and then runs it again.

All these Actions are then implemented by the specific language compiler to produce the correct kind of code.

To build the Jobs I use a tool based on `FlowVisitor`. It runs through the **Flow** and creates the jobs. We could of cause just create a job for each node in the

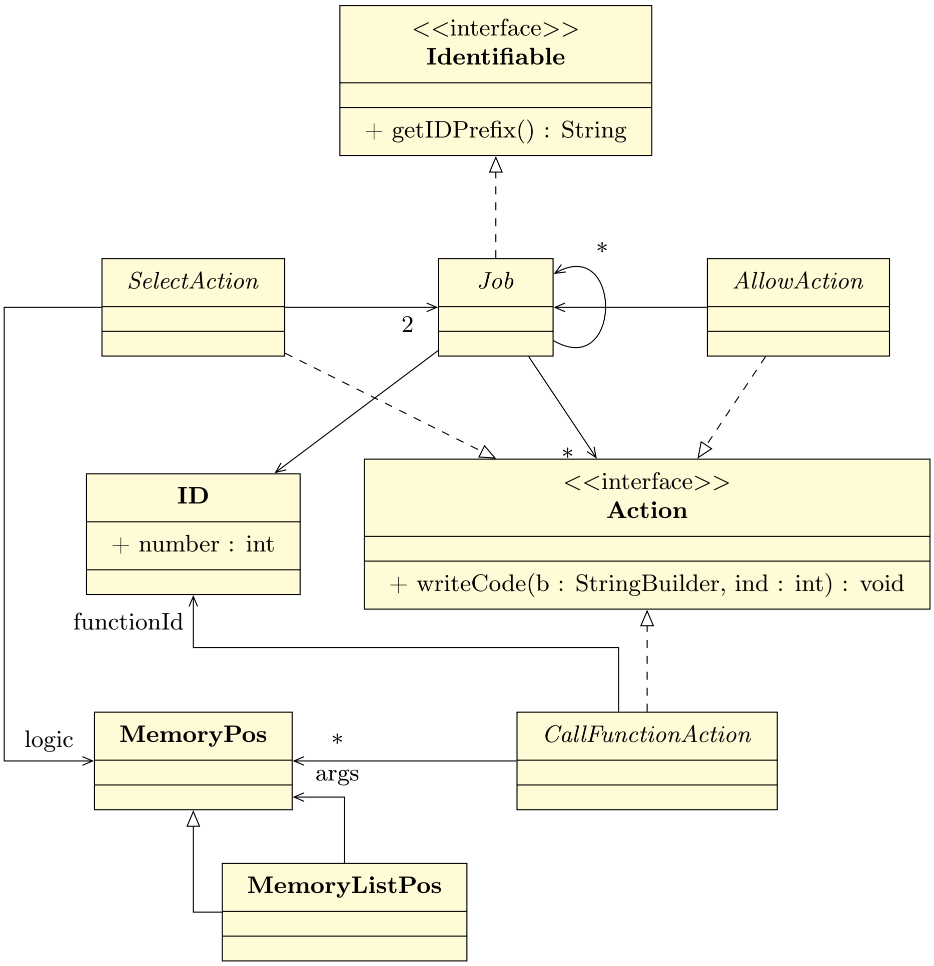


Figure 3.7: Class diagram over the Code-compiler

Flow , but that would not be efficient. So the algorithm needs to join jobs if they cannot be run in parallel.

The direct implementation is a bit messy, but the overall approach is simple.

- All pipes get the Job of their writer, If they don't have a writer they create a new job.
- The nodes is then treated like this.
  1. we find the possible dependencies by running through all the sources of the node and finding the associated jobs.
  2. for each of these dependencies we check that they don't depend on each other. If a job depend on a second job, we can remove the second job from the dependency list as it is already represented by the first job. The dependency check walk trough the entire graph, unless we give it a max depth. The max depth is currently set to 3.
  3. if there is only one dependency left after the filtering, and that the dependency is open for serial access, we can just extend it. A job is open for serial access by a node if one of the node's pipe's writer are the last node added to that job.
  4. if there are more, or zero dependencies we need to make a new job and add all the dependencies as dependencies to it.

3.3.2.1 Problems

During the coding of the Job-compiler I had some problems, the worst of these was with the SelectAction. The select in itself is different from the other nodes as the exact behavior of the select is first known at runtime.

The current problem occurs when one of the, true or false source is dependent on the other. If we only want one to run the other may be blocking it. It is therefore necessary to check for this. Currently I have chosen the simplest possible solution, to allow the sources to compute to the very last step that needs permission to run from the select.

3.3.2.2 Simple

The pseudocode is then produced by implementing the classes with the `writeCode` method. I have implemented one class for each of the Actions that describes how they are supposed to translated to code.

I have called the compilation Simple as it is an simplification of the features of C. And it looks like this:

Listing 3.11: Output from running of test2

```
flowFunction0 [String : arg0]
  init the String into arg0
end

flowFunction1 c-code [String : o]
  printf("%s",*((char**)o));
end

Main []

  on call {
    Allocate local space (2)
    run Job0
    run Job1
  }

  Job0 depend on ()
    allow Job2 to run
  end

  Job1 depend on ()
    call flowFunction0 with [Memory(2135760193)]
    allow Job2 to run
  end

  Job2 depend on (Job0, Job1)
    call flowFunction1 with [Memory(2135760193)]
    Allow Function to exit
  end

  on exit {
    Deallocate local space (2)
  }
end
```

The first thing we se in the top is the function names, these are found using a tool that transverses the **Flow** and finds all required functions. After this we see the actual Main function. It has no arguments, but the first thing we see is the "on call" event. Everything in the event is executed when we call the function; It allocate some local space, and starts Job0 and Job1.

The jobs the allow each others to run in an orderly fashion.

There is also a "on exit" event that cleans up after the function.

As we can see the algorithm for finding jobs is not optimal jet. **Fig. 3.8** shows that a more optimal solution could be found. Job0 does not do anything in our example.

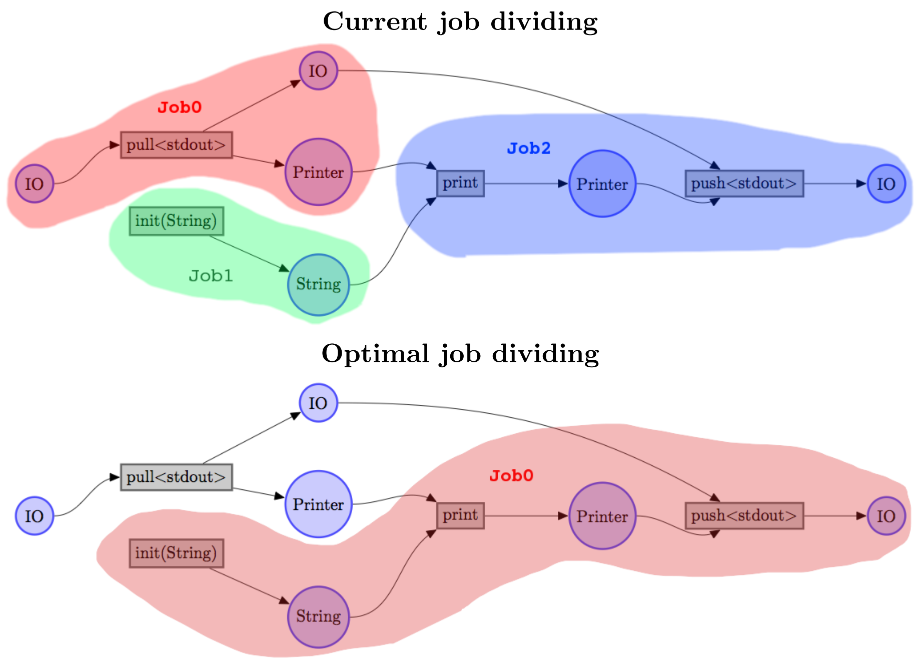


Figure 3.8: Job dividing

Chapter 4

Proof of Concept

This chapter is focused on illustrating the features of **Flow** . The goal of this project has not been to produce failsafe code but to showing and proving that it is possible to create a functioning compiler using a flow approach. The presentation will not cover all cases, but will only cover special cases of interest.

A problem with illustrating **Flow** is that it is a declarative language, which means that each test needs a purpose to be able to run. This makes building simple test cases harder.

All the SimpleCode is also in the appendix.

4.1 Experiment 1 - Setup

In this Experiment, I will talk about the testing environment, which is used by the the other Experiments. I have implemented some standard features in a flow package. To see the entire `FlowCode` Standard Library see **appendix C**.

Setup for the test is simple. First I translate the tests, which is done using the FlowCodeTranslater. From that I get a Module from which I compile all functions to Tikz. I also compile a version of each function after they have been optimized with the flow flattener.

The Simple compiler is only run on the flattened main function of the module, because the compiler compiles a program.

A class runs all the tests after each other and the tests are named after the experiment they are a part of.

4.2 Experiment 2 - Hello World

Experiment 2 is a simple Hello World program and is the example that is used most throughout the report. It covers the features of pulling and pushing a field from a struct and printing it. The Program also checks that the import function works and that the pipes can connect correctly to the functions.

Listing 4.1: The `FlowCode` of test2

```
module test2;
import flow.IO;
import flow.string;

def [IO in]:main:[IO out] {
  [in]:pull<stdout>:[io,p];
  [p,"Hello World!\n"]:print:[p_1];
  [io,p_1]:push<stdout>:[out];
}
```

We have already seen both the Tikz graph and the Simple code, as they is used as illustrations in the

The Tikz generated graph is shown in **Fig. 4.1**.

which looks like this in simple code:

Listing 4.2: Simple code from test2

```
flowFunction0 [String : arg0]
  init the String into arg0
end

flowFunction1 c-code [String : o]
  printf("%s",*((char**)o));
end
```

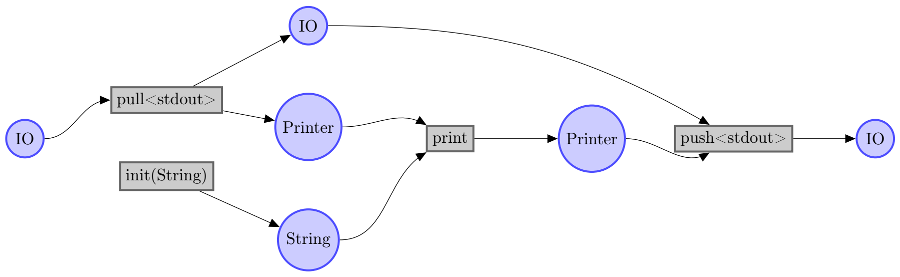


Figure 4.1: Graphical representation of test2

```
Main []

  on call {
    Allocate local space (2)
    run Job0
    run Job1
  }

  Job0 depend on ()
    allow Job2 to run
  end

  Job1 depend on ()
    call flowFunction0 with [Memory(2135760193)]
    allow Job2 to run
  end

  Job2 depend on (Job0, Job1)
    call flowFunction1 with [Memory(2135760193)]
    Allow Function to exit
  end

  on exit {
    Deallocate local space (2)
  }
end
```

4.2.1 Reflections

We can see that we can create a **Flow** from input and that we can compile the **Flow** into something that looks like an imperative language. The Tikz graph looks is nice.

The memory management is working, beacuse we can see that the same string that is instantiated is also used by the printer. Notice that the localspace allocated is 2 this is do to do with that a String is a List, and a list also has a size.

On the bad side is the job creation in the simple output not optimal, and the instantiation of the list size is not show. But this was expected as the functionality was not implemented.

4.3 Experiment 3 - Parallel Program

Experiment 3 is a test that shows that special operator like names is allowed for functions, see the '+' function, and it also produces something that is implicitly parallelizable. The two first plus operations can be run in parallel.

Listing 4.3: The `FlowCode` of test3

```
module test3;
import flow.math;
import flow.IO;

def [IO in]:main:[IO out] {
  [10,2]:+:[a];
  [12,3]:+:[b];
  [a,b]:+:[c];
  [in]:pull<stdout>:[io,printer];
  [printer,c]:print:endl:[printer_3];
  [io,printer_3]:push<stdout>:[out];
}
```

But as the code gets longer so does the Tikz-graph which is shown in two parts; see **Fig. 4.2**.

The Simple code is also longer than it was before, so I only present the part of the main where the Jobs is.

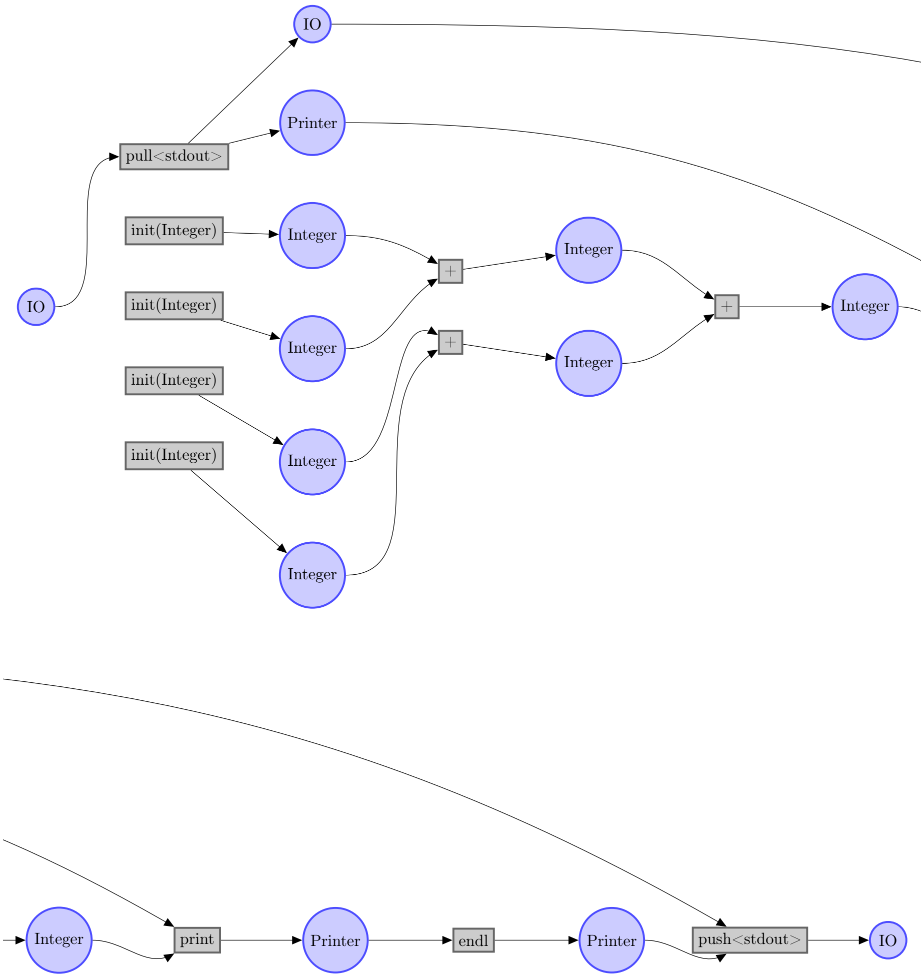


Figure 4.2: Graphical representation of test3

Listing 4.4: Simple of test3

```
Job0 depend on ()
  allow Job8 to run
end

Job1 depend on ()
  call flowFunction0 with [Memory(1109376982)]
  allow Job3 to run
end

Job2 depend on ()
  call flowFunction1 with [Memory(845913220)]
  allow Job3 to run
end

Job3 depend on (Job2, Job1)
  call flowFunction2 with [Memory(1109376982),
      Memory(845913220), Memory(921745400)]
  allow Job7 to run
end

Job4 depend on ()
  call flowFunction3 with [Memory(1975391989)]
  allow Job6 to run
end

Job5 depend on ()
  call flowFunction4 with [Memory(315976503)]
  allow Job6 to run
end

Job6 depend on (Job4, Job5)
  call flowFunction2 with [Memory(1975391989),
      Memory(315976503), Memory(2037510537)]
  allow Job7 to run
end

Job7 depend on (Job6, Job3)
  call flowFunction2 with [Memory(921745400),
      Memory(2037510537), Memory(665360297)]
  allow Job8 to run
end

Job8 depend on (Job0, Job7)
  call flowFunction5 with [Memory(665360297)]
  call flowFunction6 with []
  Allow Function to exit
  end
```

4.3.1 Reflections

The graph is fine and it shows that we should be able to calculate the two plus operations in parallel. We can also see that the parallelism is reflected in the simple code, if we follow the dependencies of the two jobs 6 and 3 which is the jobs associated with the parallel plus operations. Function 2 is the plus operation.

The Compiler also claims that we can run the instanciation of the 4 init function in parallel, which is both a good and a bad thing. It is a good thing because it is right, it is possible to run those in parallel, but the bad thing is that the instanciation of data is often very fast, so the bonus we get by parallelizing the code is often lost in the time of the synchronization.

4.4 Experiment 4 - Mapping and Lists

Experiment 4 illustrates that we are able to make a mapping, and that we are able to use functions defined in flow. The test calculates the Caesars code of a string.

The caesars code is a coding algorithm where you take each letter in a message and then translates it up or down the alphabet. If we use Caesars code with a value of 2 on this sentence, it would look like:

```
Kh"yg"wug"Ecguctu"eqfg"ykvj"c"xcnwg"qh"4
"qp"vjku"ugpvgpeg."kv"yqwnf"nqqm"nkmg
```

The beautiful about this encryption algorithm is that it is perfect for mapping as the algorithm doesn't need knowledge about the former or the next fields in the list, or data from computations of other fields.

The mapping function of the Caesars code has a graphical representation like this:

It takes 2 Bytes and produces one Byte.

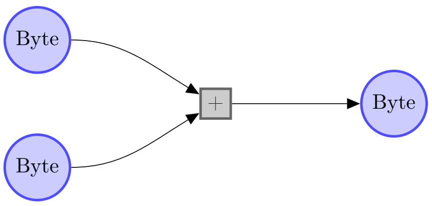


Figure 4.3: Graphical representation of Caesars Code function

Listing 4.5: The `FlowCode` of test4

```
module test4;
import flow.IO;
import flow.string;
import flow.math;

def [IO in]:main:[IO out] {
  [in]:pull<stdout>:[io,p];

  ["Hello World"]:[hello];
  [1]:toByte:[b];
  [-1]:toByte:[mb];
  [hello,b]:ceasarsCode:[code];
  [code,mb]:ceasarsCode:[decode];

  [p,hello]:print:endl:[p_2];
  [p_2,code]:print:endl:[p_3];
  [p_3,decode]:print:endl:[p_4];
  [io,p_4]:push<stdout>:[out];
}
```

As we can see is `hello` a list of bytes and `b` is a byte with value 1. `FlowCode` uses automatic mapping if the function can map. So the the mapping is done over the list with Caesars code keeping the `b` constant to 1.

The graphical illustration is cut up in 3 parts, and can be seen in **Fig. 4.4**. The red functions is the once that is mapped and the red arrows means that this is done for each value in the list. Notice that the the single Byte is not red, this means that **Flow** do not map over this.

The Simple Code is to long to show here but I would like to show the succeeded builded map.

Listing 4.6: Mapping in simple

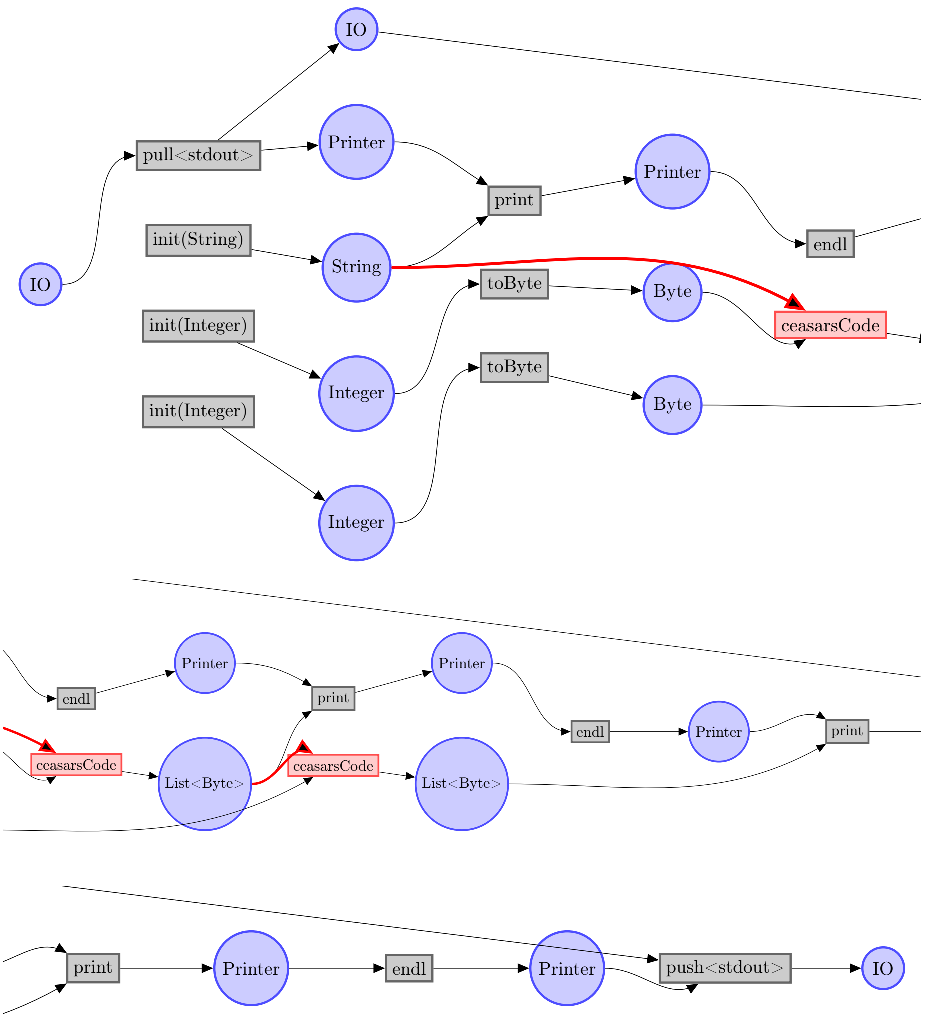


Figure 4.4: Graphical representation of test4

```
  Job7 depend on (Job6, Job4)
    Map from i : 0 to Memory(1545595021):
      call flowFunction6 with [Memory(1083228271)[i],
          Memory(1764128329), Memory(853323835)[i]]
    end
    allow Job8 to run
    allow Job10 to run
  end
```

I haven't done that much on this part as parallel mapping is already an well explored matter in C. But notice that it uses a specific memory position to map to and if we look at the hole output we can see that it is the same as in the other mapping.

4.4.1 Reflections

The mapping work perfect both in the graphical representation and in the simple code.

4.5 Experiment 5 - Recursion and Select

Experiment 5 experiments with the use of recursive functions. The program should print out the numbers from 0 to 4, using the function printRange. printRange uses both the recall and the select functionality.

Listing 4.7: The `FlowCode` of test5

```
module test5;
import flow.IO;
import flow.math;

def [Printer in, Integer from,Integer to]:printRange:[Printer out] {
  [from,to]:lt:[bool];
  [from,1]:+:[nextIter];
  [in,from]:print:[p];
  [p,nextIter,to]:recall:[true_out];
  [bool,true_out,in]:select:[out];
}

def [IO in]:main:[IO out] {
  [in]:pull<stdout>:[io,p];
```

```
    [p,0,4]:printRange:[p_1];
    [io,p_1]:push<stdout>:[out];
}
```

For once is the code not to large represent graphically so here is the main function.

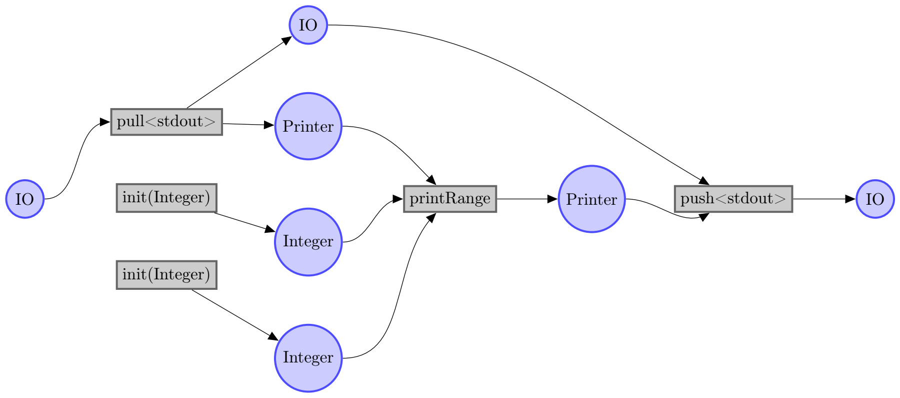


Figure 4.5: Graphical representation of test5's main function

The main function is not that interesting, because all of the nice features happens in the printRange function, which can be seen in **Fig. 4.6**.

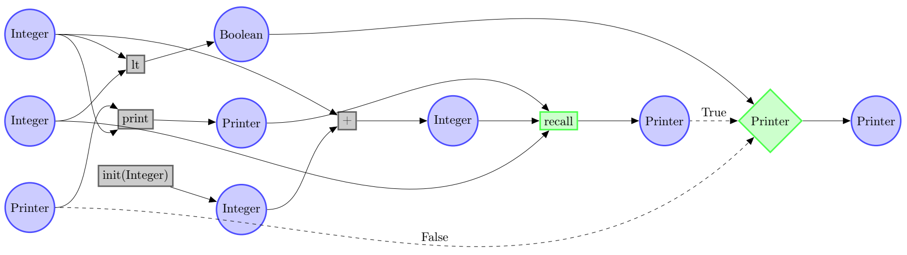


Figure 4.6: Graphical representation of test5's printRange function

We see that every thing is represented nicely. This also transfers nicely to Simple code. The entire code can be found in the appendix, but here is a sample of the select based code.

Listing 4.8: Example of Select simple output

```
Job2 depend on (Job1, Job0)
  call flowFunction2 with [Memory(880403204),
      Memory(774321798), Memory(275558166)]
  if Memory(275558166) is true:
    allow Job7 to run
  else
    allow Job8 to run
  end
  Allow Function to exit
end
```

and a example of the recall code.

Listing 4.9: Example of Recall simple output

```
Job7 depend on (Job1, Job6, Job4, Job2)
  Put Memory(1418257117) in Memory(880403204)
  Put Memory(774321798) in Memory(774321798)
  Recall
end
```

If you notice that the two last operations before the recall, moves the source data of the recall onto the source data of the function. This is done so that the functions can be run again with no reallocation of memory.

4.5.1 Reflections

Nothing much to say about this example except that exact execution of the simple select is a bit unclear at the moment. And a bit more thinking is needed when we need to translate it to C.

4.6 Experiment 6 - Flattening and Structs

Experiment 6 is about advanced struct control and a test of the FlowFlattener tool. The program does simply add values on a Point and then it prints it using a function. A feature that has not been directly tested before is the struct creation.

Listing 4.10: The `FlowCode` of test6

```
module test6;
import flow.math;
import flow.IO;
import flow.string;

struct Point {
  x : Integer,
  y : Integer
};

def [Printer p, Point point]:print:[Printer p_5] {
  [point]:pull<x>:[point-x,x];
  [point-x]:pull<y>:[_,y];
  [p   ,"["]:print:[p_1];
  [p_1, x ]:print:[p_2];
  [p_2,","]:print:[p_3];
  [p_3, y ]:print:[p_4];
  [p_4,"]"]:print:[p_5];

}

def [IO in]:main:[IO out] {
  [in]:pull<stdout>:[io,p];
  Point:[point];
  [point]:pull<x>:[point-x,x];
  [point-x]:pull<y>:[point-x-y,y];
  [10,x]:+:[new_x];
  [10,y]:+:[new_y];
  [point-x-y,new_x]:push<x>:[point+x-y];
  [point+x-y,new_y]:push<y>:[point+x+y];
  [p,point+x+y]:print:endl:[p_1];
  [io,p_1]:push<stdout>:[out];
}
```

I have of cause also made a graphical representation of the two functions, first one of both of them without the use of the FlowFlattener, **Fig. 4.10** & **Fig. 4.8**, and then the one that that is the product of the flattening, **Fig. 4.9**.

The simple code does not do anything new in this, because we always run the simple compiler on a flat **Flow** .

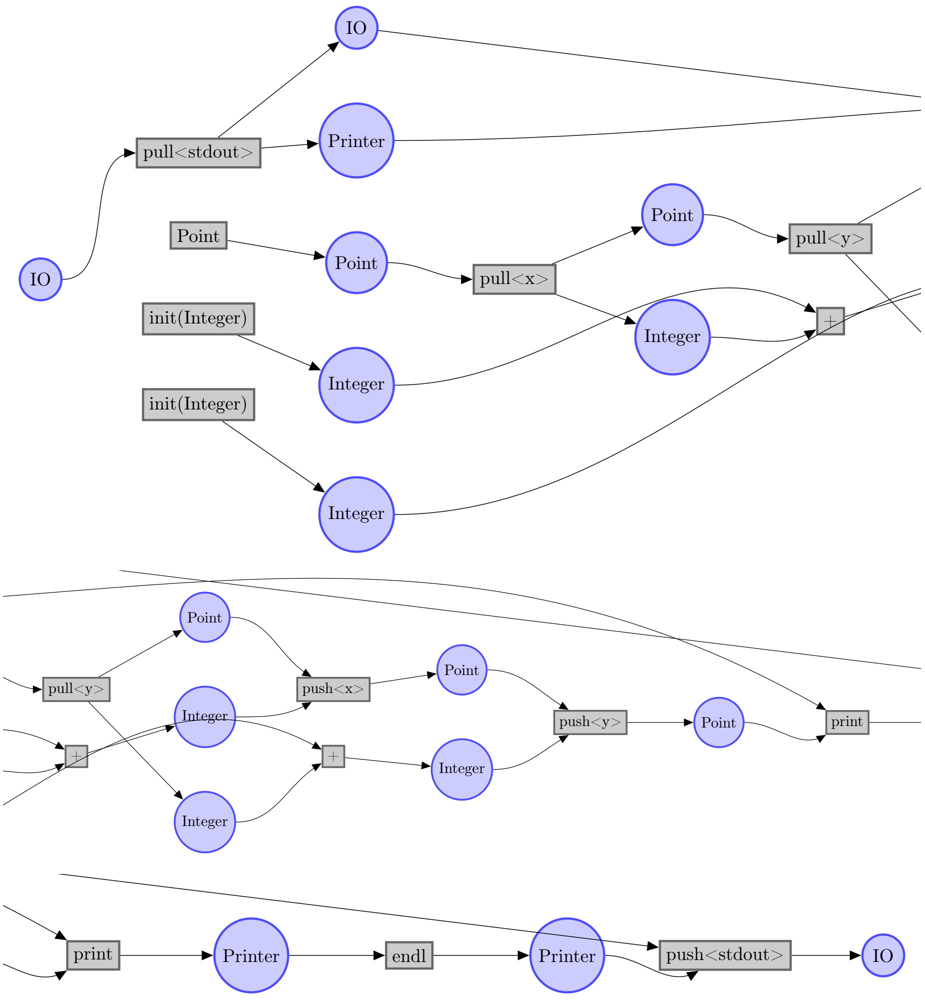


Figure 4.7: Graphical representation of test6's main function

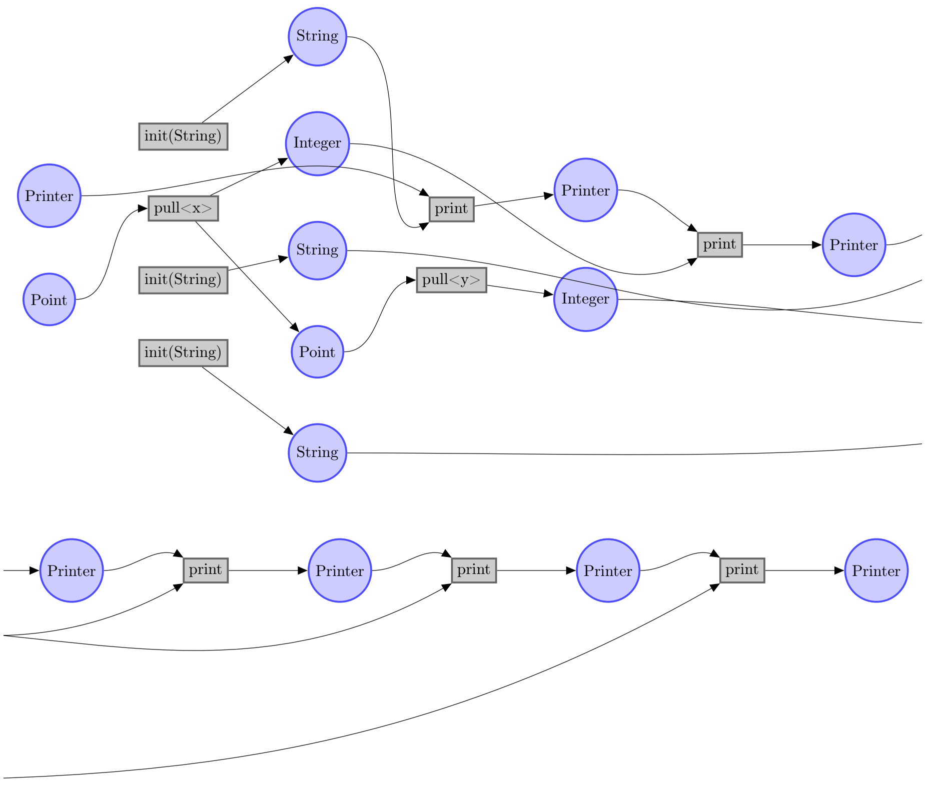


Figure 4.8: Graphical representation of test5's print function

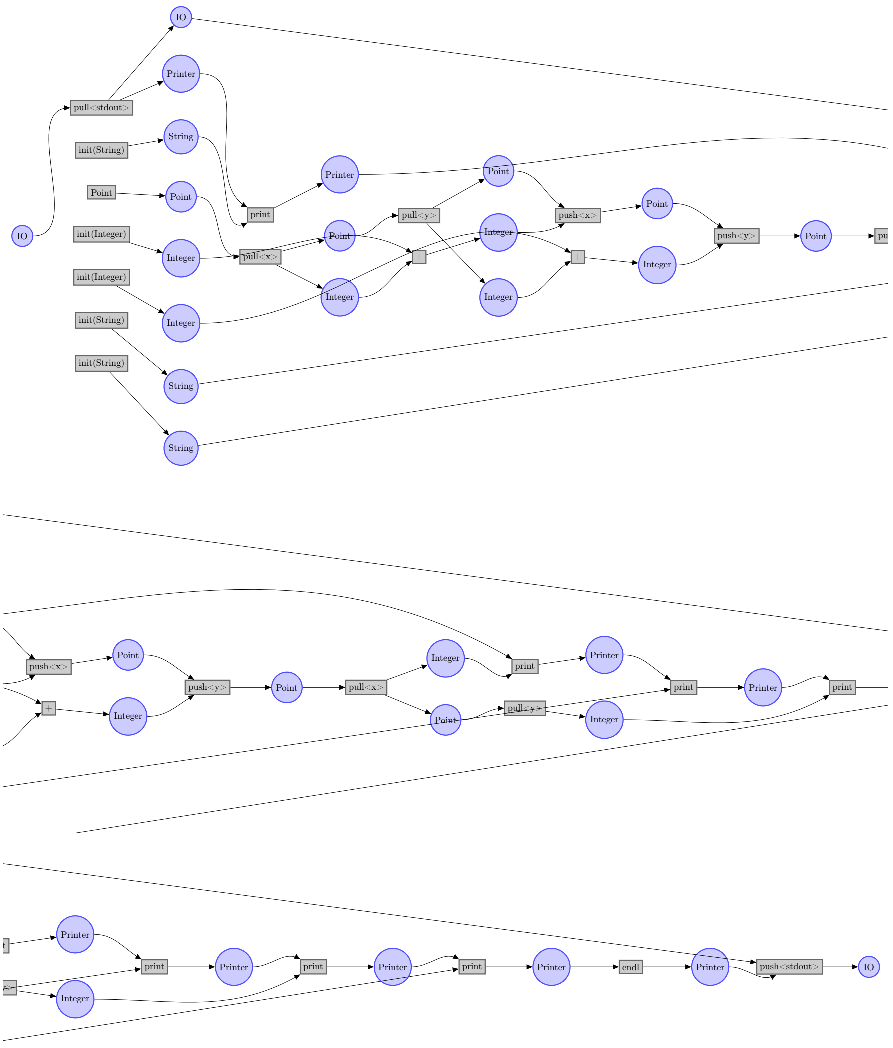


Figure 4.9: Graphical representation of test6's main function after the flow flattening

4.6.1 Results

If we look at the tree graphs we can see that the flattened graph is more parallel, that the first graph, because allot of the instantiation can be done in parallel. But when we look at the flow we could like that the structs could be unravelled so that the x and the y could be calculated in parallel, as they are newer actual needed by the same function. It could shorten the calculation by 4 pull/push operators thereby making the computation more parallel.

4.7 Experiment 7 - FlowSize and Multible List Mapping

Experiment 7 is a test of the mapping with multiple lists. This program therefore also test the compile-time checking of multiple lists and give an example on how to setup such list so they are compatible.

Listing 4.11: The `FlowCode` of test6

```
module test7;
import flow.math;
import flow.IO;

def [IO in]:main:[IO out] {
  [in]:pull<stdout>:[io,p];
  [10]:List<Integer>:[A];
  [A]:List<Integer>:[B];
  [A,B]:+:[C];
  [p,C]:printList:[p_1];
  [io,p_1]:push<stdout>:[out];
}
```

Just to walk though the program, first we create a list of size 10. Then we create a List of similar size. We now assume that there is something in these lists and then we add them field for field using the build in add function. And then we print It.

Currently is the printList not filled with correct c-code, as the actual c formulation of the list is not yet known.

The graph is actual not that big this time.

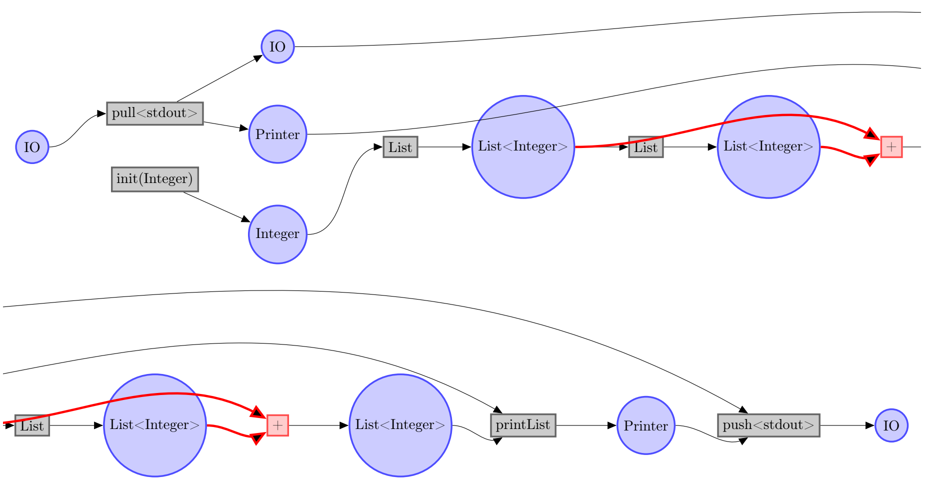


Figure 4.10: Graphical representation of test7

There is nothing new in the simple code but you can of cause inspect it in the appendix.

4.8 Results

The test shows that it is possible to statically check list sizes. Especially for this test is the graphical output a bit clumsy, which is something that would be needed to

Chapter 5

Discussion

In this chapter I will review the project. I will talk about what is done, what can be done and what this prototype means. I have subdivided the discussion acording to the layers of the **Flow** ; **Flow** , translator, and compiler.

5.1 Flow

In this section I will discuses the features of **Flow** .

5.1.1 Language

I chosen Java as the programing language when developing the prototype, because I knew that it would make the developing time shorter, but is it also the language that I recommend for the real representation of **Flow** ?

Java is safe to program in which will definitely reduce the maintenance of the **Flow** compiler. It is easy to develop from and to implement frameworks, which would make the lives of the compiler developers easier. It works on almost every platform, which makes it portable and it is one of the faster interpreted languages.

What I don't like about Java is that if we base **Flow** upon it, **Flow** will depend on development of Java. Besides, some developers might want to build their own **Flow** interpreters in other languages.

Because Portability is one of the big focuses in **Flow** , I think that the best way to go is to produce a simple and understandable binary intermediate language that can be translated to and read from by any platform.

But to build the translator and the compiler I would recommend using a language like Java because of its build-in type safety. In a final product a **Flow** framework, should be developed both for both Java and for C.

5.1.2 Tools and Optimizations

The visitor pattern works very well and have proven itself in the development. Many of the tools develop, even the larger compilers, only took a couple of hours to build.

The visitor pattern does have some disadvantages. First, its very rigid structure, which also is it strength, makes it hard to use advanced algorithms on the **Flow** . Secondly the transversal of the tree is done by function calls, which will give a unintended memory and performance issue when working with very huge programs.

The visitor is a great way to build fast typed algorithms for **Flow** , and I think that with a more iterative solution the visitor should be part of any framework developed to **Flow** . But when that is said I do not think that the Visitor should be the only way to access the internal parts of **Flow** , mostly because it reduces some of the options what you can do with it.

5.1.2.1 Possible Optimizations

Besides the Flow Flattener, I can think about a lot other optimizations. Each of these optimizations could be a project in them selves.

Operation Optimization simply evaluating as much as the code as possible before compiling. So if it wants to add two integers known at compile time we can evaluate the result and remove the operation form the flow.

Recall finding running though the recursive function and try to optimize the flow so that we can use a recall function instead of a recursive function call.

Map concatenation if multiple maps are running in serial, we could optimize the flow by concatenating them and thereby having them in the same map.

Advanced FlowSize checks if two list is instantiated with the same Integer, they should be evaluated to have the same size. But such a check would require an advanced approach.

and much more .

5.1.3 What misses?

This section is about the shortcomings of **Flow** .

5.1.3.1 Traceability

Traceability is when we can walk back thru the code and find the exact position of the problem, no matter which optimizations we have run on the **Flow** . Currently is the traceability of **Flow** is all most non existing. The traceability has been sacrificed to make **Flow** as independent from the former steps as possible.

A solution would be to add an class to the nodes, that can hold the trace information and help us to trace back errors to the source code. This class could be implemented by different translates to allow for customized tracings.

Optimizations could prove to be a big problem, as they could easily disturb the traceability.

Luckily is the need for traceability not as necessary in **Flow** as in other languages, because we can catch most of the problems at translation.

5.1.3.2 Running times

A nice feature to add would to be the ability to calculate the asymptotic and average running time of a program before its run. That would allow for better optimization and better compilations, because the optimizer and the compiler will know how and what to optimize.

One simple approach could use List as they is already given dynamic sizes at compile time. This is the same as associating the list size with a variable n. The when we map over the list we know that at least n computations will be made, if this then is nested in a map over an other list of length m, then the asymptotic running time would be $O(n \times m)$.

It is not that simple when we work with recursive functions that runs on numbers, as there is no way to ensure how many iterations there will be done, except if we fill in some key-functions where the user can hint what they want. An example would be in a case where we want to recourse over an integer where we always want to decrease it by one for each iteration. Then instead of using the minus operator a decrement operator could tell the optimizer that this integer is getting smaller for each step, and if we abort when it zero. We know that the asymptotic running time must be the integer. The solution could be transferred to similar cases like transversing lists, etc.

5.1.3.3 List, Sets, Dictionaries

In the design chapter I talked about the Set and Dictionaries, but I never actually implemented them, most of all because I did not have the time. The set and the dictionary is are a crucial part of a programing language and is needed in most algorithms.

The set is especially important as it is a part of the reduce pattern used in many parallel algorithms.

There are also some vital important features of the List that isn't implemented. A List should be able to tell in which indexing it wants to be accessed by a Map. This means if we want to calculate a[i] + a[i+1], we do not have to use the Dictionaries. And then there is the padding feature, which allows the program to map over two list guaranteed of the same size. It should be able to enable for different types of padding.

I think that keeping these three definitions of collections, is crucial to make an effective and secure mapping where the compiler is not supposed to guess.

5.1.4 Testing

The fact that functions do not have a state, does give some nice side effects besides that the code can be run in parallel. The most nameworthy is that the functions are completely predictable from their inputs. This means that a compiler could implement automatic dynamic programing on functions. Collection the outputs associated with the inputs. But more importantly is it possible to create unit test on the single function without thinking about the rest of the program.

There are two approaches to testing a flow, the first would be to list a number of cases of input and the the required outputs. And the the tester would run the function with all the inputs and test that the outputs are in equal to the requested outputs.

The second approach would be to set some requirements of the function, i.e. that a number should always be positive, with all inputs. Then the compiler should be able to determine if this is true by reverse engineering of the function. The checker could use the proven properties of subfunctions to prove new functions, and so on. I know this is an extensive functionality but on the other side, this could end up with software we can "prove" to run correctly.

5.2 Translator

This section is dedicated to the translation of **Flow** .

5.2.1 `FlowCode`

I have found that ANTLR is a perfect tool for parsing the code and the transition from `FlowCode` to **Flow** went, from a developers view, almost without problems.

5.2.1.1 Syntax

`FlowCode` turned out to be exactly what I think that a programing safe programing language should look like, as there is no doubt on what is done, and in which order things are executed. Sadly the effect of the explicit workflow, is that the language is also a bit more clumsy to work with and code can quickly become confusing, and spaghetti code might be an undesired side effect of allowing the programmer to put the code in a arbitrary order. This problem can be met by a good coding discipline and by divide the problem up in chunks and put them in smaller functions thereby reducing the number of lines per function.

Programing in `FlowCode` would require that we change the way we think about programming, which is something that I think the worlds need. But due to the cost of reeducating the current programmers there will probably go some years before a language like `FlowCode` can be used in actual production.

5.2.1.2 Modules

The coded module functionality is currently not as good as it could be. This is mainly due to a relative module loader. Sometimes a module is loaded twice, or more which is a waste of time. I have not uses so much time on this part, as it was not needed for the proof of the concept.

5.2.2 GUI based languages

The original idea was that **Flow** should be programmed with some kind of GUI graph tool. This would make the feel of the graph more present to the programmer and the idea of parallelism more real. This would also force the programmer away from the serial thinking of the linear coding system.

The GUI based language would be the preferred final solution to how to program **Flow** . Currently this kind of programming is used, but mostly in creative themed programming, like developing movies or composing music. This method is also used in event driven programming, as LEGO Mindstorms, where it helps keeping an overview in a maybe complex computation.

5.2.3 Imperative programing languages

One of the biggest challenges of the future is that we have a lot of code that do not run in parallel. Therefore there is a demand for a smart compiler that can compile serial code to parallel code. Often this is not possible because the programmer could be using tricks that makes the parallelization impossible.

If we want to translate a program to **Flow** we need to pull out the idea of the program instead of the code itself. First of all we would like to build up the control flow, like in **Flow** , this requires that we know if variables are pointing to the same value. Sometimes we can we find out, if this is the case by analyzing the code, but sometimes, especially with C, it can be almost impossible to determine the real value of the function before running it.

But the basic idea is to analyze what we think the programmer wants:

```
a = 10;
a += 10;
b = a;
c = a + b;
```

could be translated to **Flow** like this (represented in `FlowCode`).

```
[10]:[a];
[10,a]:+:[a_2];
[a_2]:[b];
[a_2,b]:+:[c];
```

And with some analysis we could shorten it down to something as simple as:

```
[40]:[c];
```

Even though it is simple in this cases it gets harder when pointers or objects is introduced. To transform a complete imperative language into **Flow** is more in the category of a masters project or a Ph.D. I do not even claim that it is possible, what I claim is that if it is possible to transform the language into **Flow** , it can compile to all the platforms that compiles **Flow** .

5.3 Compiler

This part is dedicated to discuss the compile layer of **Flow** .

5.3.1 Tikz

The Tikz graphical compiler is working without doubt, but sometimes the representation is a bit flawed. The way I draw the edges in the graphs tend to cross over the node, and sometimes to be placed so that they are hard to differentiate.

In the future I would like to have that pipes are drawn over multiple layers if they are used in another layer so that we also illustrate how long a pipe is allocated. This could really help the memory optimization process, because then the developer could se the exact effect of the optimization.

Tikz is a genius tool for creating graphs with LaTeX. I think that a distribution of **Flow** should be delivered with a graph compiler, to help developers checking their programs, but I don't think that Tikz is the way to go. This is due to multiple factors.

- The output of a Tikz compiler is static, which makes it almost useless as documentation.
- Tikz uses the LaTeX compiler which is really slow, and some what buggy.
- The output is in PDF, and if the compiler should be a web server we would like to use a format like SVG that is embedable in webpages.

5.3.2 Simple

The Simple compiler is build to test whether it is possible to transform **Flow** into code that could be run on a processor. Even though Simple is not close to contain all the features of **Flow** , it has shown some weaknesses and some strong points in **Flow** .

I did not expect that it was so difficult to skip over the non processing nodes, and I think that a special approach is needed to correctly handle the select. Memory management is on the other hand a walk in the park, it is easy to evaluate how much memory is needed and when to allocate and deallocate memory, but to do it efficiently we need to know the targets of the nodes. The recall function did also prove to be easy to implement,just a while loop.

5.3.3 Job finding

As I also described in the Proof of Concept and Implementation chapters, is the current Job finding algorithm not as good as we could request. Accurate Job finding is important if we want the programs to run fast and stable.

Since Job finding is an essential part of any compilations, a job finding algorithm should be provided within the framework of **Flow** , so that the compiler builders can focus on optimizing code to their specific platform.

5.3.4 Interpretation vs Compilation

Currently I have only worked with the idea that **Flow** needs to be compiled, but we could also interpret the code on the go. It can be as efficient to interpret the code as it can be to compile it in cases where there is are frequent access to the hardware. The reason is that interpreted code can get super-optimized to the platform while it is running, with a small overhead.

Interpreted code does also have the advantage that it can be run on a sever without a compile delay. This is crucial especially in the future where the internet connections are going to be faster, so it is actually profitable to run programs on one ore more servers instead of directly on the local computer.

Compiled code is of cause much of the way faster, and I think that most of the energy should be focused on developing a compiler to **Flow** , but there is no reason not to make both.

5.3.5 GPU

One of the fundamental ideas behind **Flow** is "write once, run everywhere", and of cause a modern language should be able to compile directly to a GPU.

The current structure of the GPUs is that they interpret some C-like code at runtime[1]. In that optic it is not long from running **Flow** natively, which could

[1]see CUDA and OpenCL

be interpreted in the same way as C-code. The main focus for GPU performance would be in places where simple operations are done many times. The map is actually made for this, the decision whether to calculated the map on the CPU or the GPU could be made at runtime.

CHAPTER 6

Conclusion

Based on the points in the discussion, I have come to the conclusion that flow based languages has a future. Flow based languages has the advantage that the code can be checked for most errors at compile time. It is especially important that it eliminates the risk of race conditions and deadlocks.

When we reach the multi core era, current programming languages will not suffice. To make the transition from the current languages to new parallel languages as easy as possible, it is important to start the development now. The approach to design the flow based language as an intermediate language, would make it a solution that could help not only in the transition but also after.

Overall, I am convinced, that after removing some of the rough edges, flow based programs could become a solution to the Multicore Dilemma.

Appendix A

FlowCode Specification

This was the first initial definition of `FlowCode` , and should be seen as a vision for were I want programing a language to end. Some of the notions, like sets and Dictionaries is not mentioned here.

This is the official specification of the `FlowCode` language. `FlowCode` is a data-driven parallel programming language, that is designed to make parallel computations easy and serial hard. `FlowCode` is build on the petri-net, and functional programing, and `FlowCode` is therefor a declarative programing language.

The basic design idea behind `FlowCode` is to part the computations up in two basic elements functions and pipes. Functions is the only way to read from, and write to pipes.

All code examples in this report is for illustrative purposes, and is not necessary working with the current version of `FlowCode` .

A.1 Basic Features

This section contains the the basic features, syntax.

A.1.1 Pipes and Functions

The simplest structures of the program language is the pipes and function. The way the functions is connected defines the program.

A.1.2 Functions

The functions is the working part of the program. A function is state less, this make them ideal for parallel programing. By making function state less there is a

A.1.3 Pipes

A pipe connects the functions and that way creates a program. When a pipe has connected two functions the output of the one function will be calculated as an input in the next. When using pipes there some simple rules:

Rule 3 *A pipe can not be the output of multiple functions, but can be the input to any number of functions.*

Rule 4 *A pipe can not be connecting a function output of one type to the input of an other type.*

A.1.4 Combining Function

When combining functions with pipes we create a program. First a simple example; a function connected to 2 input pipes and 2 output pipes would look like this:

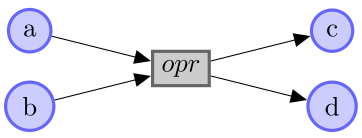


Figure A.1: A simple function example

This could be written like this in `FlowCode` syntax:

Listing A.1: `FlowCode` syntax

```
[a,b]:opr:[c,d];
```

Whats happening is that we associate the pipe a and b as input pipe and c and d as output pipes. This means that every time thats something is placed on the a or b pipe, *opr* gets to calculate on it.

The next example illustrates two functions connected to one pipe:

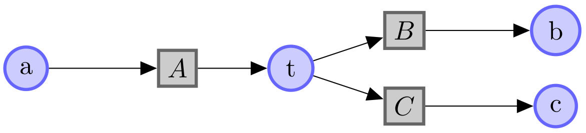


Figure A.2: Double output pipe

Which is coded like this:

Listing A.2: `FlowCode` syntax of the double output pipe

```
[a]:A:[t];
[t]:B:[b];
[t]:C:[c];
```

A.1.5 Defining Functions

Defining new functions is done using the `def` keyword, followed by input parameters, function name and output parameters. In and output parameters describes which pipes the functions can be connected to.

Listing A.3: `FlowCode` function definition

```
def [type_name b_1,...,type_name a_n]:f_name<template>:[type_name b_1,...,t
  /* Pipes and functions here. */
}
```

In this example `<type>` is a type like `Integer` or `Double` see more in types **section ??**. `<f_name>` is the name of the function, and an eventual template `template` parameter can be added, see **section A.2.2**

A.1.6 Empty function

It is possible just to move the contents of one pipe to an other pipe.

Listing A.4: The empty function

```
[any T a]:[any T b];
```

A.1.7 Empty Pipe

If the output of a function is not use we can put it on the empty pipe, noted as a _. In the next example we assume, that we need the output of the first parameter but don't care about the second.

Listing A.5: The empty pipe

```
[ ... ]:function:[a,_];
```

A.1.8 Multifunctions

If we want the functions to be run in serial, and of output of the function matches the input of the next function, we can remove the pipe and write it more compact as:

Listing A.6: Linked function

```
[ ... ]:function1:function2:...:functionN[...];
```

And this line can be intersected by any number of pipes:

Listing A.7: Linked function with pipes

```
[ ... ]:function1:[...]:function2:[...];
```

A.1.9 Types

Even though there is no variables in `FlowCode` there is a lot of types involved to make the code safer. By convention all types are in Pascal Case.

A.1.10 Standart Types

The standard types are looking like the types of C++, see Figure A.3

Type	C++
`Integer`	`int`
`Long`	`long`
`Byte`	`char`
`Float`	`float`
`Double`	`double`
`Boolean`	`bool`

Figure A.3: Standart types

These types has different properties meaning that they implement different interfaces.

Type	Interfaces
`Integer`	`Calculable`, `Comparable`, `Incrementable`, `Decrementable`
`Long`	`Calculable`, `Comparable`, `Incrementable`, `Decrementable`
`Byte`	`Calculable`, `Comparable`, `Incrementable`, `Decrementable`
`Float`	`Calculable`, `Comparable`
`Double`	`Calculable`, `Comparable`
`Boolean`	`Negateable`, `Comparable`

Figure A.4: Standart types, intefaces

The interfaces they are associated with explains witch functions works on the Type.

An other build-in type is the List type. This is the optimized version of storing list of information. The list is templated type, the template in this case declares what the internal type is.

A.1.11 Type defining

Sometimes when using interfaces, or to make the program more modular, it is needed to define a simple type. The syntax for that is simple and much like the syntax in C.

Listing A.8: type definition

```
type type_name oldtype_name;
```

An example would be that a String is a List of Byte elements.

```
type String List<Byte>;
```

This way we still use the String as a List<Byte>, but we don't have to write it each time, and we can write special functions that only works on Strings.

A.1.12 Interfaces

Interfaces is the type safety in FlowCode , making it possible for multiple types of using the same functions. This also ensures the modularity of functions. A interface is simply defined as such:

```
interface interface_name
  inherit {other,inter,faces}
  require {func, tions};
```

The interface construct has two names `inherit` and `require`, inherit describes the interfaces the interface owner also needs to uphold, and require references to the declared function needed to be defined by the types implementing the interface.

A interface needs followed up by a definition for the referenced functions declarations, using the interface. The syntax is to `declare` them, which means that it has no function body, and no need for pipe names.

```
declare [type if_name T, ...]:function_name<templates>:[type if_name T,...];
```

When using a interface in a function it is defined like this:

```
def [Man m, type Sitable a]:sitOn {...}
```

but if we want to return the Sitable object we'll need to inform that its the same, we do this by the using command.

```
using Sitable S {
def [Man m, S a]:sitOn:[S b] {...}

// other functions
}
```

This means that the function can't be run with a chair and get a sofa as an output.

A.1.13 Structs

Struct is a collection of types. When defined a Struct becomes a type , and is not different from the standard types.

Listing A.9: Struct definition

```
struct struct_name<template> (parent_type) is inter,faces {
  a : type_1,
  b : type_2
};
```

A struct can only have one parent, as multiple inherence is not allowed. When a structs has a parent it inherit all the internal types, interfaces and functions. When a struct `is` some interface, it promises to define all declared functions in the interface. The body of the struct is the elements in the struct, named so they can be referenced.

A.1.14 Push / Pull

To be able to access the information in the structs we add two commands, pull and push. Their primary job is to fetch the internal information of the struct.

```
[type_name]:pull<v1>:[type_of_v1, type_name];
[type_of_v1, type_name]:push<v1>:[type_name];
```

The compiler will check that a type can only be pull one variable once before it's pushed back on the struct.

A.1.15 Operators

The programming language supports some standard features. To keep the simplicity of the language, as few build in operations as possible will be supported. The following should be seen as an example of some implemented functions.

A.1.16 Unary

```
using Negateable T
declare [T a] : ! : [T d];

using Incrementable T
[T a] : incr : [T d];

using Decrementable T
[T a] : decr : [T d];
```

A.1.17 Binary

```
interface Comparable require {>,=,>=};
interface Calculable require {+,-,/,*};

using Comparable T {
declare [T a, T b] : >  : [Boolean c];
declare [T a, T b] : =  : [Boolean c];
declare [T a, T b] : >= : [Boolean c];
}
using Calculable T {
declare [T a, T b] : + : [T c];
declare [T a, T b] : - : [T c];
declare [T a, T b] : * : [T c];
declare [T a, T b] : / : [T c];
}
```

A.1.18 Select

To be able to make the program depend on the input we have added a feature called the select. A Branch takes an input and put it onto one of to pipes:

Listing A.10: The select

```
[Boolean logic, any T in_true, any T in_false]:select:[any T out]
```

If the boolean is true then the in_true pipe will be transferred to out, and else the in_false will be transferred to out. The power of this function is compiler can decide how much it want to calculate so if the logic is false then only the instructions needed to calculate the in_false is executed.

A.1.19 Main

As in most programing languages there is a main function, it looks like this:

Listing A.11: Main function

```
def [IO io_in]:main:[IO io_out] {
  /* do something */
  };
```

The IO in pipe and out pipe, is the place where the functions of the operating system is found. An example is the std-out, a file-manager, or the graphic card. The exact functions of the IO struct is not decided yet but it should be very modular.

A.2 Advanced Features

A.2.1 Mapping

Mapping is one of the more exiting features of `FlowCode` . When a function that only takes a single element of type A gets a list of A's then it calls the function for each element of the list, and places the output in an list of same size.

Its possible to do this with all of the input or only some of the input. If one or more of the inputs is not at list, then the function will see this as a constant and it will be distributed over the entire array.

If two lists is used on one function then the function will be called once for each element in the lists. The map is checked at compile time to ensure that the lists is of the same size.

To handle problems before they occur, It is not allowed both to have two functions with the same name and same number of arguments, where the one function holds a list of the type the other is defined with.

Listing A.12: Error because multiple ambiguous declared functions

```
def [Printer p, Integer a]:print:[Printer out] {...};
/* --------  Wrong --------- */
def [Printer p, List<Integer> a]:print:[Printer out] {...};
/* --------  Right --------- */
def [Printer p, List<Integer> a]:printList:[Printer out] {...};
```

A.2.1.1 Examples

Basic Lets look at +.

Listing A.13: + function

```
[Calculatable T a, Calculatable T b]:+:[Calculatable T c];
```

First basic example a contains 1 and b contains 2, this will result in that 3 is put on pipe c. But what if a contains an List of `[1,2,3]`, then the function will put `[1+2,2+2,3+2] = [3,4,5]` on c. But if b also contains a List `[5,6,7]`, then the result would be `[1+5,2+6,3+7] = [6,8,10]`.

If the two arrays were of different size, then the program would not be able to compile.

Advanced When working with Lists of list, a problem emerges. When using a function expecting a list of any types, the compiler always takes the outer type. But if we want to use function on the inner values how is that programed. Lets say that a is a list of list of integers and we want the first value of each sublist, then this is wrong:

Listing A.14: Wrong element access

```
[aList,1]:List.getElement:[bList];
```

As this only results in the first element in aList is put in bList. The correct way to do this is the following:

Listing A.15: Right element access

```
// Other place.
def [List<Integer> aList,Integer i]:auxGetElement:[bList]
  [aList,i]:List.getElement:[bList];

// In Function
[aList,1]:auxGetElement:[bList];
```

As aList is not a List of integers, but its a list of list of integers, it is possible to map over the elements in aList. Hopefully a better solution will be found in later versions of `FlowCode` .

A.2.2 Templates

Templates is a big part of `FlowCode` as it is one of the ways to make your programs and modules modular. Templates can be used in both functions and in types. In List the template is the what is filled in the List, in the other can also be used when functions need a special, hardcoded property.

An example of a struct template definition.

Listing A.16: Template example struct

```
struct Complex<type Calculable T> is Calculable {
  R : T,
  I : T
};
```

Allows to both use all different types as long that they are Calculable.

It's also possible to use templates with functions. Here we se an example where we create a list of type T, where we increment the value for per fields, with an integer of size S for each element.

Listing A.17: Template example function

```
def [Interger length]:createIncrList<type Calculable T,Integer S>:
  [List<T> out] {
  [0]:List.create<T>:[f_out];
  [[length]:derc]:createIncrList<T,S>:[list];
  [list,[lenght,S]:*]:List.addBack:[t_out];
  [[length,0]:>,t_out,f_out]:select:[out]:
}
```

This is of cause a constructed example, made to illustrate different points like select and recursion. In real life we will write it like this.

Listing A.18: Template example real situation

```
def [Interger length]:createIncrList<type Calculable T,Integer S>:
  [List<T> out] {
    [0,length]:List.range<T>:[templist];
    [S,templist]:*:[out];
}
```

A.2.3 Modules

You denote the name of the module in the top of each file. This is done like this:

Listing A.19: Module declaration

```
module module_name;
```

And it needs to be the first thing written in the file, and have the same name as the file. After the module declaration we can declare any imports.

```
import directory.subdirectory.module_name [as  other_name];
```

The imports declare the path to module, from the directory the compiler is run in, from the directory of the file, and from the libraries path. The syntax also enable the programer to choose a other name for the reference.

After the module declaration and the imports, the definitions of the functions, interfaces and structs appear.

To call a function from a module is simple and like many programing languages like this:

Listing A.20: Module usage

```

//in imports
import flow.list as List
import flow.algorithms

  //in function
    [100]:List.createRandom<Integer>:[list];
    // calling sort from algoritms
    [list]:sort:[sortList];
```

Appendix B

Simple Code

B.1 Test2

Listing B.1: Simple code from test2

```
flowFunction0 [String : arg0]
  init the String into arg0
end

flowFunction1 c-code [String : o]
  printf("%s",*((char**)o));
end

Main []

  on call {
    Allocate local space (2)
    run Job0
    run Job1
  }

  Job0 depend on ()
    allow Job2 to run
  end
  Job1 depend on ()
    call flowFunction0 with [Memory(2135760193)]
    allow Job2 to run
  end

  Job2 depend on (Job0, Job1)
    call flowFunction1 with [Memory(2135760193)]
    Allow Function to exit
  end

  on exit {
    Deallocate local space (2)
  }
end
```

B.2 Test3

Listing B.2: Simple code from test3

```
flowFunction0 [Integer : arg0]
  init the Integer into arg0
end

flowFunction1 [Integer : arg0]
  init the Integer into arg0
end

flowFunction2 c-code [Integer : a, Integer : b, Integer : o]
  *((int*) o) = * ((int*) a) + * ((int*) b);
end

flowFunction3 [Integer : arg0]
  init the Integer into arg0
end

flowFunction4 [Integer : arg0]
  init the Integer into arg0
end

flowFunction5 c-code [Integer : i]
  printf("%d",*((int*)i));
end

flowFunction6 c-code []
  printf("\n");
```

```
end

Main []

  on call {
    Allocate local space (7)
    run Job0
    run Job1
    run Job2
    run Job4
    run Job5
  }

  Job0 depend on ()
    allow Job8 to run
  end

  Job1 depend on ()
    call flowFunction0 with [Memory(1109376982)]
    allow Job3 to run
  end

  Job2 depend on ()
    call flowFunction1 with [Memory(845913220)]
    allow Job3 to run
  end

  Job3 depend on (Job2, Job1)
    call flowFunction2 with [Memory(1109376982),
        Memory(845913220), Memory(921745400)]
    allow Job7 to run
  end

  Job4 depend on ()
    call flowFunction3 with [Memory(1975391989)]
    allow Job6 to run
  end

  Job5 depend on ()
    call flowFunction4 with [Memory(315976503)]
    allow Job6 to run
  end

  Job6 depend on (Job4, Job5)
    call flowFunction2 with [Memory(1975391989),
        Memory(315976503), Memory(2037510537)]
    allow Job7 to run
  end

  Job7 depend on (Job6, Job3)
    call flowFunction2 with [Memory(921745400),
        Memory(2037510537), Memory(665360297)]
    allow Job8 to run
  end

  Job8 depend on (Job0, Job7)
    call flowFunction5 with [Memory(665360297)]
    call flowFunction6 with []
    Allow Function to exit
  end

  on exit {
    Deallocate local space (7)
  }
end
```

B.3 Test4

Listing B.3: Simple code from test4

```
flowFunction0 [String : arg0]
  init the String into arg0
end

flowFunction1 c-code [String : o]
  printf("%s",*((char**)o));
end

flowFunction2 c-code []
  printf("\n");
end

flowFunction3 [Integer : arg0]
  init the Integer into arg0
end

flowFunction4 c-code [Integer : a, Byte : o]
  *((char*) o) = (char) *((int*) a);
end

flowFunction5 c-code [Byte : a, Byte : b, Byte : o]
  *((char*) o) = * ((char*) a) + * ((char*) b);
```

```
end

flowFunction6 [Byte : arg0, Byte : arg1, Byte : arg2]

  on call {
    Allocate local space (3)
    Put arg0 in Memory(621631806)
    Put arg1 in Memory(257820787)
    run Job0
    run Job1
  }

  Job0 depend on ()
    allow Job2 to run
  end

  Job1 depend on ()
    allow Job2 to run
  end

  Job2 depend on (Job1, Job0)
    call flowFunction5 with [Memory(621631806),
        Memory(257820787), Memory(1719451110)]
    Allow Function to exit
  end

  on exit {
    Put Memory(1719451110) in arg2
    Deallocate local space (3)
  }
end

flowFunction7 [Integer : arg0]
  init the Integer into arg0
end

Main []

  on call {
    Allocate local space (8)
    run Job3
    run Job4
    run Job6
    run Job9
  }

  Job3 depend on ()
```

```
    allow Job5 to run
    allow Job12 to run
  end

  Job4 depend on ()
    call flowFunction0 with [Memory(1083228271)]
    allow Job5 to run
    allow Job7 to run
  end

  Job5 depend on (Job4, Job3)
    call flowFunction1 with [Memory(1083228271)]
    call flowFunction2 with []
    allow Job8 to run
  end

  Job6 depend on ()
    call flowFunction3 with [Memory(1598675078)]
    call flowFunction4 with [Memory(1598675078),
                  Memory(1764128329)]
    allow Job7 to run
  end

  Job7 depend on (Job6, Job4)
    Map from i : 0 to Memory(1545595021):
      call flowFunction6 with [Memory(1083228271)[i],
          Memory(1764128329), Memory(853323835)[i]]
    end
    allow Job8 to run
    allow Job10 to run
  end

  Job8 depend on (Job7, Job5)
    call flowFunction1 with [Memory(853323835)]
    call flowFunction2 with []
    allow Job11 to run
  end

  Job9 depend on ()
    call flowFunction7 with [Memory(917900179)]
    call flowFunction4 with [Memory(917900179),
                  Memory(1773272052)]
    allow Job10 to run
  end

  Job10 depend on (Job7, Job9)
    Map from i : 0 to Memory(1545595021):
      call flowFunction6 with [Memory(853323835)[i],
          Memory(1773272052), Memory(605324898)[i]]
    end
    allow Job11 to run
  end

  Job11 depend on (Job8, Job10)
    call flowFunction1 with [Memory(605324898)]
    call flowFunction2 with []
    allow Job12 to run
  end

  Job12 depend on (Job3, Job11)
    Allow Function to exit
  end

  on exit {
    Deallocate local space (8)
  }
end
```

B.4 Test5

Listing B.4: Simple code from test5

```
flowFunction0 [Integer : arg0]
  init the Integer into arg0
end

flowFunction1 [Integer : arg0]
  init the Integer into arg0
end

flowFunction2 c-code [Integer : a, Integer : b, Boolean : o]
  *((char*) o) = * ((char*) a) < * ((char*) b);
end

flowFunction3 [Integer : arg0, Integer : arg1]

  on call {
    Allocate local space (5)
    Put arg0 in Memory(880403204)
    Put arg1 in Memory(774321798)
    run Job0
    run Job1
  run Job3
  run Job5
}


on recall {
  run Job0
  run Job1
  run Job3
  run Job5
}

Job0 depend on ()
  allow Job2 to run
  allow Job4 to run
  allow Job6 to run
end

Job1 depend on ()
  allow Job2 to run
  allow Job7 to run
end

Job2 depend on (Job1, Job0)
  call flowFunction2 with [Memory(880403204),
      Memory(774321798), Memory(275558166)]
  if Memory(275558166) is true:
    allow Job7 to run
  else
    allow Job8 to run
  end
  Allow Function to exit
end

Job3 depend on ()
  allow Job4 to run
end

Job4 depend on (Job3, Job0)
  call null with [Memory(880403204)]
  allow Job7 to run
end

Job5 depend on ()
  call null with [Memory(1331353030)]
  allow Job6 to run
end
```

```
  Job6 depend on (Job5, Job0)
    call null with [Memory(880403204),
    Memory(1331353030), Memory(1418257117)]
    allow Job7 to run
  end

  Job7 depend on (Job1, Job6, Job4, Job2)
    Put Memory(1418257117) in Memory(880403204)
    Put Memory(774321798) in Memory(774321798)
    Recall
  end

  Job8 depend on (Job2)
  end

  on exit {
    Deallocate local space (5)
  }
end

Main []

  on call {
    Allocate local space (2)
    run Job9
    run Job10
    run Job11
  }

  Job9 depend on ()
    allow Job12 to run
  end

  Job10 depend on ()
    call flowFunction0 with [Memory(352697688)]
    allow Job12 to run
  end

  Job11 depend on ()
    call flowFunction1 with [Memory(735176496)]
    allow Job12 to run
  end

  Job12 depend on (Job10, Job9, Job11)
    call flowFunction3 with [Memory(352697688), Memory(735176496)]
    Allow Function to exit
  end

  on exit {
    Deallocate local space (2)
  }
end
```

B.5 Test6

Listing B.5: Simple code from test6

```
flowFunction0 [String : arg0]
  init the String into arg0
end

flowFunction1 c-code [String : o]
  printf("%s",*((char**)o));
end

flowFunction2 [Point : arg0]
  init the Point into arg0
end

flowFunction3 [Integer : arg0]
  init the Integer into arg0
end

flowFunction4 c-code [Integer : a, Integer : b, Integer : o]
  *((int*) o) = * ((int*) a) + * ((int*) b);
end

flowFunction5 [Integer : arg0]
  init the Integer into arg0
end

flowFunction6 c-code [Integer : i]
  printf("%d",*((int*)i));
end

flowFunction7 [String : arg0]
  init the String into arg0
end

flowFunction8 [String : arg0]
  init the String into arg0
```

```
end

flowFunction9 c-code []
  printf("\n");
end

Main []

  on call {
    Allocate local space (20)
    run Job0
    run Job1
    run Job3
    run Job4
    run Job6
    run Job10
    run Job12
  }

  Job0 depend on ()
    allow Job2 to run
    allow Job14 to run
  end

  Job1 depend on ()
    call flowFunction0 with [Memory(797130442)]
    allow Job2 to run
  end

  Job2 depend on (Job1, Job0)
    call flowFunction1 with [Memory(797130442)]
    allow Job9 to run
  end

  Job3 depend on ()
    call flowFunction2 with [Memory(1368348708)]
    allow Job5 to run
    allow Job7 to run
  end

  Job4 depend on ()
    call flowFunction3 with [Memory(434359633)]
    allow Job5 to run
  end

  Job5 depend on (Job3, Job4)
    call flowFunction4 with [Memory(434359633),
      Memory(985435678), Memory(823554482)]
  allow Job8 to run
end

Job6 depend on ()
  call flowFunction5 with [Memory(948074059)]
  allow Job7 to run
end

Job7 depend on (Job6, Job3)
  call flowFunction4 with [Memory(948074059),
      Memory(2099532520), Memory(1539259783)]
  allow Job8 to run
end

Job8 depend on (Job7, Job5)
  allow Job9 to run
end

Job9 depend on (Job2, Job8)
  call flowFunction6 with [Memory(1751161119)]
  allow Job11 to run
end

Job10 depend on ()
  call flowFunction7 with [Memory(501544898)]
  allow Job11 to run
end

Job11 depend on (Job9, Job10)
  call flowFunction1 with [Memory(501544898)]
  call flowFunction6 with [Memory(591786211)]
  allow Job13 to run
end

Job12 depend on ()
  call flowFunction8 with [Memory(1154079020)]
  allow Job13 to run
end

Job13 depend on (Job12, Job11)
  call flowFunction1 with [Memory(1154079020)]
  call flowFunction9 with []
  allow Job14 to run
end

Job14 depend on (Job13, Job0)
```

```
    Allow Function to exit
  end

  on exit {
    Deallocate local space (20)
  }
end
```

B.6 Test7

Listing B.6: Simple code from test7

```
flowFunction0 [Integer : arg0]
  init the Integer into arg0
end

flowFunction1 [Integer : arg0, List<Integer> : arg1]
  init the List<Integer> into arg1
end

flowFunction2 [List<Integer> : arg0, List<Integer> : arg1]
  init the List<Integer> into arg1
end

flowFunction3 c-code [Integer : a, Integer : b, Integer : o]
  *((int*) o) = * ((int*) a) + * ((int*) b);
end

flowFunction4 c-code [List<Integer> : list]
  for(int i = 0; i < (*((List*) list)).getLength(); i++){
    printf("%d",(*((List*) list)).get);
  }
end

Main []

  on call {
    Allocate local space (5)
    run Job0
    run Job1
  }

  Job0 depend on ()
    allow Job2 to run
  end
  Job1 depend on ()
    call flowFunction0 with [Memory(2064721795)]
    call flowFunction1 with [Memory(2064721795), Memory(590956692)]
    call flowFunction2 with [Memory(590956692), Memory(97255069)]
    Map from i : 0 to Memory(330889316):
      call flowFunction3 with [Memory(590956692)[i],
        Memory(97255069)[i], Memory(1421571929)[i]]
    end
    allow Job2 to run
  end

  Job2 depend on (Job1, Job0)
    call flowFunction4 with [Memory(1421571929)]
    Allow Function to exit
  end

  on exit {
    Deallocate local space (5)
  }
end
```

Appendix C

FlowCode Standard Library

C.1 IO

```
module flow.IO;
import math;
import string;
import list;
codeimport <stdio.h>;

type Printer signal atomic;

struct IO {
  stdout : Printer
};

codedef [Printer in, List<Integer> list]:printList:[Printer out]
@start
  for(int i = 0; i < (*((List*) list)).getLength(); i++){
    printf("%d",(*((List*) list)).get);
  }
@end

codedef [Printer in, String o]:print:[Printer out]
@start
```

```
  printf("%s",*((char**)o));
@end

codedef [Printer in, Integer i]:print:[Printer out]
@start
  printf("%d",*((int*)i));
@end

codedef [Printer in]:endl:[Printer out]
@start
  printf("\n");
@end
```

C.2 String

```
module flow.string;
codeimport <string.h>;
import math;
//import list as List;

type String as List<Byte>;

def [Byte char, Byte ofset]:ceasarsCode:[Byte code] {
  [char,ofset]:+:[code];
}
```

C.3 Math

```
module flow.math;

codedef [Integer a,Integer b]:+:[Integer o]
@start
  *((int*) o) = * ((int*) a) + * ((int*) b);
@end

codedef [Integer a,Integer b]:-:[Integer o]
@start
  *((int*) o) = * ((int*) a) - * ((int*) b);
@end

codedef [Integer a,Integer b]:*:[Integer o]
```

```
@start
  *((int*) o) = * ((int*) a) * * ((int*) b);
@end

codedef [Integer a,Integer b]:/:[Integer o]
@start
  *((int*) o) = * ((int*) a) / * ((int*) b);
@end

codedef [Integer a,Integer b]:&:[Integer o]
@start
  *((int*) o) = * ((int*) a) & * ((int*) b);
@end

codedef [Integer a,Integer b]:|:[Integer o]
@start
  *((int*) o) = * ((int*) a) | * ((int*) b);
@end

codedef [Byte a,Byte b]:+:[Byte o]
@start
  *((char*) o) = * ((char*) a) + * ((char*) b);
@end

codedef [Byte a,Byte b]:-:[Byte o]
@start
  *((char*) o) = * ((char*) a) - * ((char*) b);
@end

codedef [Byte a,Byte b]:*:[Byte o]
@start
  *((char*) o) = * ((char*) a) * * ((char*) b);
@end

codedef [Byte a,Byte b]:/:[Byte o]
@start
  *((char*) o) = * ((char*) a) / * ((char*) b);
@end

codedef [Byte a,Byte b]:&:[Byte o]
@start
  *((char*) o) = * ((char*) a) & * ((char*) b);
@end

codedef [Byte a,Byte b]:|:[Byte o]
@start
  *((char*) o) = * ((char*) a) | * ((char*) b);
```

```
@end

codedef [Integer a,Integer b]:lt:[Boolean o]
@start
  *((char*) o) = * ((char*) a) < * ((char*) b);
@end

codedef [Integer a,Integer b]:gt:[Boolean o]
@start
  *((char*) o) = * ((char*) a) > * ((char*) b);
@end

codedef [Integer a,Integer b]:gte:[Boolean o]
@start
  *((char*) o) = * ((char*) a) >= * ((char*) b);
@end

codedef [Integer a,Integer b]:lte:[Boolean o]
@start
  *((char*) o) = * ((char*) a) >= * ((char*) b);
@end

codedef [Integer a,Integer b]:=:[Boolean o]
@start
  *((char*) o) = * ((char*) a) == * ((char*) b);
@end

codedef [Integer a]:toByte:[Byte o]
@start
  *((char*) o) = (char) *((int*) a);
@end
```

Bibliography

[Hut11] Luke Hutchison. The flow programing language. `http://www.flowcode.net/`, jun 2011.

[JJK+11] Daniel R. Johnson, Matthew R. Johnson, John H. Kelm, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. Rigel: A 1,024-core single-chip accelerator architecture. *IEEE Micro*, 31:30–41, 2011.

[Met04] Steven John Metsker. *Design Patterns in C#*, chapter 29. Addison-Wesley, 2004.

[Res10] CITO Resarch. The multi-core dilemma. *IEEE Micro*, 2010.