Typescript compiler

Implementation of a native typescript compiler

Project document

Yau Chung Lam

Table of Contents

Part 1
Research and Analysis

1.1 Introduction

Typescript is a growing language that is slowly taking over JavaScript in various aspects including web development, back-end servers and mobile app development such as react-native. Typescript is a superset of JavaScript with a type system that inherits the nature of dynamicness. Typescript however, relies on JavaScript for runtime execution. As a matter of fact, Typescript must be lowered into JavaScript for runtime execution. JavaScript on the other hand, has dynamic typing and suffers from performance issues in computational heavy tasks. Most modern JavaScript engine implementation uses an interpreter/ Just in time compilation(JIT) blend to achieve bootstrip execution speed and performance. The engine will record types during execution and compile the source code just in time using these information. However, these compiled functions maybe discarded if the type reflection during runtime were mismatched. A single function may be recompiled several time in the whole execution process and because of the fast nature of JIT, these codes were left alone with not a lot of optimisation. The compiler also takes up significant amount of memory and CPU resources which is not suitable for lower end hardware.

This project aims to compile a set of Typescript ahead of time(AOT), providing performance and low memory footprint while using the same Typescript source code. With type information provided by Typescript and restricting dynamic features, it is possible to generate native machine code near C performance.

1.2 outlining the problem

For my project I am planning to create a compiler that is similar to compilers such as C, C++, Rust or Go. It will be a compiler that takes in source codes and generates machine codes and can target multiple platforms and hardware. The compiler would be able to accept the majority of the current Typescript specification. Further more, a runtime would also be created to assist the execution of generated machine code in a runtime environment.

For this to work, the compiler would require: a parser to parse source code into abstract syntax tree, a type checker to validate the source code, a machine code generator to generate machine codes. The runtime would require a memory management system to manage memory allocation, an exception handling system to handle user generated exceptions and a task scheduler to scheduler user events, callbacks and asynchronous tasks.

1.3 stake holder

The nature of a compiler has ruled out its stake holders. A compiler is a computer programme that translates code written in a programming language into another form of codes. In this case, the compiler is aimed to translate high-level Typescript source codes into low-level machine codes. The fact that it generates machine codes means that unlike conventional Typescript codes used in web pages and user applications, it would be used under native platform runtime environments.

Due to the factors mentioned above, the target audience would be developers that wants to use the Typescript language that develops native applications or server backend applications. They would most likely want something that has higher performance that conventional Typescript/JavaScript with better memory profile and does not posses any constrains on current Typescript source codes. It is suitable for the stakeholder’s needs because it provides the means to translate source code into machine code.

The stakeholders would be able to use the compiler through the command line once it is finished. The stakeholders can also alter its behaviour by using configuration files.

1.4 How the problem can be solved by computational methods

This problem is well suited for a computer as it can be solved using computational methods. This is due to the fact that compilers are designed to take in human written text, transforming it using computational methods and algorithms, while analysing the source code base on multiple conditions. It also generates binary information, machine codes that are specifically used by computers.

1.5 Analysis

1.5.1 Abstraction

I need to decide what to include in the compiler and what is necessary for the compiler to function. Abstraction is used to determine what is needed.

- Remove support for dynamic objects

- Add constrains on type rules to reduce solution complexity

- Remove unnecessary optimisations on the generation of source code

- Remove unnecessary commands on the command line interface

- Add the ability to read configuration files to give user the ability to communicate with the compiler statically without involving the command line interface.

- Add warnings when the compiler detects any suspected errors to indicate the user.

1.5.2 Thinking Ahead

Thinking ahead is important to know what we should be doing in the project. We will list out the desired features to allow us to plan for the design.

- I would plan to use an IR code in the compiler to process the source code

- Input source codes by user would likely be separated into module or files, I would plan to let the compiler have the ability to read local files, internet files and link modules.

- The output of the compiler will be an executable linked with the runtime.

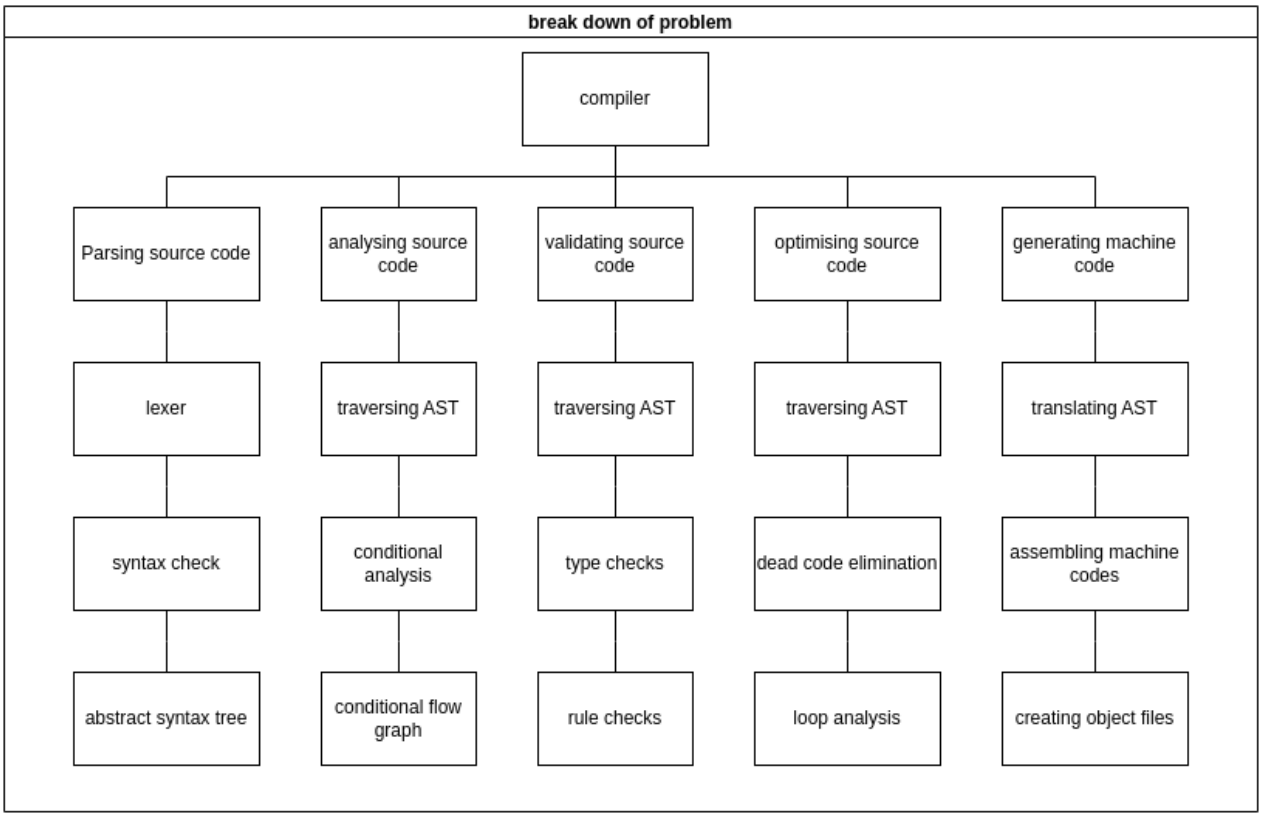
- Runtime should be provided as a static library linked against the executable so that user does not have to maintain a copy of the runtime as a shared library.

1.5.3 Thinking Procedurally

This gives the problem structure by breaking down the problem into smaller bits making it easier to work with. By thinking procedurally, I can work at the problem bit by bit making the process more efficient.

1.5.4 Decomposition

The system can be decomposed into smaller problems:



The compiler has been broken down into five main problems, by thinking procedurally, that should be solved in order for the compiler to have basic functionality.

Parsing source code

This will determine how the user input source code would be parsed and if local files and internet files can be imported as modules. This will convert the text source code into lexers. Syntax analysis will then be performed. It will check the lexers against predefined syntax rules and will return any errors if the lexer does not match syntax rule. It will then turn the lexers into an abstract syntax tree(AST) for later processing.

Analysing source code

analysing source code is necessary for later further operation for validating and optimising source code. It will analyse the source code by traversing the abstract syntax tree. The analysis will focus on conditional branching and should generate a control flow graph for later operations.

Validating source code

This will validate the source code to find any errors. It does this by traversing the abstract syntax tree. It will make sure that the source code is logically and syntactically correct so that the source code is valid and can be compiled.

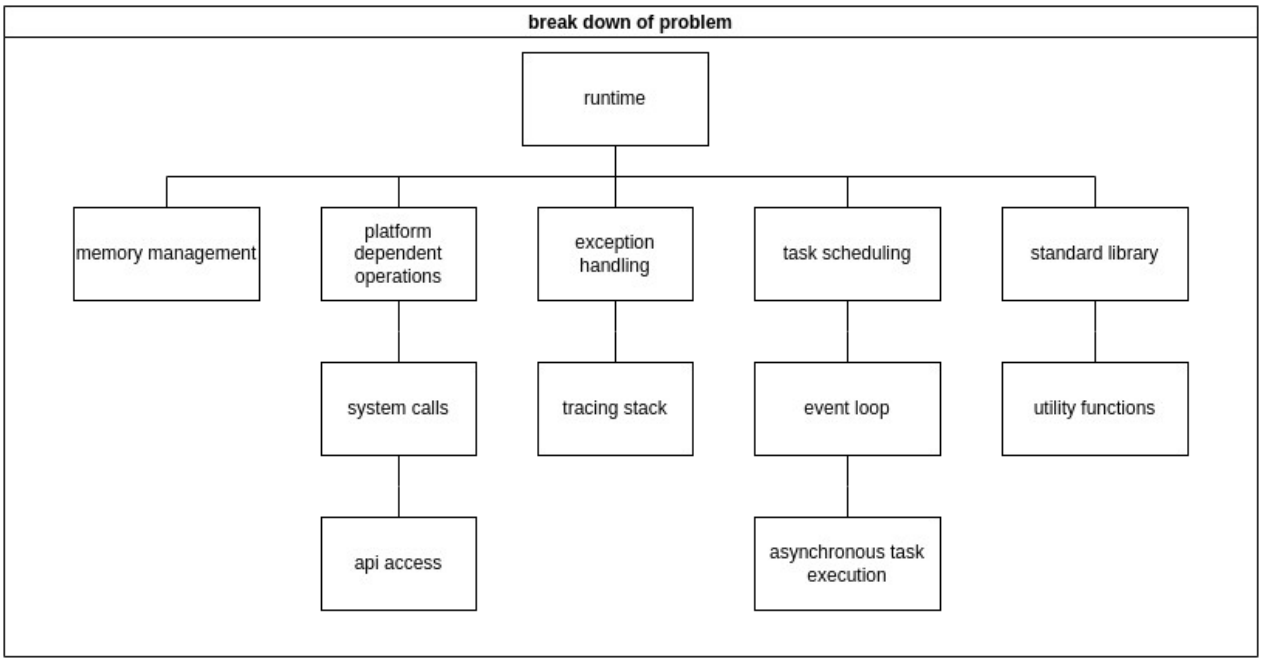
Optimising source code

This optimises the source codes. It is done by traversing the abstract syntax tree and using the control flow graph created during the analysis of source code. It will perform dead code elimination to reduce the code required to be converted to machine code and in turn reduce the output binary size. This will also perform loop analysis so that loop optimisation can be perform to boost code efficiency.

Generating machine code

This will generate machine code form the source code given. This is done at last of the compiler after parsing, analysis, validating and optimisation is done. This is done by traversing the abstract syntax tree. This will translate nodes in the tree into machine code by assembling machine codes together. This will also create object files and write the binary machine code in the object file using the correct format.

The runtime is decomposed into the smaller problems:



The runtime has been broken down into five main problems. These should be solved in order for the runtime to have basic function.

Memory management

This will manage memory allocation for user code. The memory strategy will determine how the machine code should be generated in the compiler. It will also affect the performance of the executable generated by the compiler.

Platform dependent operation

This will handle platform dependent operations from the user code. This includes system calls that are different on different operating systems and API access to system API. This will effectively hide platform dependent operation from the user so that the user does not have to worry about platform dependent source codes and compatibility. This will also allow the runtime to run on various different platforms.

Exception handling

This will handling exceptions during runtime for the user. This will allow user to throw out exceptions whenever they want. This will also allow user to create their own exception handler and the exception handler would be correctly performed when an exception is thrown no matter in what context.

Task scheduling

This will schedule tasks for the user code. Tasks spawned by the user must be handled and executed in runtime at some point. The scheduler will handle this for the user. In Typescript, asynchronous tasks may be spawned by the user. The scheduler will take in these tasks and determine at what point at the programme will it be executed bit by bit. The event loop will also be handled by the scheduler. User may create event handlers that may be triggered when certain events are emitted. The scheduler will determine when the handler is executed when an event is emitted.

Standard library

This will provide a standard library for the user code. The standard library would include utility functions that are useful for the programme but cannot be performed by user code. This include reflection where internal type information is retrieved. The standard library may also provide file system access, network access etc.

1.5.5 Pattern Recognition

The similarity among all the problems listed above is the need to traversal the abstract syntax tree. Traversal is required on every operation on the source code. We can therefore rule out the traversal algorithm. We can therefore implement a traversal algorithm once and use it on every module.

1.5.6 Thinking Logically

By thinking logically, I could determine what kind of logical solution can solve the problem.

When a syntax error is detected in the source code during parsing, the compiler should not exit but instead continue parsing the source code, so that all the syntax error can be detected and reported to the user. If a source code file is required but not found in the local file system, the compile should exit with an error. If a configuration file is not found in the user's directory, the compiler should continue with default configuration. If it fails when validating the source code, the compiler should exit with an error. If the runtime detects an exception but no handler was found, the runtime should exit immediately with the exception reported to the user.

1.5.7 thinking concurrently

By thinking concurrently, I can determine what part of the problem can be done at the same time to make the compiler more performant.

The source code from different source code files can be parsed and validated at the same time so that the overall compilation time would be reduced.

1.5.8 conclusion

The previous sections have demonstrated how my problem can be solved using computational methods making it suitable for a computer solution. These methods have helped structuring my problem making it easier to be solved.

Once the compiler is implemented in a computer system, the stakeholders would be able to compile Typescript source code into machine codes for their specific applications.

1.6 Current solutions: Compiler Architecture

The compiler architecture must be designed carefully as it is the core of a compiler. In this section, we will look at different architecture implemented in languages.

1.6.1 The Rust compiler

In Rust, the compiler is separated into four main stages: AST, HIR, MIR and Backend. The source code is firstly parsed into lexers and then Abstract Syntax Tree. It is then transformed into High-level IR which itself is also an implementation of AST (*1*). When lowering to HIR, all symbols and identifiers are represented by IDs and stored inside a map (1).

It is then converted into Typed HIR, where exhaustive type checks are performed. The reason why Mid-level IR is required is that specialised checks must be performed. Flow level sensitive checks is done and several transformations are applied to to Mid-level IR such as destructors and dynamic dispatches (2). This architecture allows more in depth source code analysis and better optimisation. However, compilation speed is greatly sacrificed in this implementation.

1.6.2 The Go compiler

The Golang compiler features compilation time. It is a stand alone compiler that does not rely on any other compiler back-end. The compilation process is performed in seven stages: Parsing, Type checking, IR construction, Middle end, Walk, Generic SSA and Generate Machine code (3).

In the Parsing stage, source code is tokenized (lexical analysis), parsed (syntax analysis), and a syntax tree is constructed for each source file. The syntax tree also includes position information which is used for error reporting and the creation of debugging information (3).

In the next stage of compilation, type checking is performed on the AST. The next step after type checking is to convert the syntax and type representations to IR and types. This process is referred to as "noding" (3). It uses a process called Unified IR. Unified IR is also involved in import/export of packages and inlining (3).

The IR then goes through the middle end where several passes are done to optimise source code. These includes dead-code elimination, de-virtualisation, function inlining and escape analysis (3).

The next stage of go compilation is the Walk stage where statements and expressions are decomposed into simpler statements, introduce temporary variables and respect order of evaluation. It desugars higher-level Go constructs into more primitive ones. For example, `switch` statements are turned into binary search or jump tables, and operations on maps and channels are replaced with runtime calls.

The IR representation is then converted into SSA form and function intrinsics are applied. Certain nodes are also lowered into simpler components during the AST to SSA conversion, so that the rest of the compiler can work with them. For instance, the copy builtin is replaced by memory moves, and range loops are rewritten into for loops. Then, a series of passes and rules are applied. These includes dead code elimination, removal of unneeded nil checks, and removal of unused branches.

The last stage of compilation is to generate machine code. The machine-dependent phase of the compiler begins with the “lower” pass, which rewrites generic values into their machine-specific variants. Once the SSA has been “lowered” and is more specific to the target architecture, the final code optimization passes are run. This includes yet another dead code elimination pass, moving values closer to their uses, the removal of local variables that are never read from, and register allocation.

1.7 Current solutions: Memory Management

Memory management strategy is a key part when implementing a compiler. It determines how the compiler should be implemented and the complexity of source code analysis. Here, we will look at two different Memory Management strategies: Garbage collector and Smart Pointer.

1.7.2 Garbage collection

Garbage collection is very common amongst object oriented languages such as JavaScript and Java. The garbage collector attempts to reclaim memory allocated by the program which is no longer referenced. There are several garbage collecting strategies including Tracing, Precise, Conservative, Generation and Copying.

In a Conservative GC, all possible memory locations are scanned during the garbage collector’s marking phase. It scans the program stack and registers to look for any values that may possibly be a reference to data. Some implementation of conservative GC implements markers for pointer to hint the garbage collector. One most known implementation is the Boehm-Demers-Weiser garbage collector. Most garbage collector implementation are conservative. This is due to the nature of scripting languages being dynamic and Just In Time.

In a Tracing GC, the user provides methods for the garbage collector to look for objects referenced by an object. This reduces marking time compare to Conservative scanning, however, the implementation of scanning is object specific and may result in a larger binary size.

In a Precise GC, the point of flow of execution is known at compile time. The GC simply looks at locations that are possibly references a data.

In a generational GC, the heap is split into two: a young heap and an old heap. The young heap is responsible for storing newly allocated objects. When an object survives a garbage collection, it is moved to the old heap. The young heap is more frequently collected to reclaim data.

A Copying GC splits the Heap into two with equal size. The objects that survived is moved to the other(shadow) heap and the shadow heap becomes the primary heap so that objects that did not survive will no longer be alive. This method has a huge memory overhead, its heap size is doubled.

A garbage collector often implements multiple strategies, combining their strengths and weaknesses.

1.7.1 Smart pointers

Smart pointers are pointers that stores meta-data. Smart pointer implementation relies on the compiler to analysis the source code and issue destruction of object when possible.

Some references cannot be determined at compile time when to be freed. These are usually solved using automatic reference counting. Reference counting stores a reference count about how many references were made to the object. The reference count is incremented when a reference is created for example assigning to a variable. It is decremented when no longer referenced. When the reference count hits zero, the object is destructed and released.

Other references are known at compile time and can be allocated on the stack so that heap allocation can be avoided to increase performance.

1.5.3 Comparing reference counting and garbage collection

Garbage collection has several advantages and disadvantages over smart pointers.

Garbage collectors allows a higher throughput of execution when compared to reference counting. This is because a reference counted pointer must be atomically incremented/decremented every time the pointer is referenced or dereferenced. This slows down the programme as a whole. Garbage collector introduces short pauses in the programme to activate a garbage collection. A concurrent garbage collector would perform most of its task out side of the main thread thus only disturbs the main programme when a garbage collection starts and ends. Compared to reference counting which is consistent throughout the whole execution, garbage collection is not as stable and pauses may vary on the state of execution.

Both reference counting and garbage collector introduces memory overhead for heap allocations. However, Garbage collectors tends to create more overhead then reference counters. This is because a reference counter only consist of a single reference count integer as its meta data while a garbage collector may require other types of metadata to accurately perform garbage collection such as the colour of pointer in a tri-colour garbage collector, destructor of object, remapping of memory etc.

ARC may encounter cyclic references. This means that two references references each other directly or indirectly. When this occurs, the memory will not be able to be freed even when it is no longer referenced. This is solved by the use of strong and weak references. Strong references keeps the object alive while weak references does not. This however introduces more overhead to the programme as it has to check for cycle referencing. Garbage collectors on the other hand does not have this problem.

1.8 Current solutions: Exception handling

Exception handling plays an important role in how the compiler is implemented. The specific exception handling method must be integrated into the compiler's backend. This will determine if further action is required to generated exception handling code in each stage of the compiler.

Below we will look at different ways to implement exception handling.

1.8.1 Handling exception in user code

Exceptions can be seen as a normal return value in user code. This implementation is used by Go and Rust to handle errors. Exceptions are thrown by returning execution from the current stack frame(returning from the function). The user code then checks for an exception and proceed to handle the exception however they like. In the case of Typescript, exceptions are not visible to the user, however we can generate exception handling codes along with the user code during compilation. This is by far the easiest implementation as it does not involve the runtime and is platform independent.

However, this implementation creates a significant overhead on the programme. Unlike Go and rust, we do not know when an exception will be raised therefore injecting exception checks whenever possible is needed. The performance penalty will offset any performance gained from static compilation and will further disprove the compiler for optimisation. Therefore this implement is the easiest however most costly that it is not practical to use. A different strategy most be investigated that may provide the same functionality with less overhead.

1.8.2 Exception handling: SJLJ (set jump/long jump)

The set jump/long jump exception handling method uses the c library set-jump and long-jump function to achieve exception throwing and handling. Upon creation of a new stack frame, the programme counter and registers are stored in a global frame list using set jump. When an exception unwinds, the frame list is used to determine which function needs processing by the runtime. (9)

The runtime returns to the function after unwinding using long jump to restore the registers and programme counter.

SJLJ exception handling builds and removes the unwind frame context at runtime. This results in faster exception handling at the expense of slower execution when no exceptions are thrown. However, exceptions are rarely thrown in an actual programme.

1.8.2 Exception handling: Itanium CXX ABI

The Itanium C++ ABI is an ABI for C++. It gives precise rules for implementing the Exception handling system, ensuring that separately-compiled parts of a program can successfully interoperate. Although it was initially developed for the Itanium architecture, it is not platform-specific and can be layered portably on top of an arbitrary C ABI. Accordingly, it is used as the standard C++ ABI for many major operating systems on all major architectures, and is implemented in many major C++ compilers, including GCC and Clang. (6)

An important concept in the ABI is the **landing pad.** It is a section of user code intended to catch, or otherwise clean up after an exception. It gains control from the exception runtime via the personality routine, and after doing the appropriate processing either merges into the normal user code or returns to the runtime by resuming or raising a new exception. (6)

The Itanium ABI defines a base API for the exceptions to be thrown and processed in runtime. The programme unwinds the stack when an exception is thrown. A landing pad is issued to each stack frame during execution. When unwinding the stack, the personality routine is used to determine if the stack frame wants to catch, process or to pass through the exception. The landing pad then decides if the programme is to be merged into normal execution, to resume the exception or to return to the runtime.

The unwinding process of the stack frames are separated into two stages. The unwinding begins with the raise of an exception specifically the _Unwind_RaiseException function defined in the API. The function is provided with an exception object allocated on the heap and an exception class which provides the type information required for personality routine to filter exceptions.

The **search phase** is the first stage of unwinding process. the framework repeatedly calls the personality routine, with the _UA_SARCH_PHASE flag, first for the current PC and register state, and then unwinding a frame to a new PC at each step, until the personality routine reports either success (a handler found in the queried frame) or failure (no handler) in all frames. It does not actually restore the unwound state, and the personality routine must access the state through the API.

The **clean up phase** is the second phase of the unwinding process. It is only executed when the search phase reports a failure. Again, it repeatedly calls the personality routine, with the _UA_CLEANUP_PHASE flag, first for the current PC and register state, and then unwinding a frame to a new PC at each step, until it gets to the frame with an identified handler. At that point, it restores the register state, and control is transferred to the user landing pad code.

1.9 Research: Compiler Backend

The compiler backend provides a middle layer between Intermediate Representations and machine codes. By using a compiler backend, a compiler frontend can port its software to targets which the compiler backend supports. It relieves the front end implementation from low level optimisations and target specific optimisations. Compiler frontends can therefore focus on compilation accuracy and source code optimisation. There are three possible options that we can look at for this project.

The LLVM compiler also known as Clang/LLVM is a compiler backend that is commonly used in static typed language implementations. It has a very wide range of supported targets and have very good optimisations. It uses its own IR representation to bridge the gap between frontend and backend. Language implementations are supposed to translate its languages into a target independent code. However, LLVM suffers from large compile times due to its complexity of analysis.

LLVM ships in a stand-alone shared library package independent of any dependencies. It is designed with portability and modularity in mind making it very easy to integrate and ship.

The GCC compiler is created by the GNU project. It has faster compile time compared to LLVM with machine code at similar performance. GCC does not provide a stand-alone library that allows frontend implementation. libgccjit must be used to implement a custom frontend. However, libgccjit is still in development and some features are not as well supported as to LLVM. Another factor to consider here is that GCC is distributed under the GPL licence. Meaning that if were to use GCC for my project, my project will have to also be in GPL or other compatible licences.

The Cranelift compiler backend is created to boost compilation time and portability for the rust compiler. It features a very fast compile time, small binary size and very portable since it is written entirely in Rust. Its generated machine code is significantly less performant then LLVM or GCC. It is considered as a backend for debugging code and JIT compilation. It is used by WasmTime and Wasmer engine to compile WebAssembly source codes.

I believe that LLVM is the more suitable option. It has good portability, good debugging support and generates high quality machine code. The only disadvantage it has is the compilation time which is a minor concern to this project. The goal of this project is to deliver a compiling system that inputs Typescript and outputs high performance machine codes. If the user requires fast compile time, they should be looking at current JavaScript engines.

1.10 Research: Dwarf debugging specification

DWARF is a debugging information file format used by many compilers and debuggers to support source level debugging. It addresses the requirements of a number of procedural languages, such as C, C++, and Fortran, and is designed to be extensible to other languages. DWARF is architecture independent and applicable to any processor or operating system. It is widely used on Unix, Linux and other operating systems, as well as in stand-alone environments.

It is initially designed for the Executable and Linkage Format (ELF) but at is later adopted in other object formats. It is the de-facto standard for storing debugging information on the majority of compiled languages.

The dwarf specification uses a Debugging Information Entry(DIE) to represent each element in the programme, for example variable, type, function etc. A DIE has a tag that represent the type of element it represent. It also have attributes with key-value pairs. The DIE can be nested forming a tree structure. A DIE attribute can refer to another DIE anywhere in the tree—for instance, a DIE representing a variable would have a DW_AT_type entry pointing to the DIE describing the variable's type.

The data required for dwarf debugging is split into two tables. The debug information table stores debugging information mentioned above in DIE format. The call frame information table stores information about call sites and frame structure allowing debuggers to locate and trace call frames on the call stack.

The dwarf debugging information should be included by the compiler. This is to ensure code generated by the compiler is debug-gable and is compatible with other main stream systems.

The dwarf information is also required to perform Itanium CXX ABI Exception handling. Itanium stack unwinding completely relies on the call site frame information in the dwarf table.

Implementing dwarf debugging into the compiler by ourself is not an easy task. However there is an easier path to integrate dwarf. LLVM supports the building of debugging information during compile time. LLVM will automatically generate debugging tables and place it into object files. We then only have to concern about interfacing LLVM instead of writing it our selves.

1.11 Research: File format for configuration files

The compiler relies on the configuration file to tell how it should compile the source code. The configuration file also known as manifest contains meta data that is needed to compile the project. There are 4 formats that are potential options. The XML format, the JSON format, the INI format and the TOML format.

1.11.1 XML

Extensible Markup Language (**XML**) is a markup language and file format for storing, transmitting, and reconstructing arbitrary data. The XML format is made up of tags, elements and attributes.

1.11.2 JSON

Json is a markup language that store data in object format. I personally think that it is not readable as a configuration file.

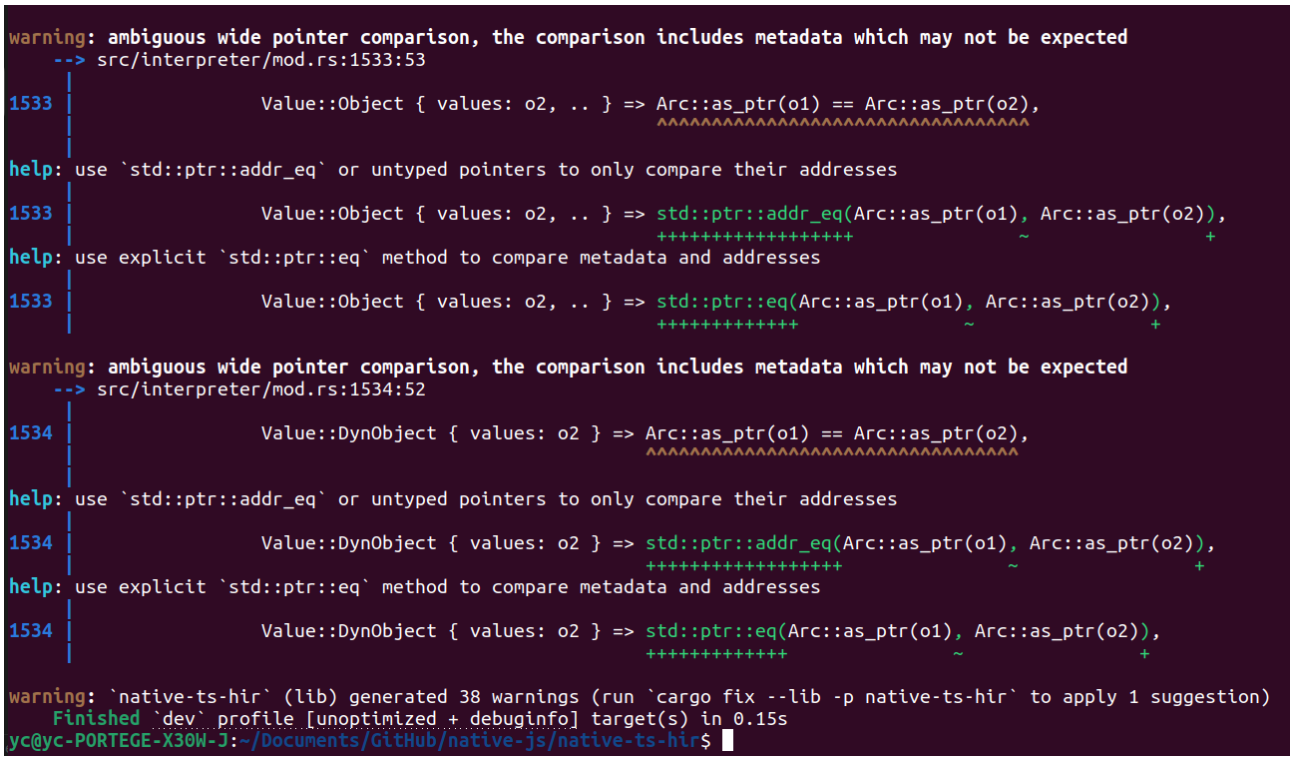
1.11.5 INI

1.11.4 TOML

Tom’s obvious markup language.

1.12 Compiler warnings

1.12.1 Rust compiler



Warnings emitted from the rust compiler on my project.

The warning first prints out the reason of the warning. In this case, it is an ambiguous wide pointer comparison.

It then points out the location of which the source code leads to the warning.

It then highlight the source code at which the warning is located and underlines it.

It also prints out helpful suggestions as which how it is going to be fixed.

Advantages:

- The warning is clear and readable

- The location of source code is clearly shown

- The suggestion given may be helpful.

Disadvantages:

- It is very dense

- Some suggestions may not be accurate.

1.12.2 C compiler

```
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub$ gcc main.c
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o: in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: error: ld returned 1 exit status
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub$
```

Error emitted by the c compiler.

The location of the file of the error is printed first. The name of the function at which it is located is then printed.

The binary location of which error originated is printed (.text+0x1b).

The reason for the error is printed which is “undefined reference to main”.

At last it prints out what causes the compiler to exit which is the error emitted.

Advantages

- The messages are straight forward

- The exact binary location is shown

Disadvantages

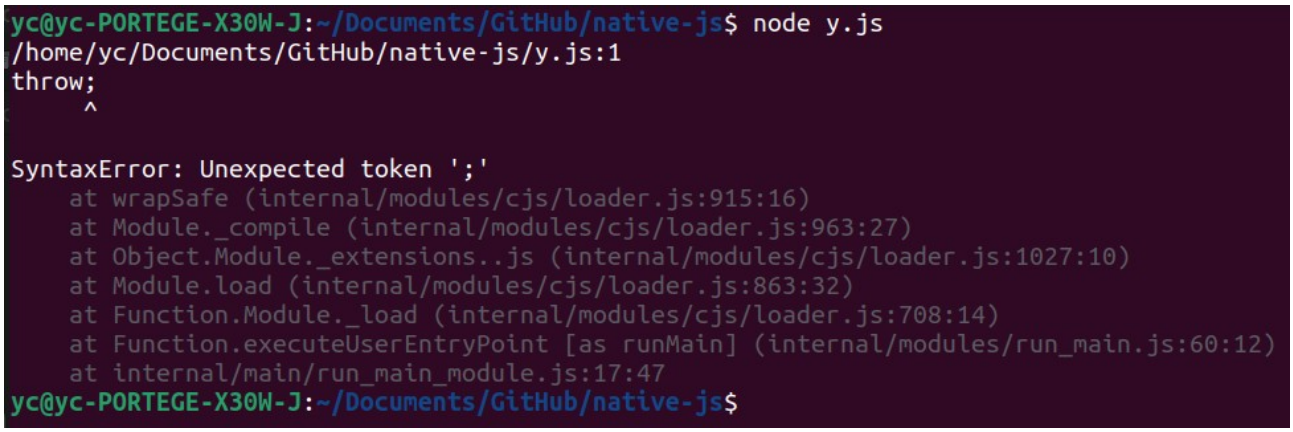
- The location of error is unclear, it is an internal object file

- The reason of error is unclear.

- No references to the source code of error

- No suggestions given

1.12.3 The v8 engine



error emitted from v8 engine

The location of error is first printed.

A reference of the source code at which causing error is printed.

The type of error and its reason is then printed.

A trace back of functions is then printed.

Advantages

- The location of error is clear

- The reference to source code is clearly shown

- The reason of error is clear

Disadvantages

- The trace back is confusing

- The trace back is showing internal functions

- No suggestion on how to fix the error

1.12.4 Potential solution for displaying warnings

With the above research on different warning systems, I think that the rust system is the most sophisticated.

I will implement a warning system that will first show the type of error and the reason of the error. I will then print out the source code location and the byte location of the warning.

I will then print the section source code that causes the error and underline the exact bits that causes the error.

I will then print a trace back of functions of where the error is located.

If the error is common, a possible fix can be suggested to the user to assist their development.

The error messages should be printed with syntax highlighting.

1.13 Feasibility of project

Modern compilers are complicated. In the past, compilers are hand-crafted to translate source codes into machine code. Several Compiler frameworks has made it easy to implement a front-end language compiler. The compiler back-end will handle all the optimisations and transformation to machine code from a framework specific Intermediate Representation code. This project will only have to concern about translating source code into Intermediate code and perform type checks.

Parsing source code may be a complicated process depending on the complication of Typescript syntax. It is not feasible to implement our own parser as the ECMA and Typescript specification is huge. We will be using a third-party library to do the parsing for us.

Type checking is the tricky part in this project. Types may not be known during parsing or translation. Instead, it must be performed after all parsing is done. Types may have generics which adds complexity to the process. It is however feasible to implement a type checker if generics are ignored.

By estimation, this project will be around 20~50k LOC(lines of code). Currently, I can produce averagely 600 lines of code every day during holidays and 200 during week days. So this project should take me around 12 months to finish.

1.13 Survey

1. Will you use this compiler to compile Typescript

- Yes
- Maybe

2. Do you think implementing a Typescript compiler would be beneficial

- Yes
- no, current JavaScript engine already fulfilled my needs

3. what do you consider a good compiler.

- A compiler that generates highly optimised code.
- I don’t know.

4. Do you think types are important

- Yes
- No

5. Do you think implementing this project in Rust is appropriate

- Yes
- What is Rust

6. Do you think using LLVM is appropriate for this project

- Yes, but other backend such as GCC may also be considered
- What is LLVM

1.14 Features for the solution

These are main proposed features for the compiler.

Display syntax errors to user	This allows user to debug their source code as they are using this compiler.
High-light and display the section of source code where the syntax error happens	This allows the user to easily locate the origin of source code where the syntax error happens. This helps developers to faster develop code.
Display the reason of error to the user when occurred	This allow user to better understand what kind of error has happened so that they can fix their source code.
Perform validation of source code without compiling into machine code	This allows users to debug their code without using much resources to create machine code. This is useful for user when writing source code.
Caching files from the internet	This reduces the use of network resources and speed up the parsing process. Recent files installed from the internet are cached and will not be downloaded again in a short period.
Emitting intermediate representation codes to user when requested	This allow users to better understand how their source code is processed and to debug their source code.
Presenting logs to user at each stage of the compilation process	This gives an indication to the user as where the compiler is up to so that user can estimate time required for the compiler to finish.

1.15 Software and hardware requirements

The software and hardware limitations are mostly limited by the runtime and LLVM. Below is a table of software requirements:

platform	supported	description
Linux	Yes	Linus platform has first class support for any sort of software development. LLVM can be install or deployed with package managers in Linus distributions.
Windows 10	Yes	Windows 10 platform is supported by LLVM. However, compiling LLVM on windows is a bit tricky. This will not affect the user as it will be distributed as a binary.
Windows 11	Yes	Windows 11 is the same platform as windows 10. It uses the same kernel, same libraries, same drivers, same everything except an updated user interface.
Windows 8	No*	windows 10 has made some changes in API since, making it inconsistent. It is not supported but it may work.
Macos	Yes	The Macos is based on the FreeBSD system and is a POSIX compliant system. The compiler utilises generic UNIX and POSIX system calls on all unix like systems.
FreeBSD	Yes	FreeBSD is POSIX compliant. LLVM also supports FreeBSD.
NetBSD	Yes	Same as above.
WebAssembly	No	LLVM does not support WASM
IOS	Yes*	Only aarch64 is supported
Android	Yes*	Only aarch64 is supported

Below is a table of hardware requirement:

architecture	support	description
x86_64	Yes	x86_64 has first class support
Amd64	Yes	Amd64 is fully compatible with x86_64
x86	Yes	x86 is supported
Aarch64	Yes	Aarch64 is supported
Arm	No*	User can cross compile code on arm devices. However, targetting arm devices with the compiler is not supported. Although LLVM supports Arm32, the custom unwind exception handling library in the runtime does not support arm. This is because Arm uses a custom ABI called EHABI for exception handling. It is not compatible with the Dwarf CFI extension. A separate implementation must be made, therefore it will not be supported for now. On IOS, SJLJ exception handling is used on arm devices. We will not support that.
i386	Yes	I386 is an early adoption of x86 architecture.
Riscv32	Yes	Both LLVM and unwind supports riscv, the Dwarf CFI is used for exception handling
Riscv64	Yes	Same as above

The recommanded RAM size is 4GB. That is because LLVM may take up around 1-2GB when programme is large.

1.16 success criteria

criteria	justification	reference
parse Typescript and ECMA standard up to 2023 edition	The compiler must be able to understand the source code in order to process it.	/
identify syntax errors and report to user during parsing	The compiler cannot process source codes that are not valid. The error is reported to the user for debugging purpose.	/
Identify dependencies of source code	If the source code is dependent on other source codes, it cannot function without those codes.	/
identify exported symbols of a module.	If the source code of a module exports symbols, this means that some other source code depends on it.	/
Able to reference and link modules	If the source code module has dependencies or is dependent on another module, they must be linked together to function.	/
normalise generic representations within AST.	Generic types should be normalised before further processed. This is because they are not actual types and cannot be represented by machine code.	Rust compiler
translate AST into HIR.	The high level intermediate representation is used to simplify the expression of source code so that further process can be done easier.	Rust compiler
perform type checks in HIR	Type checks are used to validate the legitimacy of the source code. Invalid operations cannot be convert to machine code.	Rust compiler
Perform automatic type conversions in HIR.	Machine codes does not recognise types. When different types are mixed together, they must be converted to another type for operations to be valid.	V8 engine
translate HIR into MIR.	The MIR is used so that source code can be represented in SSA format.	Go compiler

represent specialised types in MIR.	Some layout of types cannot be known before further analysis is done for example asynchronous tasks and generators.	Rust compiler
construct virtual tables in MIR.	Values may be stored as interfaces during runtime. Virtual tables are used to store abstracted information of an object.	Go compiler
integrate memory management strategy in MIR.	The memory strategy used by the runtime should be integrated into MIR. The operations must be compatible with the runtime.	/
decompose async operations into lower level operations.	Asynchronous operation cannot be represented by machine code. They must be translated into operations that have the same behaviour.	Rust compiler
decompose generator operations into lower level operations.	Generators cannot be represented by machine code. They must be translated into operations that have the same behaviour.	Rust compiler
translate MIR into LLVM IR.	MIR is translated to LLVM IR so that we can use LLVM to compile our source code.	Rust compiler
compile LLVM IR into targeted machine code.	Machine code must be generated as this is our project’s goal.	Rust compiler
link object files into executables.	Object files cannot be executed. They must be linked into an executable so that the user can execute it.	C compiler
link runtime to object files.	The runtime must be linked to the object files so that the executable contains the runtime.	Go compiler
perform garbage collection during runtime.	This is to recycle memory that is not used by the user during execution of programme so that memory would not be used up when executing the programme.	Go runtime
provide built in functions in runtime.	This allows the user code to execute normally	Go runtime

perform type reflections during runtime.	This allow users to debug or to perform dynamic operations on static types during execution of programme	Go runtime
handle exceptions during runtime.	This allows users to handle exception so that the programme will not crash when an exception happens	C++ runtime

Part 2
Design and Architecture

2.1 Overview

This project aims to implement a compiler for a subset of Typescript. In order to achieve this, the compiler must be implemented with careful design. A runtime library must also be implemented to provide intrinsic and utility functions. The compiler is separated into three main stages: Parsing, HIR lowering, MIR lowering and Machine code generation.

In the Parsing stage, the source code is parsed into AST(abstract syntax tree), its dependencies are analysed and imported and parsed as well. In this dependency implementation, we will follow the Go's philosophy: to disallow cyclic dependencies. Cyclic dependencies increases complexity due to the nature of Typed languages where type definitions may be declared in different modules and must be known before analysing its dependent. This problem is not encountered by JavaScript because it is a dynamic typed language.

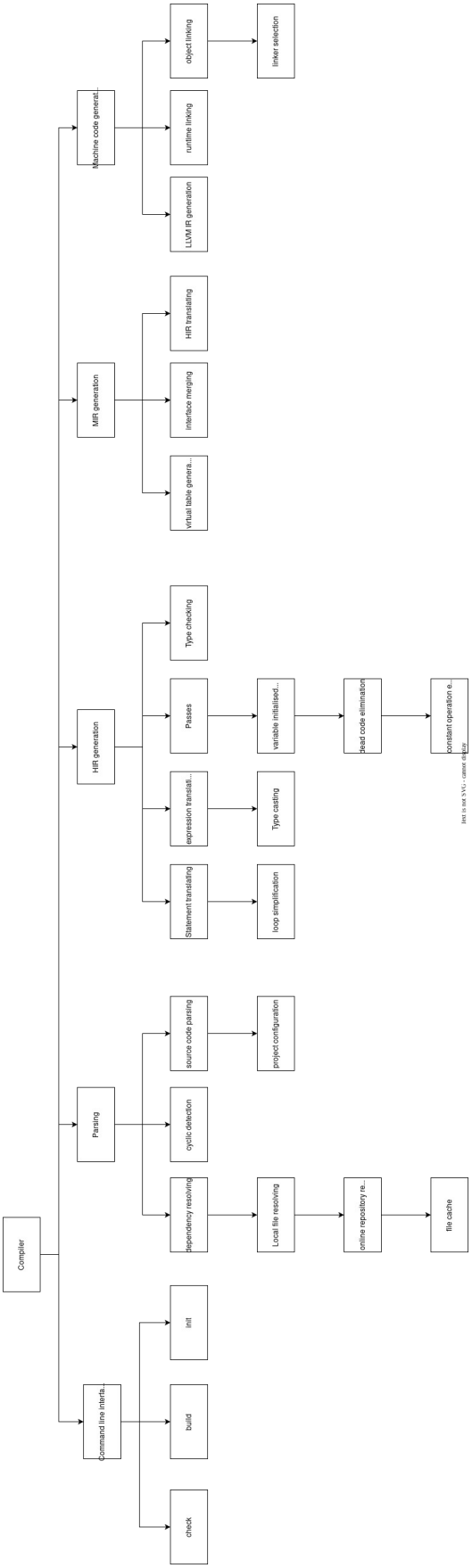
In the HIR generation stage, the AST is transformed into HIR. During this process, type checking is done. Some statements such as for, while loops are simplified into a single loop statement. Type conversion operations are also inserted into HIR after type checking.

In the MIR generation stage, the HIR is converted to MIR. MIR is a SSA form intermediate representation that represents source code with more simplified operations. Its main purpose is to perform scope base analysis, interface management and relieve the compiler from translating complex operations such as Async await, Generator yielding, Interface conversion etc. directly from HIR to machine code.

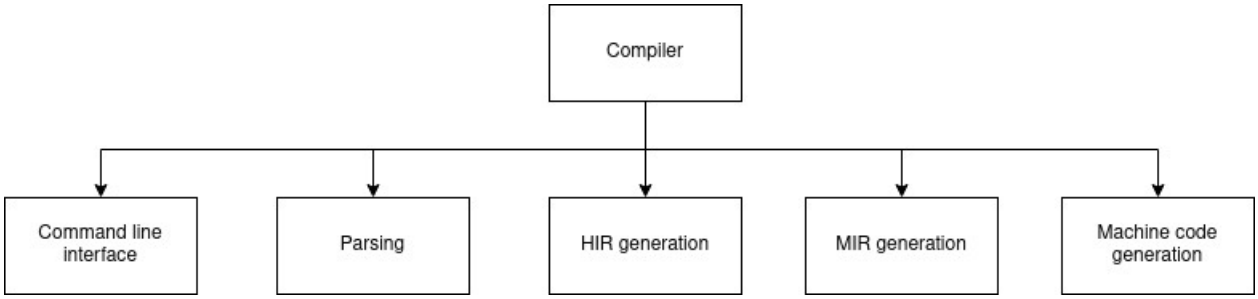
In the machine code generation stage, the MIR is translated into the LLVM IR. LLVM will then perform optimisations and generate object files.

This project will also implement a runtime library. The runtime is responsible for providing exception handling, garbage collection and other utility functions.

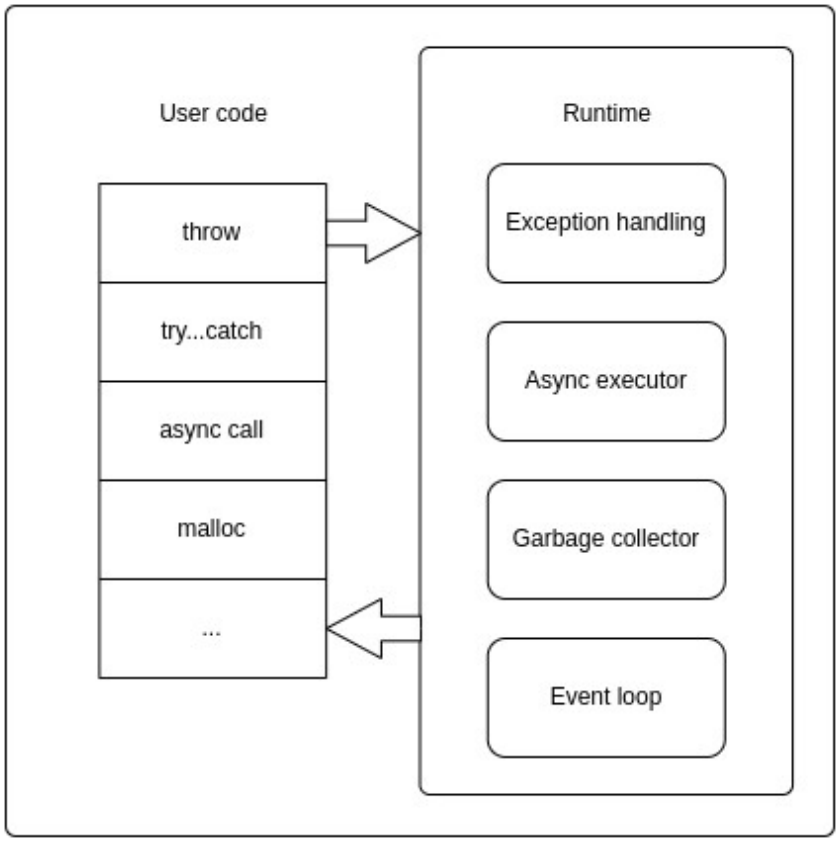
2.3 System Diagram



The compiler is separated into 5 sections. The Command line interface, the parsing stage, HIR generation, MIR generation and Machine code lowering.



A runtime is responsible for supporting the generated machine code at runtime.



2.3.1 Explanation of modules

Command line interface

This provides an entry point for the user to use the compiler. It is the interface where the user tells the compiler what to perform.

Command line check command

This command can be issued to the compiler by the user to indicate that the user wants to perform validation of the source code without translating it into machine code

Command line build command

This command can be issued to the compiler by the user to indicate that the user wants to compile the source code into machine code.

Command line init command

This command can be issue to the compiler by the user to generate a configuration file with default configurations.

Parsing

This is the module where source code is parsed and syntax analysis is performed. The source code is inputted into the parser and syntax analysis is performed to find syntax errors. If syntax errors were found, the compiler will report these errors to the user and exit. An abstract syntax tree is constructed from the source code if no errors were found.

Parsing: dependency resolving

This is where dependencies were found from the parsed source code. Dependencies are then imported and parsed into abstract syntax tree.

Parsing: Local file resolving

This is when a dependency is a local file, the local file is read and passed to the parser to be parsed into abstract syntax tree.

Parsing: Online dependency resolving

This is when a dependency is an online file. The file is downloaded from the internet and stored on a local file. This file is read and passed to the parser to be parsed into abstract syntax tree.

Parsing: Cyclic dependency detection

This is when all dependencies are parsed. It detects cyclic dependencies in the dependency graph.

Parsing: configuration file parsing

This is where the configuration file is parsed. It is parsed into a specific structure so that it can be read by the compiler.

HIR generation

This module generates HIR from abstract syntax tree. It also performs various operations to simplify the expressions and performs checking.

HIR generation: translating statements

This is where Typescript statement nodes in the AST is translated into HIR. Some simplification is done in the process, for example all kinds of loop are simplified into a simple loop statement.

HIR generation: translating expressions

This is where Typescript expression nodes in the AST is translated into HIR expressions. In the process, some rule checks are performed on the expression. Some combination of expressions are not allowed and will return an error.

HIR generation: Type casting

This is where the compiler detects a different type in the result of expressions, the compiler will automatically inject type conversion codes in HIR.

HIR generation: type checking

This is where type checking is done on the code. Invalid types cannot perform certain operations and this will result in an error.

HIR generation: check variable initialised before read

This is where the variables are checked for initialised before they are read. This is not allowed however valid expression. We have perform this check post translation.

HIR generation: constant operation evaluation

This is where expressions that can be computed at compile time are computed. This means that these user codes are not translated by the compiler into machine code be instead pre-calculated and placed as a constant

MIR generation

This is the module that translates high-level intermediate representation into mid-level intermediate representation.

MIR generation: dead code elimination

This is where dead code elimination is performed. The MIR is in SSA format meaning that it is easier to perform dead code analysis. This reduces code needed to be converted into LLVM IR.

LLVM IR generation

This is the module that translates MIR into LLVM IR

Machine code generation

This is where LLVM IR is converted into machine codes. The machine codes are then emitted to an object file.

Object file linking

This is where the object files links together with the runtime library. An executable is created.

2.4 Usability features

The compiler will offer several usability features.

The command line interface is designed to be easily understand and make use of the compiler.

The command line interface features three commands: check, build and init. While check and the init command is not essential to the compiler, they provides an easier way for users to develop their code. The command line interface also features a help page that displays the description of the commands.

The check command helper

```
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$ ./compiler check --help
Analyze the current package and report errors, but don't build object files

Usage: compiler check [OPTIONS]

Options:
  -e, --exclude <EXCLUDE>  Exclude a file from check
      --lib                check only the libraries
  -h, --help               Print help
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$
```

All the flags are listed here

Description of flags are listed here

The build command helper

```
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$ ./compiler build --help
Compile the current package

Usage: compiler build [OPTIONS]

Options:
      --features <FEATURES>  the feature to turn on
  -h, --help                 Print help
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$
```

All the flags are listed here

Description of flags are listed here

The init command helper

```
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$ ./compiler init --help
Create a new typescipt project in an existing directory

Usage: compiler init [OPTIONS]

Options:
      --lib                      place the configuration in library mode
      --version <VERSION>        The project is bounded to a specific version of the compiler
      --ts-version <TS_VERSION>  The project is bounded to a specific version of Typescript
  -h, --help                     Print help
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$
```

All the flags are listed here

Description of flags are listed here

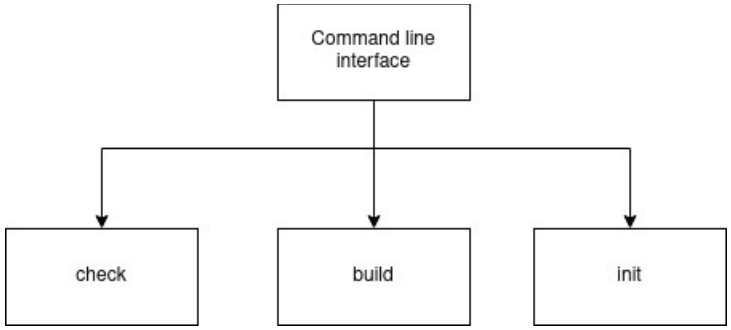
The following is the complete view of the usage helper of the command line interface.

```
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$ ./compiler
Usage: compiler [OPTIONS] --jobs <JOBS> <COMMAND>

Commands:
  check  Analyze the current package and report errors, but don't build object files
  build  Compile the current package
  init   Create a new typescipt project in an existing directory
  help   Print this message or the help of the given subcommand(s)

Options:
  -q, --quiet            Do not print log messages
  -j, --jobs <JOBS>      maximum number of parallel jobs, defaults to # of CPUs
  -r, --release          check in release mode
      --target <TARGET>  specifies the build target
      --offline          check without
  -h, --help             Print help
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$ ./compiler check --help
Analyze the current package and report errors, but don't build object files

Usage: compiler check [OPTIONS]

Options:
  -e, --exclude <EXCLUDE>  Exclude a file from check
      --lib                check only the libraries
  -h, --help               Print help
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$ ./compiler build --help
Compile the current package

Usage: compiler build [OPTIONS]

Options:
      --features <FEATURES>  the feature to turn on
  -h, --help                 Print help
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$ ./compiler init --help
Create a new typescipt project in an existing directory

Usage: compiler init [OPTIONS]

Options:
      --lib                      place the configuration in library mode
      --version <VERSION>        The project is bounded to a specific version of the compiler
      --ts-version <TS_VERSION>  The project is bounded to a specific version of Typescript
  -h, --help                     Print help
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$
```

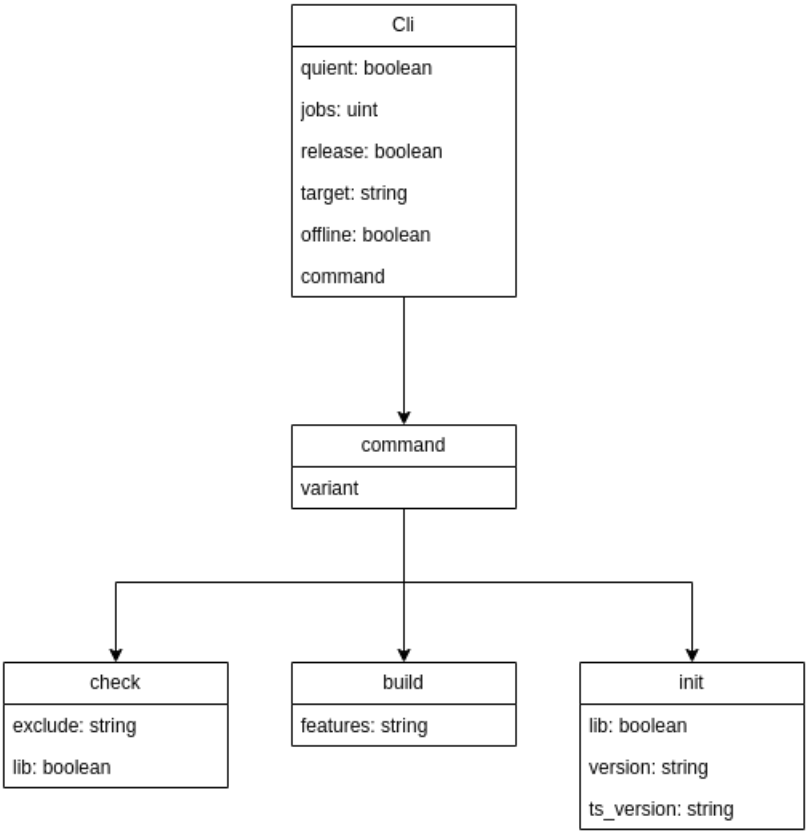
Usability feature	Justification
Command line interface	A command line interface
Prints the usage format of the command line interface	Tells the user how to use the command line interface
Lists all the possible commands	Tells the user what commands they can use.
Display description for each command	Tells the user the function of each command
Display optional flags for the command line interface	Tells the user what options they have
List all the optional flags of the command line interface	Tells the user what flags they can add to the command line interface
Display the short name and long name of each flag	Tells the user how they can add flags to cli
Display whether a flag requires an argument	Tells the user if they should provide an argument for the flag
Display description for each flag in command line interface.	Tells the user about the purpose of the flags
Display helper for command when the help flag is set	Allow the user to find detail description of each command
Display description of command in command helper	Tells the user the functionality of the command
Display command usage	Tells the user how the command should be used
Display optional flags for the commands	Tells the user what flags can be added to their command
Display short name and long name for flags in command helper	Tells the user how to add flags to command
Display description for flags in command helper	Tells the user what the flag does to the command

2.4 Command line interface

The command line interface provides an entry point for users to use the compiler. This interface is designed to have three main commands.



The data structure of command line arguments:

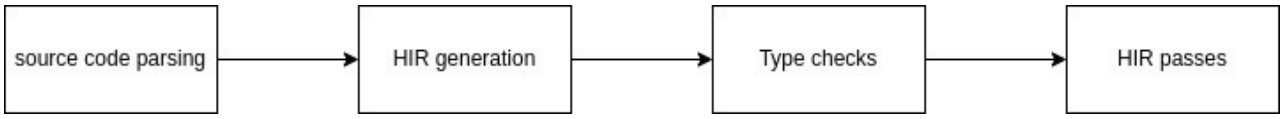


2.4.1 The check command

The check command will have the following options:

Short name	Long name	description
q	quiet	Do not print log messages
h	help	Print help messages and exit
/	exclude	Exclude a file from check
/	lib	Check only the libraries
j	jobs	Number of parallel jobs, defaults to # of CPUs.
r	release	Check in release mode
/	target	Specifies the build target
/	offline	Runs without accessing network

The check command have the following process flow:



The check command parses the source code and generates HIR. Several check passes are performed. At this point, all source code errors are known and the programme exits.

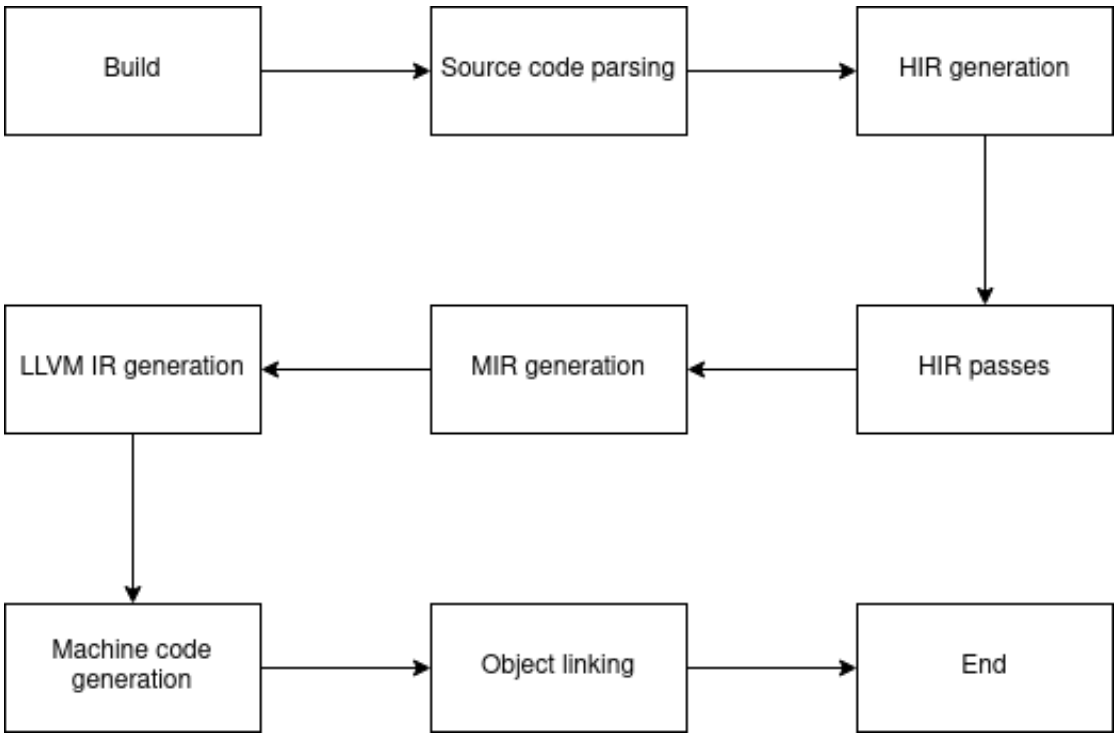
2.4.2 The build command

The build command builds the project in the current directory. It utilises every stage of the compiler . The result if this command could be binary, dynamic library or static library depending on the configuration file.

The build command has the following options:

Short name	Long name	description
q	quiet	Do not print log messages
h	help	Print help messages and exit
/	exclude	Exclude a file from check
/	lib	Check only the libraries
j	jobs	Number of parallel jobs, defaults to # of CPUs.
F	features	Enable specific features
r	release	Check in release mode
/	target	Specifies the build target
/	offline	Runs without accessing network

The build command has the following process flow:



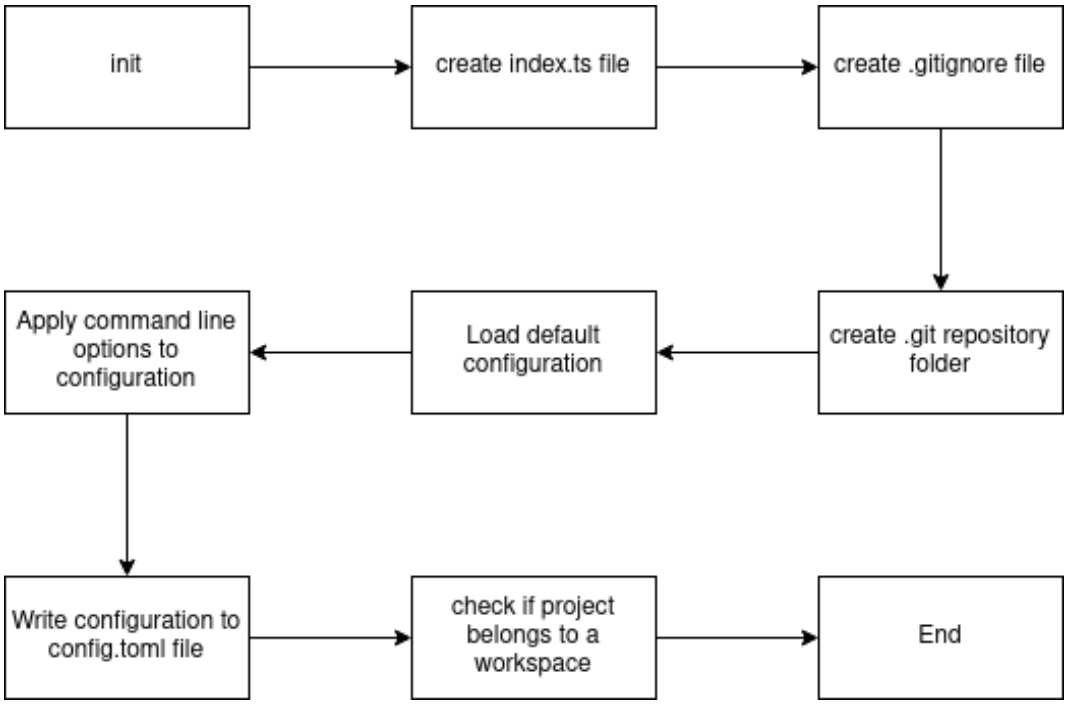
2.4.3 The init command

The init command is responsible for initialising a project directory. It creates a project configuration file with default configurations and necessary files for git repository.

The init command has the following flags:

Short name	Long name	description
q	quiet	Do not print log messages
h	help	Print help messages and exit
/	lib	The project is a library
/	target	The project is bounded to a target
v	version	The project is bounded to a specific version of the compiler
/	ts-version	The project is bounded to a specific version of Typescript

The init command has the following process flow:

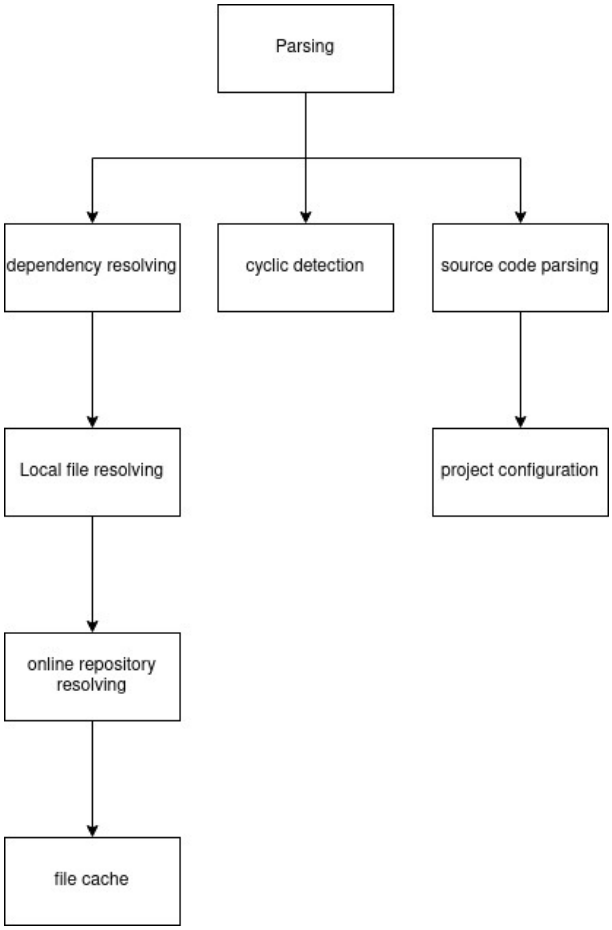


2.5 Parsing

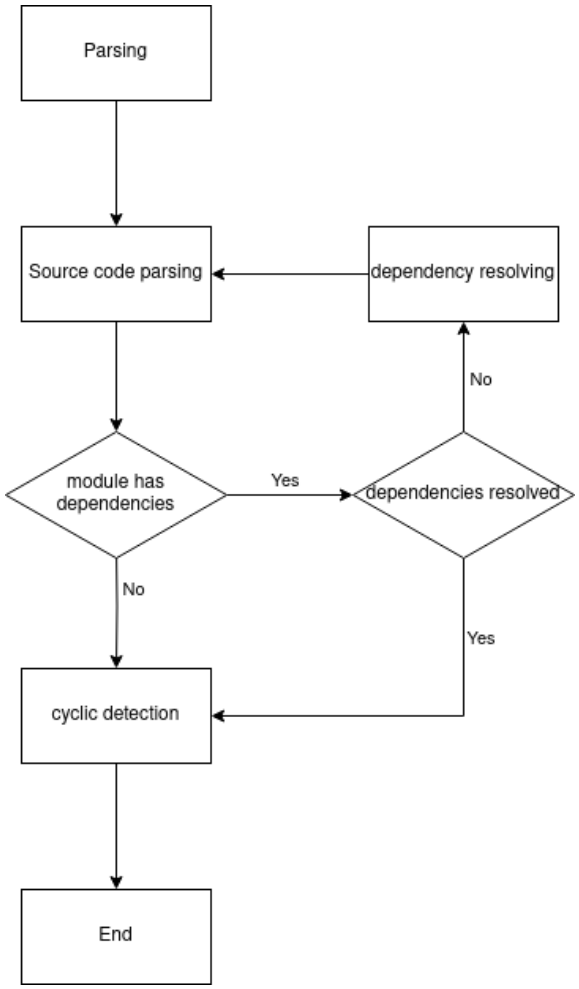
The parsing stage of the compiler parses source code into an Abstract Syntax Tree. It is responsible for lexical analysis, syntax analysis and dependency analysis. Lexical analysis and Syntax analysis is handled by a third-party library.

This module also handles parsing of the configuration file. The configuration file is also parsed using a third-party library.

The overall structure of module is as follow:



The programme flow is as follow:



2.5.1 Project configuration file parsing

The project configuration determines how the source code is parsed. The project configuration is defined by the user. It should describe the supported syntax, the version of compiler supported, the compile target etc.

The file format chosen for the configuration file is the TOML format. TOML is an easy to understand and very readable format. It is highly supported by Rust libraries and is easy to parse.

The configuration file overrides the default configuration when specified. When a field in configuration is not present, the default value is used.

The configuration file has the following sections: project, lib, bin, dependencies, target, badges, features and profile.

The project section has the following fields:

name	data type	description
name	string	The name of the project
version	string	The version of the project
authors	string array	The authors of the project
compiler-version	string	The version of the compiler
ts-version	string	The version of Typescript
description	string	A description of the project
documentation	string	URL to the project’s document
readme	string	A path to the project’s README file
homepage	string	URL to the project’s homepage
repository	string	URL to the project’s repository
licence	string	The project’s licence
licence-file	string	Path to the text of the licence
keywords	string array	Keywords for the project
categories	string array	Categories of the project
exclude	string array	Files to exclude from the project

Lib

The lib section has the following fields:

name	data type	description
test	boolean	Enable testing
lib-type	string	The type of library to be generated
features	string array	Features to enable when building as a library

Bin

The bin section has the following fields:

name	data type	description
test	boolean	Enable testing
features	string array	Features to enable when building as a binary

Dependencies

The dependencies section has no definite field. All fields in the dependencies section is defined by the user. The field name of the dependency is how the source code will see the library.

Each dependency has the following fields:

name	data type	description
path	string	Path to the library’s project directory
url	string	The URL to the dependency’s project repository
git	string	The URL of the dependency’s git repository
version	string	Version of the library
features	string array	Features to enable in the library
optional	boolean	Whether the dependency should be included by default

Target

The target section has the following sub sections:

name	data type	description
windows	/	The project targets Windows
unix	/	The project targets unix like platforms
linux	/	The project targets Linux
darwin	/	The project targets IOS or MacOS
macos	/	The project targets MacOS
ios	/	The project targets IOS
freebsd	/	The project targets FreeBSD
openbsd	/	The project targets OpenBSD
redox	/	The project targets RedoxOS
android	/	The project targets android
x86	/	The project targets x86 architecture
x86_64	/	The project targets x86_64 architecture
arm	/	The project targets arm architecture
aarch64	/	The project targets aarch64
riscv	/	The project targets riscv architecture
wasm32	/	The project targets WebAssembly

Each subsection has the following fields:

name	data type	description
dependencies	string array	Dependencies for this target
features	string array	Features enabled for this target

Features

The features section have no fields. The user specifies field names with an array value specifying what features are linked to this feature. If the specifier starts with “dep:” then the corresponding optional dependency will be enabled.

For example:
[features]
some_feature = [“dep:some_library”]

The above example means that when feature ‘some_feature’ is enabled, ‘some_library’ is no longer optional and is required to the project.

Profile

The profile section provides low-level compiler configurations. There is two profiles(sub sections) that are available. The debug profile is used when –release option is not specified. It uses level 1 optimisation and is suitable for debugging. The release profile is used when the –release option is specified. It is default to level 3 optimisation and does not contain debug info. It is suitable for deployment.

The profile section has the following sub sections:

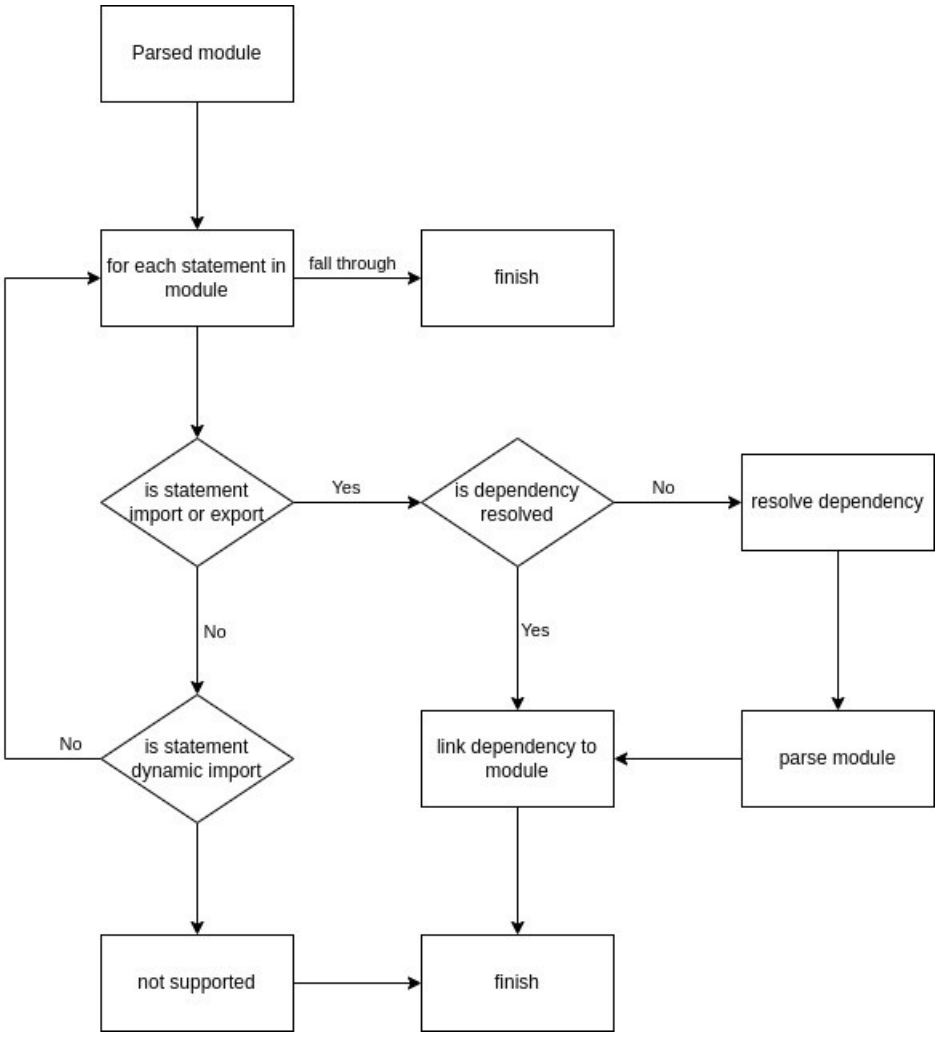
name	data type	description
debug	boolean	Debugging build, less optimisation
release	boolean	Release build, highest optimisation

Each profile sub section has the following fields:

name	data type	description
opt-level	integer	The optimisation level of the compilation
debug	boolean	Include debug info, defaults to true
lto	boolean	Enable link time optimisation
incremental	boolean	Enable incremental building

2.5.2 Source code parsing

Parsing source code is a complicated task when such a large specification like the ECMA and Typescript have to be implemented. Therefore, a third-party library is chosen for this Task. The library selected for this task is the SWC project. The SWC project is a JavaScript and Typescript transpiling library. It supports up to the latest and proposed ECMA and Typescript standard. This library also has the ability to perform several analysis and transformations. It is licenced under the Apache-2.0 licence and therefore fully usable by this project.



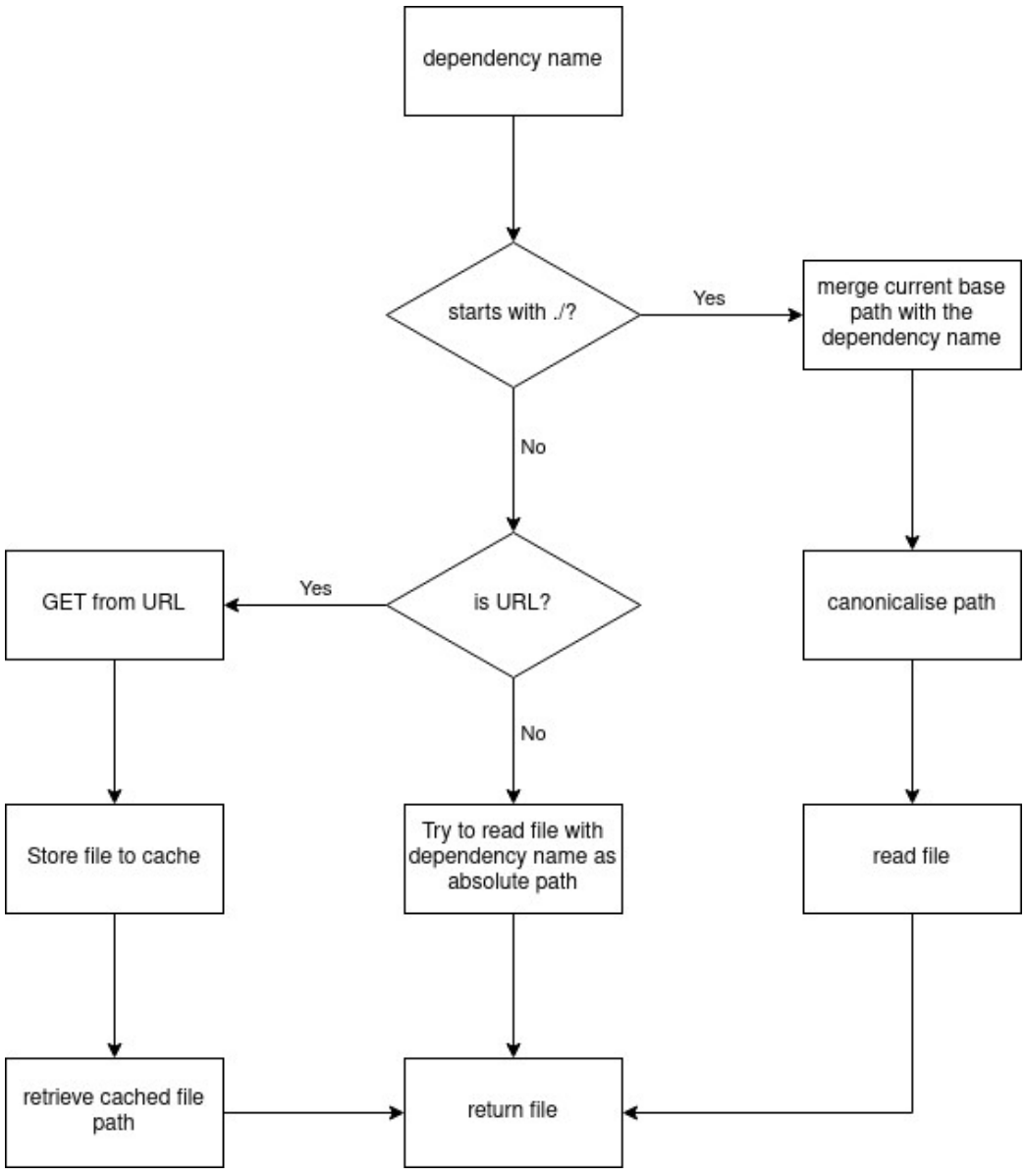
2.5.3 Dependency resolving

Dependencies must be resolved after parsing each source code module. It is to ensure that the parsed file is present before the next stage of compilation happens.

This project allows two types of dependencies: local and online. Local dependencies are source code files that are stored inside the project directory. Source code files that are stored outside the project directory is not allowed and will result in an error.

Online dependencies are dependencies that are stored online. This crate accepts git repositories and http repositories. Online repositories are pulled using GET request. These libraries are cached in file storage so that these repositories does not have to be pulled from the internet every time the compiler builds. The cache can be reused by other projects in local storage. Therefore, information of repositories must be stored in a flat file.

The following logic is used when resolving dependencies:



2.5.4 Cyclic detection

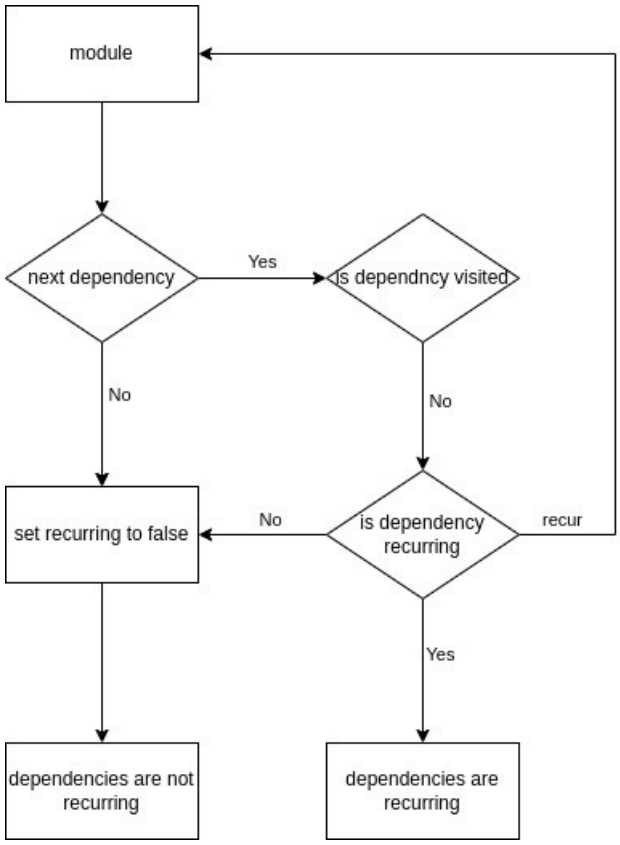
Cyclic detection is performed after all dependencies are solved. To aid the performance of compiler and better project maintainability, cyclic dependencies are not allowed. The algorithm used to detect cycles is DFS (depth first traversal). The time complexity of DFS is O(n) because each node is only visited once. It is an ideal algorithm.

A dependency graph is built during the dependency resolving stage. Each node consist of an id and its canonicalised path.

During DFS, two stacks are used to record the state: the visited stack and the recurring stack. Each stack stores boolean values stating whether each element is visited or recurring.

The algorithm loops through each node in the graph, if a child has not been visited, it will visit the child node. Upon visiting, the recurring state is set to true. The algorithm then visits all its children that has not been visited. If the child appears to be in the recurring stack, a cycle has happened. If the visit of children indicates a cycle, a cycle has happened. Upon exit, the recurring state is set to false.

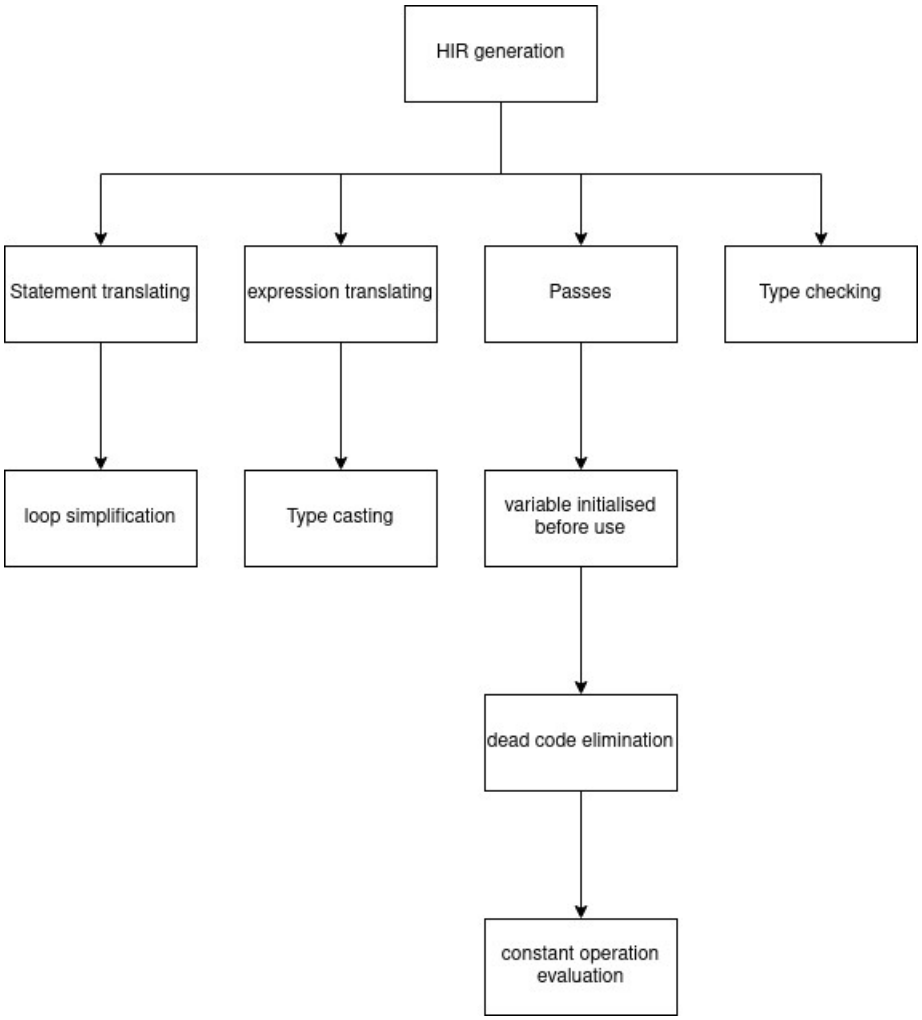
The control flow of the algorithm is as follow:



2.6 HIR

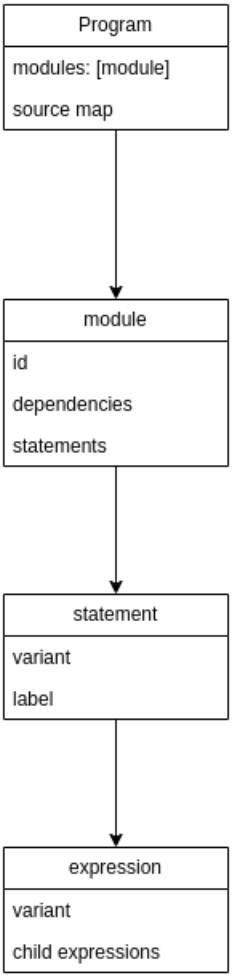
The HIR(high level IR) is essentially a more simplified version of AST. AST nodes represents direct syntax source codes. AST are therefore not suitable for computational analysis and is very hard to work with. The HIR is designed to represent source codes without destructing operation accuracy.

The AST from the parser has 240 types of nodes in total. It is very difficult to implement any analysis and optimisations on the AST. The HIR decomposes AST expression into simpler expressions thus reducing the node required to represent the source code. The HIR has in total 60 types of nodes.



2.6.1 HIR definition

The high level intermediate representation is a **Tree data structure** that represents the source code.



Below is three tables that list the variant of nodes within the HIR.

HIR expressions:

variant	description
Undefined	Loads an undefined value
Null	Loads a null value
Number	Loads a floating point number
Integer	Loads an integer
BigInt	Loads a big integer
String	Loads a string
Symbol	Loads a symbol
Regex	Loads a regular expression
Function	Loads a function
Closure	Constructs a closure
This	Loads the ‘this’ value
Array	Allocates an array
Tuple	Constructs a tuple
New	Creates an instance of a class
Call	Calls a function or closure
Member	Access a property of an object
Member Assign	Assign value to property of an object
Member Update	Update value of property of an object
Var Load	Loads value from a variable
Var Assign	Assign value to a variable
Var Update	Update value of a variable
Bin	Binary operation
Unary	Unary operation
Ternary	Ternary operation
Sequential	Executes expressions in sequence
Await	Awaits a promise, only valid in async context
Yield	Yields the value, suspends the generator
Cast	Convert a value to another type
Assert non null	Assertion that value is not null

HIR statements:

variant	description
Declare class	Declares a class type, runs the initialise block if any
Declare interface	Declares an interface type
Declare Function	Declares a function, initialise if it is a closure
Declare Variable	Declare a variable, allocates on stack or heap
Drop Variable	Drops a variable, call the destructor and release memory
Block	Creates a breaking point block
End Block	End of a block
If	Branch if condition is true
End If	End of branch
Else	Executes when If fails, must be after End If.
End Else	End of Else
Switch	A switch matches the expression
Switch Case	A switch case. Branched to when value matches
End Switch Case	End of a switch case
Default Case	A default case. Branched to when no switch case matches
End Default Case	End of a default case
End Switch	End of switch statement
Loop	Loop
End Loop	End of Loop
Try	Try statement, catches error
End Try	End of Try
Catch	Catches errors from Try, must be after End Try
End Catch	End of catch
Finally	Executed unconditionally, must be after End Try or End Catch
End Finally	End of Finally
Break	Breaks from a block or a loop
Continue	Skip the current loop and jump to the next loop
Return	Returns a value from the function
Throw	Throws an error
Expr	Executes an expression

variant	description
Any	Any value, same as an interface without properties
Any Object	Any object, could be any value except number, string, boolean, bigint, symbol, null or undefined
Undefined	Undefined type, only accepts undefined value
Null	Null type, only accepts null value
Bool	Boolean type, only accepts true or false
Number	Number, a floating point number.
Integer	32bit signed integer type. This type is used when number declared in source code is certain to be integer at compile time. This type cannot be declared by user explicitly. It is implemented to speed up arithmetic performance.
BigInt	Big integer, it is implemented as a signed 128bit value.
String	String. It is represented using a structure of null terminated pointer and a usize length value.
Symbol	A symbol is a unique representation of property key. It is implemented using a 64bit hash value.
Regex	A regular expression.
Object	An object is an instance of a class.
Interface	A dynamic type that accepts any type fulfilling the requirements.
Function	A function type. Maybe a closure or a raw function
Enum	A Enum is an integer value that represents a state.
Array	An Array has dynamic length. Type of each element must be the same.
Tuple	Tuple has fixed number of elements. However, each element can have a different type.
Iterator	An Iterator follows the ECMA iterator protocol. It is essentially an interface with ‘next’ method.
Alias	An alias type represents a possible unknown type during translation. This type is only used during translation and will not be precent outside HIR generation
Generic	A generic type represents an unknown type during translation. It is replaced by user provided type arguments during translation. Generics will not be precent outside HIR generation.

2.6.2 HIR translation

The HIR translation stage generates HIR from AST parsed by the parser. During the translation, several checks are also performed, namely Type Checking.

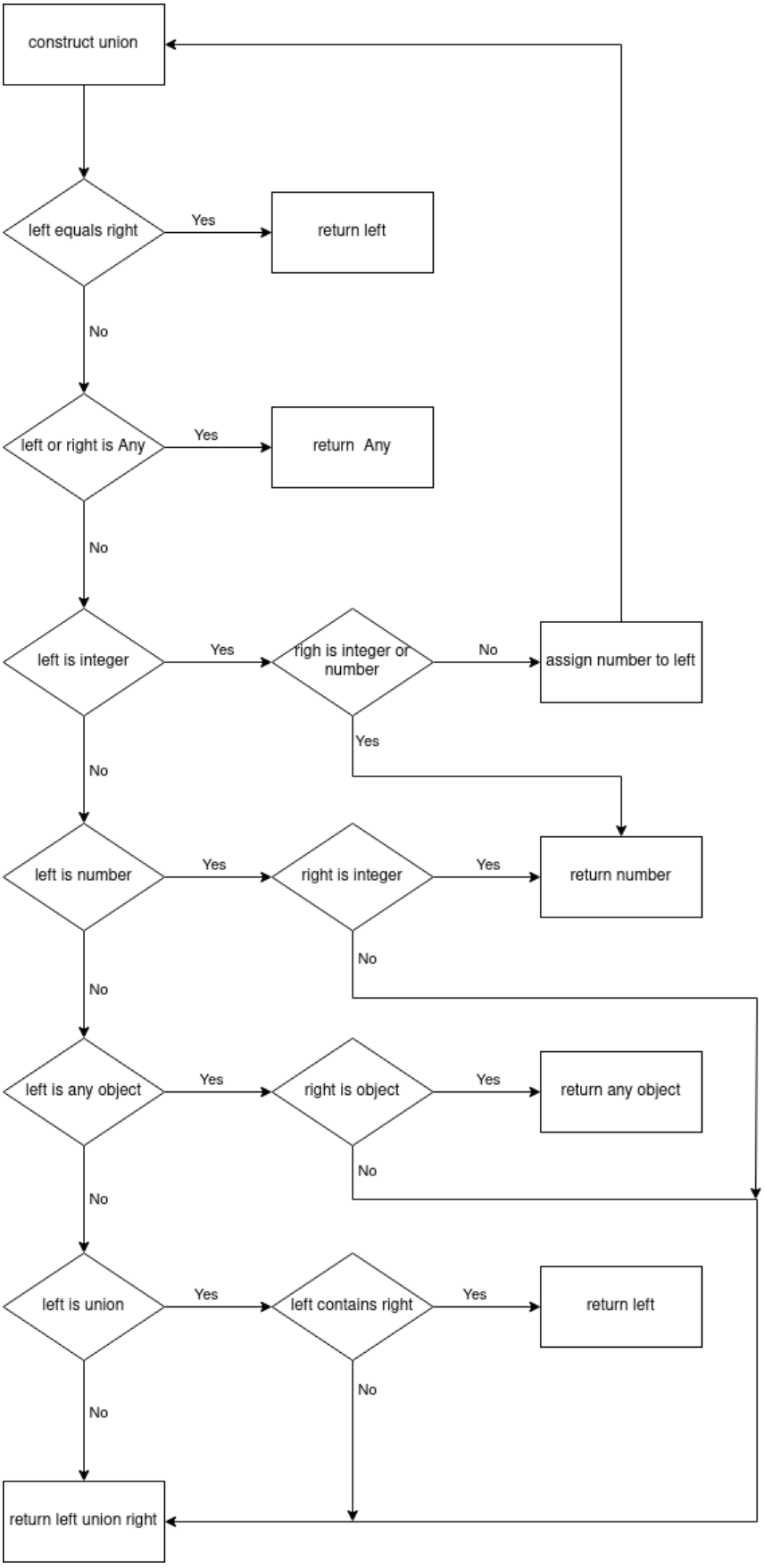
It the beginning of translation of a source code module, all its dependencies are checked and must be translated. All the Classes, interfaces, Enums and functions are hoisted. It then import bindings from its dependencies. The module exports are then registered. At last, all its statements are translated.

HIR translation is separated into three parts: type translation, expression translation ad statement translation. Type translation translate type annotations and perform type checks it also manages type aliases and generic types. Generic types must be resolved during the translation as the resulting HIR must have concrete types. This part is especially tricky since generics may be referenced by the inner contexts.

During expression translation, expressions are translated. Each expression translation will return a single expression and a type. During translation, the translator may be given an expected type. The translator references the expected type to determine how the expression should be translated. Type casting will be automatically injected during this stage. If the expression’s type will not match the expected type, a type error is returned to the user.

During statement translation, statements are translated. This includes loops, branches, break, return etc. A statement can also be an expression. In that case, The returned value is discarded. The translator records block labels during this stage. When a loop or a block statement is coupled with a label, it is pushed on the stack. When a break or continue statement is used, it checks against the label. The translator also holds a context with a symbol table. A new scope is created every time a block is declared. The scope holds its own symbol table that records classes, interfaces, enums, type aliases and variables. The last scope is closed when translator reaches the end of a block.

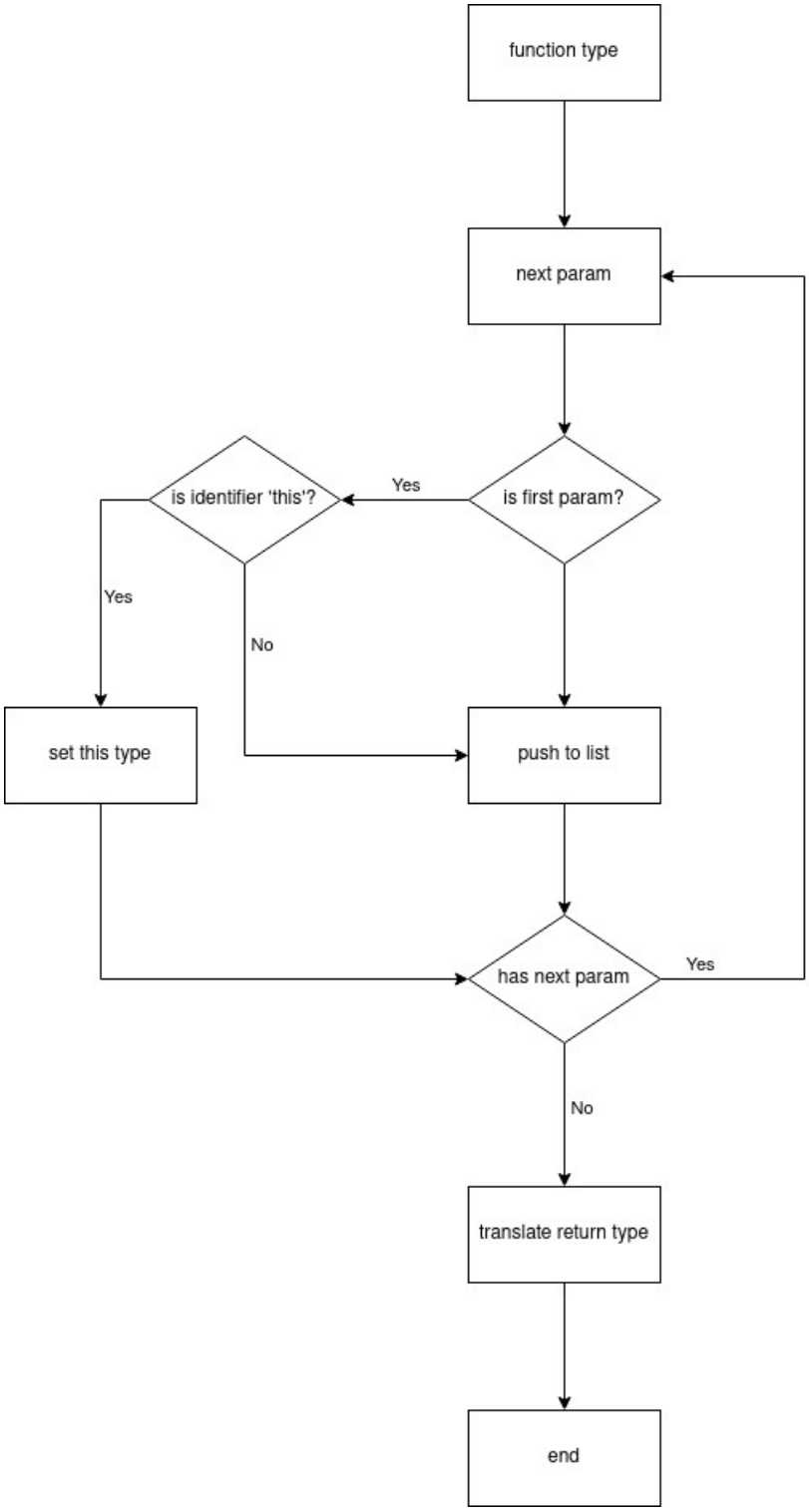
Constructing union types



Here is a branch table for translating types in the AST

condition	branch to	result
Array type	translate_type	Array type of Type
Conditional type	translate_conditional_type	Type
Function type	translate_func_type	Function type
Constructor type	/	Error
Import type	/	Error
Index Access type	translate_index_access_ty	Map type
Infer type	/	Error
Keyword type	translate_keyword_type	Type
Literal type	/	Error
Mapped type	/	Error
Optional type	translate_type	Type union undefined
Parenthesized type	translate_type	Type
Rest type	/	Error
This type	/	Current context ‘this’ type
Tuple type	translate_tuple_type	Tuple type
Type literal	/	Error
Type operator	translate_type_operator	Type
Type Predicate	translate_type_predicate	Type
Type Query	translate_type_query	Type
Type reference	translate_type_ref	Type

Translating function type



Translating key word type

keyword	result
any	Any
bigint	BigInt type
boolean	Boolean type
intrinsics	*Error
never	Undefined type
null	Null type
number	Number type
object	Any Object type
string	String type
symbol	Symbol type
undefined	Undefined type
unknown	Any type
void	Undefined union null

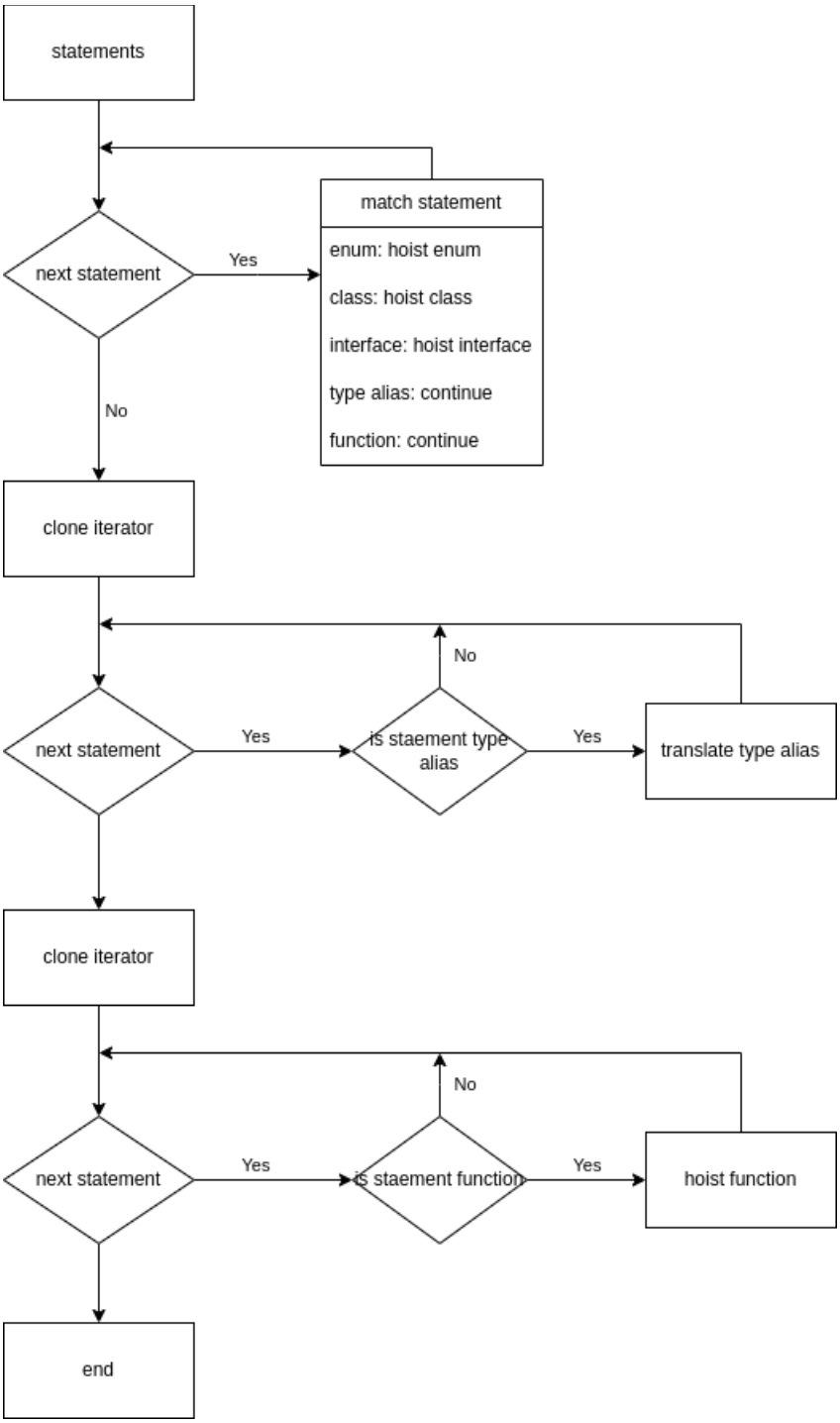
Translate type query

Binding type	result
Generic Function	Error
Function	Function type
Variable	Type
Using variable	Type
Class	Error
Generic class	Error
interface	Error
Generic interface	Error
Type alias	Error
Generic type alias	Error
Enum	Error
Namespace	Error

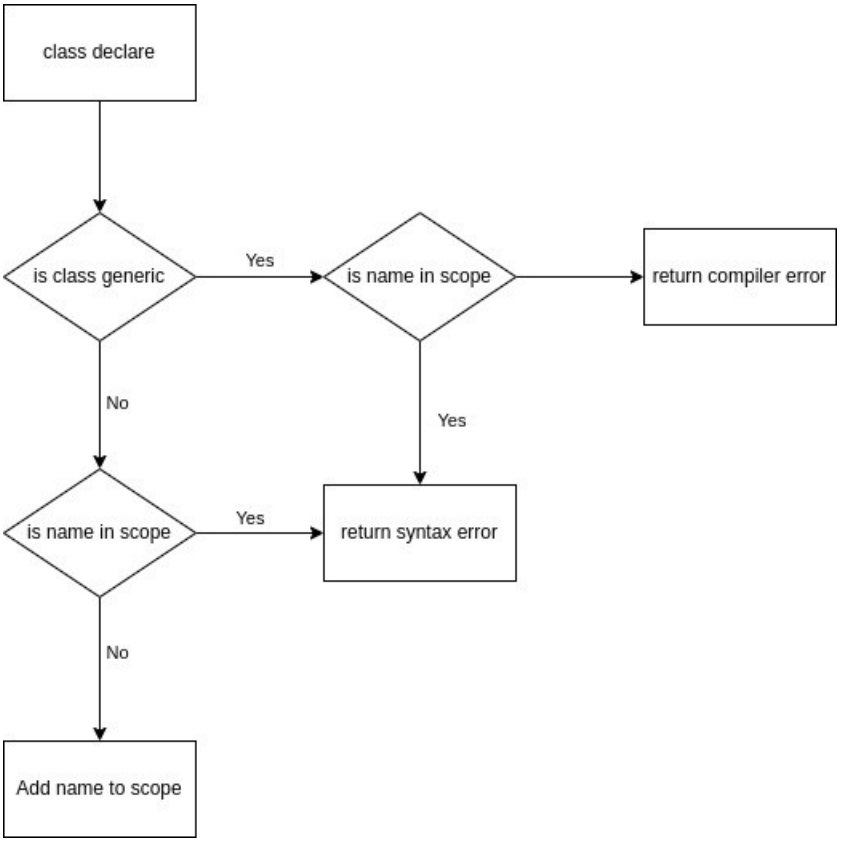
Translate type reference

Binding type	result
Generic Function	Error
Function	Error
Variable	Error
Using variable	Error
Class	Class type
Generic class	Class type
interface	Interface type
Generic interface	Interface type
Type alias	Type
Generic type alias	Type
Enum	Enum type
Namespace	Error

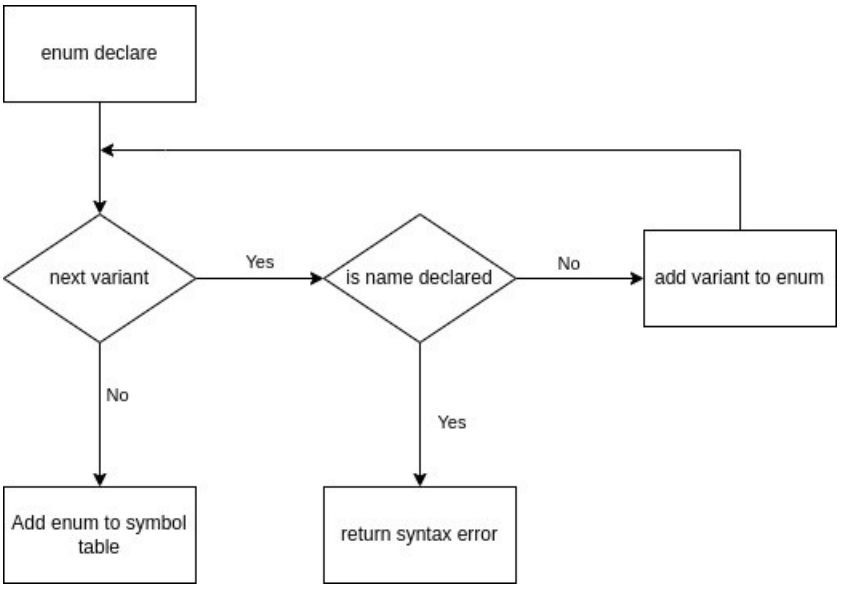
Statement hoisting



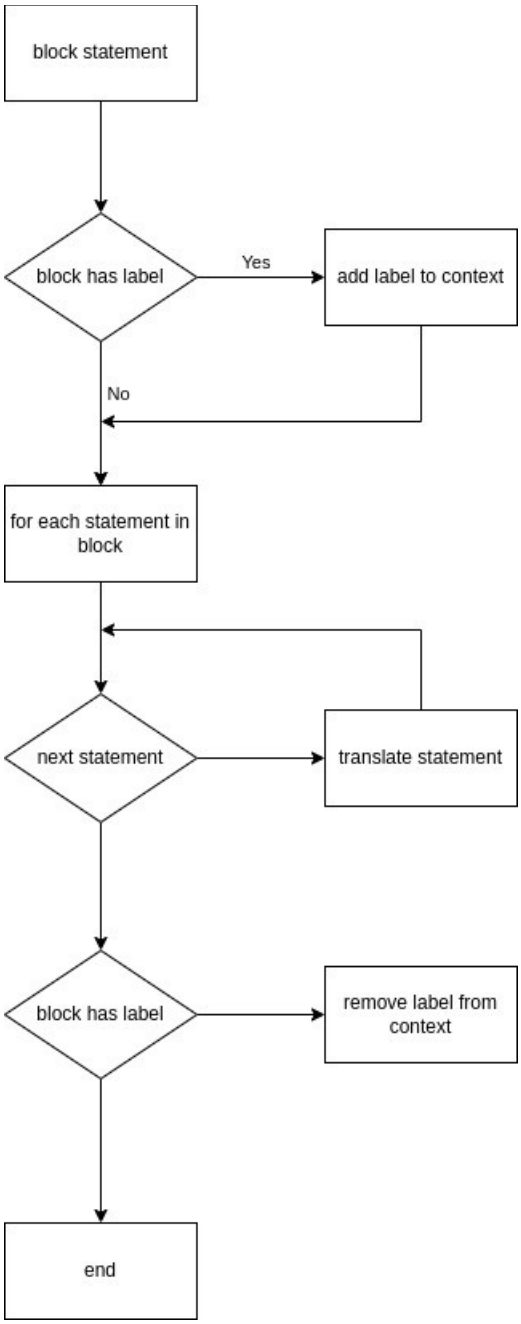
class hoisting



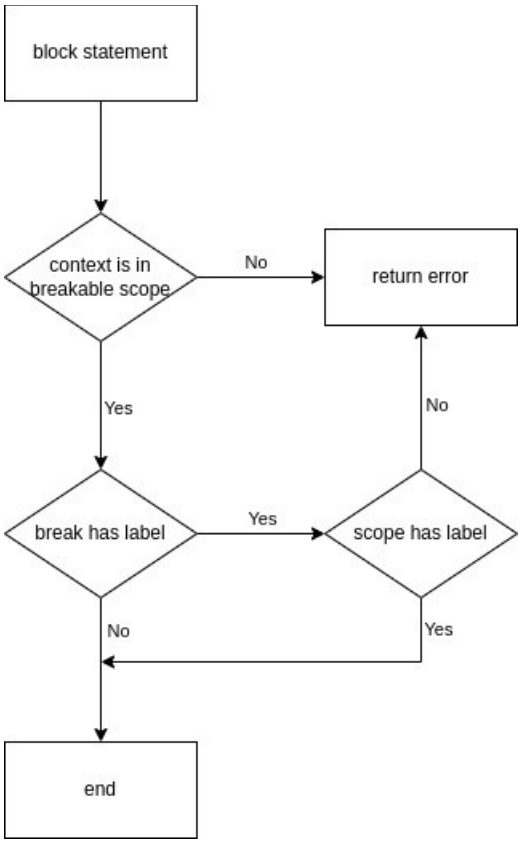
enum hoisting



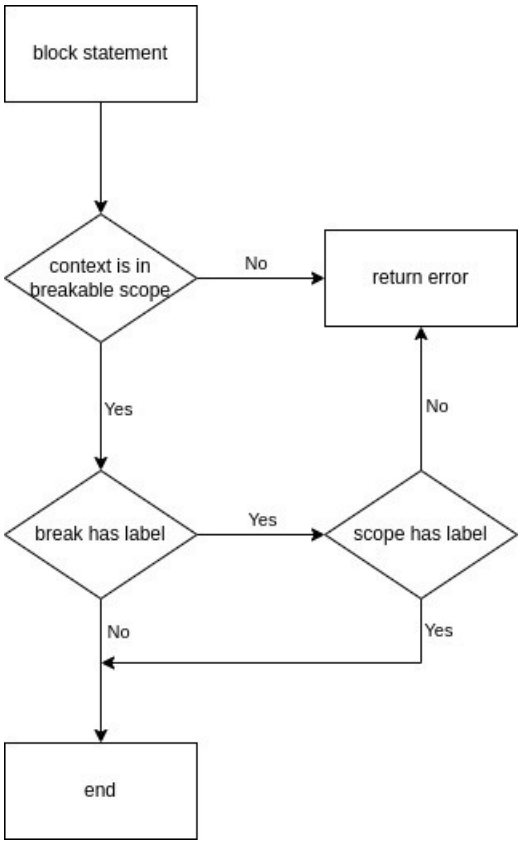
Translating block statement



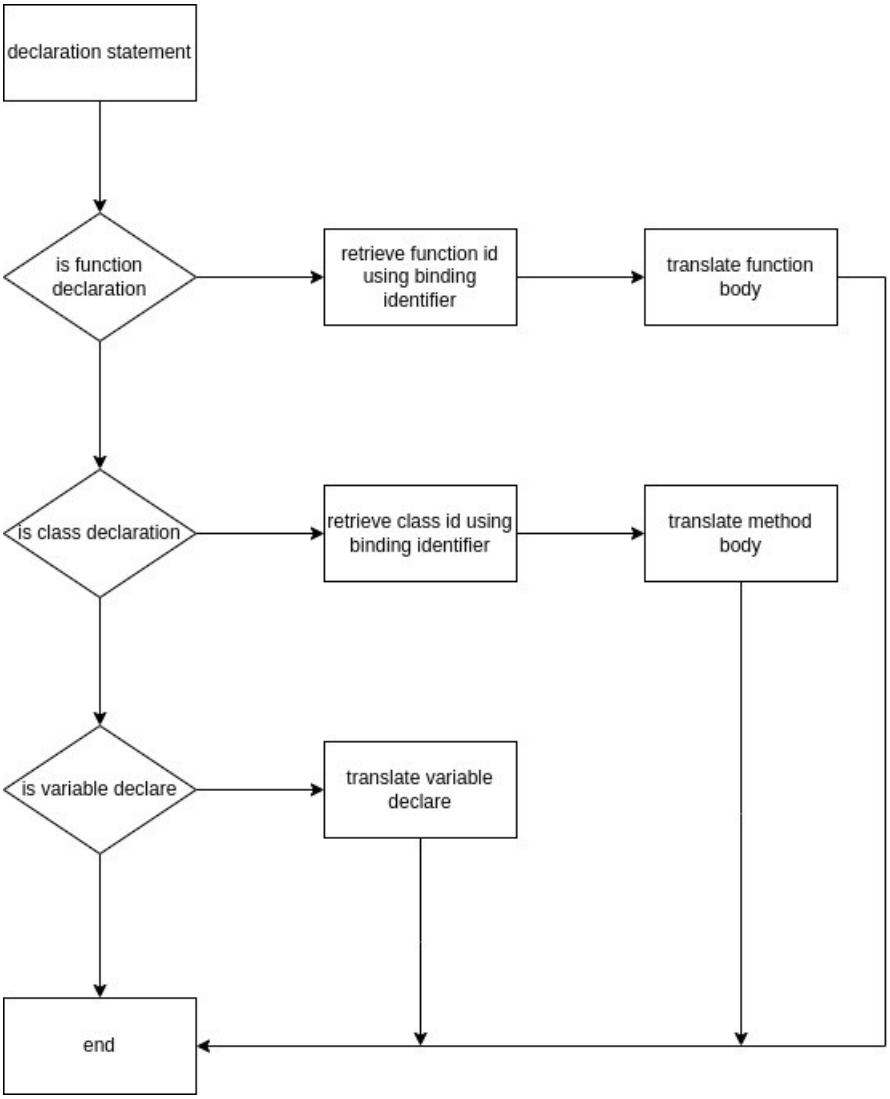
translate break statement



translate continue statement



Translate declare statement



Translate variable declaration

There are four variables types that can be declared, **`var`**, **`let`**, **`const`** and **`using`**. Each type of variable declaration behaves differently.

The `var` statement declares function-scoped or globally-scoped variables, optionally initializing each to a value. `var` declarations, wherever they occur in a script, are processed before any code within the script is executed. Declaring a variable anywhere in the code is equivalent to declaring it at the top. This also means that a variable can appear to be used before it's declared. This behaviour is called *hoisting*, as it appears that the variable declaration is moved to the top of the function, static initialization block, or script source in which it occurs.

The `let` declaration declares re-assignable, block-scoped local variables, optionally initializing each to a value. It is similar **`var`**, however when compared to **`var`** declaration, it has the following restriction:

- **`let`** declarations can only be accessed after the place of declaration is reached. It is not hoisted.
- **`let`** declarations do not create properties on `globalThis` when declared at the top level of a script.
- **`let`** declarations cannot be redeclared by any other declaration in the same scope.

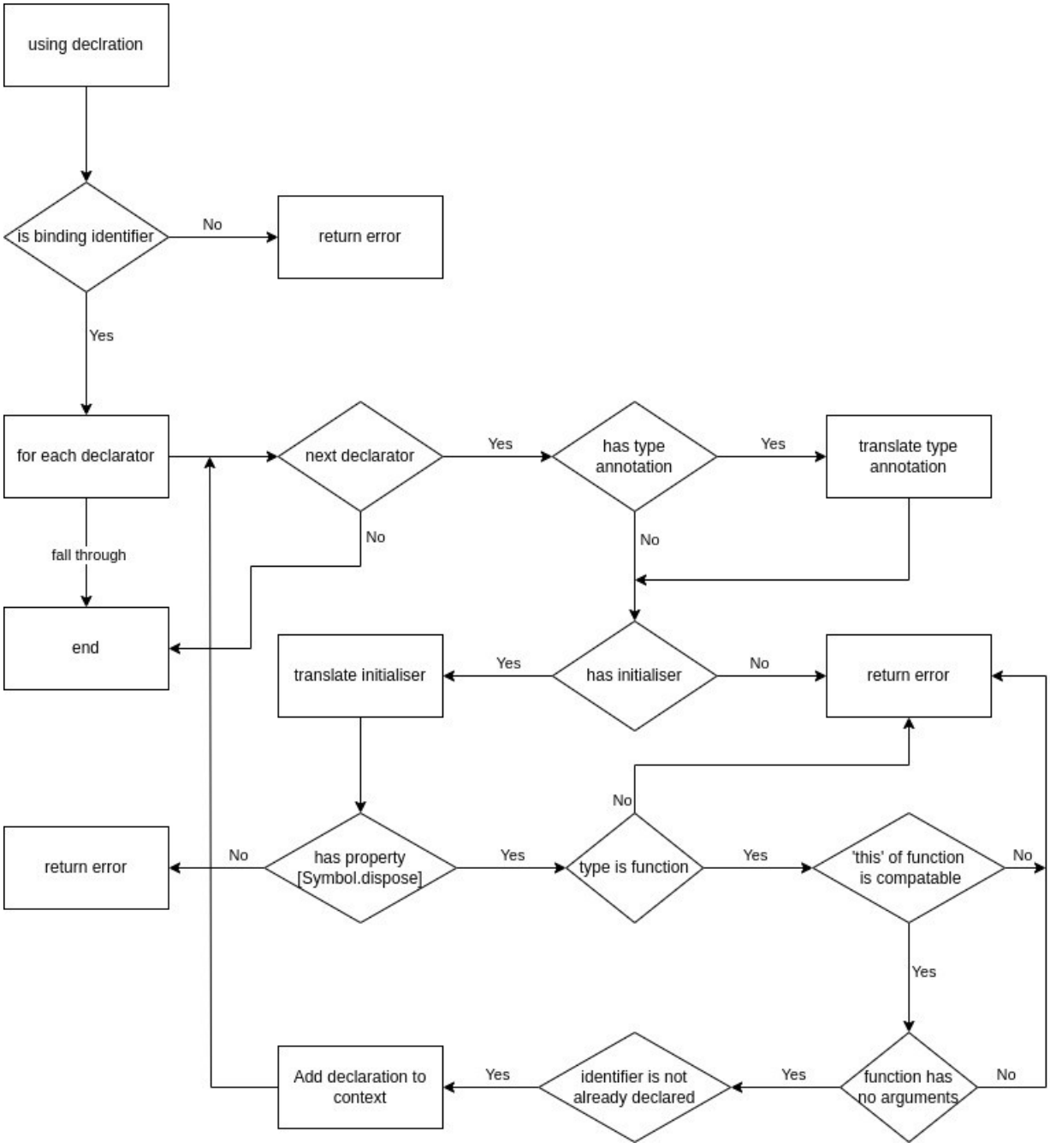
The `const` declaration declares block-scoped local variables. The value of a constant can't be changed through reassignment using the assignment operator, but if a constant is an object, its properties can be added, updated, or removed.

The **`using`** declaration declares non-assignable, block-scoped variables. It must be initialised with a value and cannot be changed through reassignment. It cannot be captured by a closure and is only visible within its local scope. The dispose method is called once variable drops out of scope whether or not an error has been thrown.

A variable declaration statement main contain multiple declarators. This means that declaring multiple variables with different initialiser is allowed in a single statement. These declarators shares a single variable type either var, let, const or using. The declarators are allowed to have destructive patterns such as object pattern and array pattern. These patterns decomposes the initial value into multiple variable by accessing its fields. However this representation is complex and is not suitable for an IR code. Therefore we will have to decompose destructive patterns into simple assignment expressions. Destructive pattern is not allowed when in **`using`** declaration.

A using variable declaration must have an initialiser this is because using variables must be called upon the dispose method when it is out of scope. Using variables cannot be left uninitialised in any instance of the programme as there is a possibility that it may panic somewhere during the execution causing the value to be disposed.

To translate the using variable declaration, we loop through each declarator. We check that the binding pattern is an identifier, or else return an error. We then translate the type annotation if given. We translate the initial value and type check against its type making sure that it has the [Symbol.dispose] or [Symbol.asyncDispose] method.



2.6.3 pass: variable initialised before use

This HIR pass is used to detect variables that are accessed before it is initialised. This is because reading from a typed variable that is not initialised is undefined behaviour, the content of the value is random and cannot be guaranteed.

This pass traversal the HIR tree using depth first search algorithm. It will remember any variable that is initialised in a particular scoop. This pass will have the ability to determine whether a variable is initialised based on conditional branches. When branching happens, only if all branches guarantees to initialise the variable shall the variable be set to initialised state, otherwise, it will be treated as if it is not yet initialised.

2.6.4 pass: dead code elimination

This pass performs dead code elimination. This is to reduce the amount of code required to translate to MIR.

This pass traversals the HIR using depth first search algorithm. When a termination condition is met, its subsequent statements would be removed.

2.6.5 pass: constant operation evaluation

This pass performs constant operations that can be calculated during runtime. This is to reduce the size of cod and increase performance.

This pass traversals the HIR using depth first search algorithm. It checks whether an expression is constant. If all the operand of an expression is constant then the expression itself must also be constant.

2.6.6 pass: class construction

This pass transforms the constructor of a class the meet the rules of the language. Additionally, it checks whether the attributes are initialised in a constructor. Attributes of a newly created class must be initialised. It also checks for super calls within the constructor. Referencing the object itself before the super class constructor is called is forbidden during construction.

This pass traversals the HIR using a pre-order search algorithm. Scopes are pushed and pop from a stack when a branch is made. The initialisation of an attribute is recorded into the scope, this is valid until the scope ends. Whether or not the attribute is initialised depends on it being initialised in all possible branches.

If the attribute is not initialised by the end of a constructor, an error would be returned.

2.7 MIR

The MIR is a middle-level IR that bridges the gap between HIR and LLVM IR. It allows the compiler to ignore low level syntax transformations that adds complexity to the process of generating LLVM IR.

The MIR has several key features: virtual table generation, generator function decomposition, async function decomposition and async runtime integration.

The MIR sets out a framework for integrating the async processing runtime. It automatically inserts runtime specific codes when async functions are used. It also decomposes async functions and generator function into logical operations.

The MIR is designed to represent values in SSA format and control flow using blocks. All SSA values is subjected to the function and the flow of block. An SSA value can only be used within its function and the child block of its creation block.

2.7.1 representing operations

Operations are encoded in an enum along with its operands of SSA values. It is stored in blocks that is apart of a function. Each operation will allocate a SSA value from the context to store its result.

Operations in the MIR is represented using an **enum data type**.

2.7.2 representing interfaces

Interfaces are virtual, meaning that all methods called upon them uses dynamic dispatch. This can be done through the use of a virtual table that stores meta data about a type’s information. The virtual table is automatically generated when converting from a type to an interface. The interface is therefore represented using two pointers: a pointer to the actual value and a pointer to the virtual table.

2.7.3 smart pointers

The MIR have built in smart pointers. The format of smart pointer is unknown to the user at construction time. The resulting type of smart pointer depends on the ownership and memory management strategy chosen by the user. For example, smart pointers allocated in a function that will not be moved will be allocated on stack instead on heap.

If the user chooses Automatic reference counting as the memory management strategy, each heap allocation will allocate 8 more bytes for reference counting. The MIR will insert necessary codes to increment or decrement the reference count.

2.7.4 representing objects and type information

Objects are encoded during MIR building. The object types should be recorded and its size and alignment is calculated. The inheritance of the object type should be normalised and included.

The type information of each object type should be encoded in a flat format so that type information can be written into the executable file's static memory for reflection. The reflection function of the runtime depends on the type information generated at this stage.

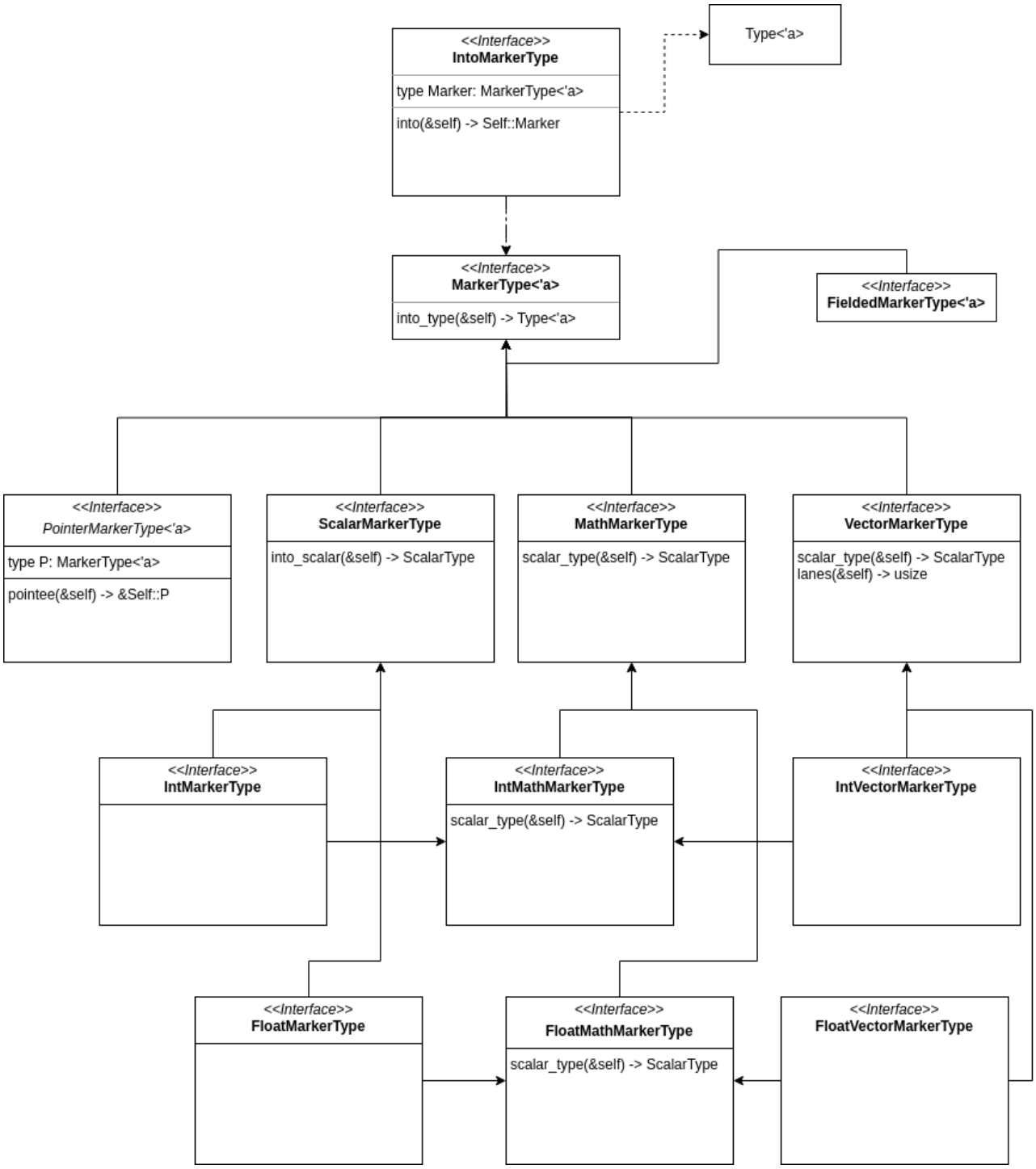
The format of encoded type information should remain consistent among MIR and the runtime. Therefore, a separate crate would be used to encode and decode type information. The MIR crate and the runtime crate will import this library to encode and decode type informations. See section 2.9.1 type reflection

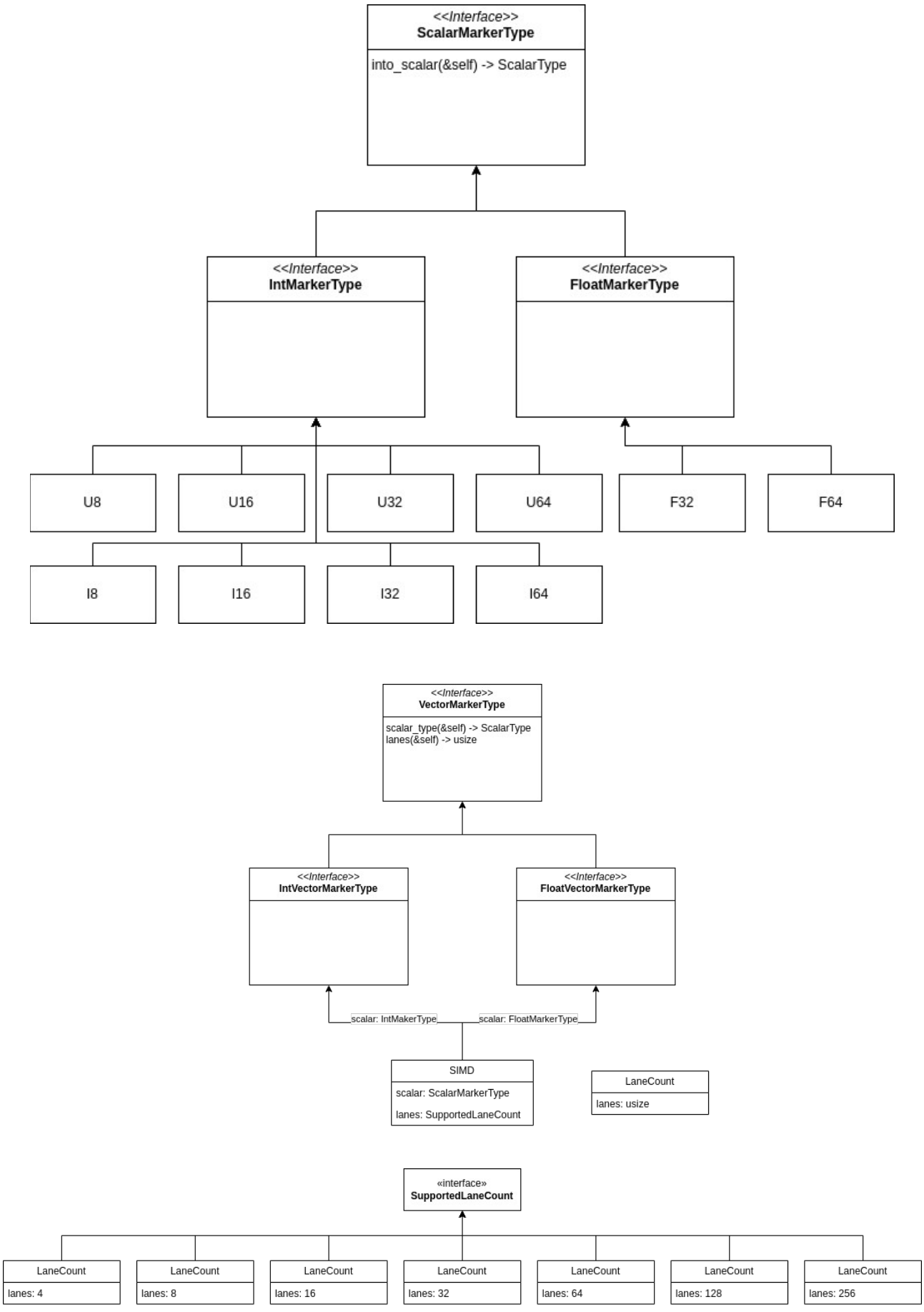
2.7.5 static type guards

Static type guards are used in the MIR to reassure types of SSA values. The SSA value is coupled with a type guard. The type guard implements a certain set of trait that will allow SSA values to be used in certain operations within the builder and disallows usage where the conditions are not met by the type in certain operations.

In some cases, type of an SSA value may not be known during compile time. A generic version of type guard is therefore used in this case. The type guard would be checked against during runtime dynamically. This type of type guard should only be used when necessary to ensure readability and maintainability of programme.

Below is a diagram of Marker Type interface relationship:




<<Interface>>
ScalarMarkerType
into_scalar(&self) -> ScalarType
<<Interface>>
IntMarkerType
<<Interface>>
FloatMarkerType
U8
U16
U32
U64
F32
F64
I8
I16
I32
I64
<<Interface>>
VectorMarkerType
scalar_type(&self) -> ScalarType
lanes(&self) -> usize
<<Interface>>
IntVectorMarkerType
<<Interface>>
FloatVectorMarkerType
scalar: IntMakerType
scalar: FloatMarkerType
SIMD
scalar: ScalarMarkerType
lanes: SupportedLaneCount
LaneCount
lanes: usize
«interface»
SupportedLaneCount
LaneCount
lanes: 4
LaneCount
lanes: 8
LaneCount
lanes: 16
LaneCount
lanes: 32
LaneCount
lanes: 64
LaneCount
lanes: 128
LaneCount
lanes: 256

2.7.6 Asynchronous framework

A standard API should be defined by the MIR to integrate an asynchronous runtime. This set of API will enable generic implementation of future updates and allows the compiled code to maintain a consistent way of handling asynchronous tasks.

The API is designed as follow:

structure	type	description
Task context	Compiler generated	A task context is a variable sized structure specific to the task. It is use to store the current state of the execution such as variables, program counters and more. Runtime should see this as an opaque.
Async Task	User defined	An Async Task is a structure that stores the poll function, context and result along with other runtime specific data. It should contain enough memory for context and result to evaluate during task execution.
Task Handler	User defined	A Task Handler references a spawned task in the runtime. It is used to poll or retrieve status of a task.

method	description
Create Async Task	Creates a task structure according to the runtime implementation. The arguments would be a poll function pointer, context size and the result size. The runtime is responsible for allocating enough memory for the context and result value.
Spawn Async Task	Spawns the Asynchronous task on the global executor. The runtime is responsible for initialising the executor if not already. How the Async task is carried out is up to the runtime implementation. This method should return a Handler to the spawned task.
Poll Task	Polls a task given the task handler. It is up to the runtime to decide what to do with the task. This function should return a reference to the result if task is ready, otherwise null.
Poll All Blocking	Polls all the task in the executor that is pending, this is a blocking function that will keep polling all the task in a round robin fashion until all the tasks are ready.

2.7.7 Exception handling framework

A standard API is designed to accommodate runtime exception handling needs. This set of standard will allow the compiler to dynamically map exception handling intrinsics to the LLVM build.

structure	type	description
Exception	Fixed	An exception must begin with an exception header defined in the dwarf standard. It can store user data it wants after the header but the header must be present.
Exception object	User defined	An exception object is the data the runtime intend to carry in each exception. This is not the standard exception header defined in Dwarf or Itanium. This is an opaque structure for user code.

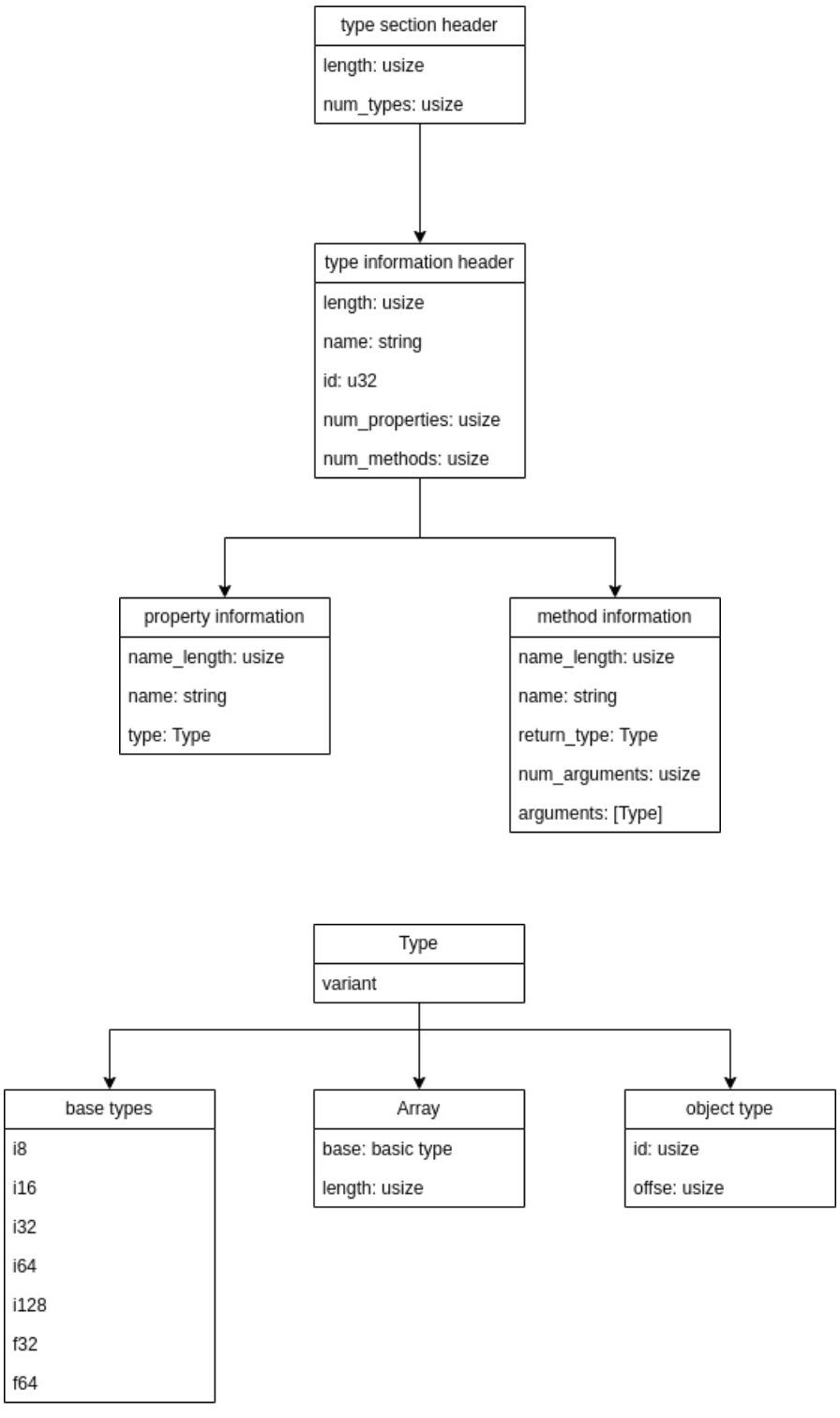
method	description
Allocate exception	Allocates an exception, returns a pointer to memory
Free exception	Frees the memory of an exception
Throw	Raise an exception. This function takes in an allocated exception and a clean up function. It is responsible to call _Unwind_RaiseException.
Begin Catch	This is called when a landing pad enters
End Catch	This is called when a landing pad exits
Resume Throw	Resume the current exception previously called by begin catch

2.9 Runtime

2.9.1 type reflection

Type reflection would be based off static binary type information generated by the compiler at compile time. A separate crate is maintained to ensure consistency across the compiler and the runtime.

The type information would be encoded in the binary format.



2.9.2 Garbage collection

A garbage collector would be implemented for the runtime. This garbage collector would be a generational, conservative concurrent collector that runs in the background similar to the golang garbage collector.

The garbage collector would incorporate a slab allocator as its primary allocation method. It would also maintain a list of allocations that are large in size that must be virtually mapped.

The garbage collector would perform garbage collection when an allocation threshold is met. This triggered during an allocation.

2.9.3 exception handling

The runtime will implement the Itanium CXX ABI for exception handling. A personality routine is defined by the runtime along with parsing of dwarf eh data in the language specific area.

We first define an exception class for our language. The exception class is an eight character long identifier. The first four byte indicates the vendor and the last four bytes indicates the language. Our exception class would be “LAM\0NATS”.

We now can define our personality routine. The personality routine takes in five parameters according to the Itanium ABI (7) :

`version`
: Version number of the unwinding runtime, used to detect a mis-match between the unwinder conventions and the personality routine, or to provide backward compatibility. For the conventions described in this document, `version` will be 1.

`actions`
: Indicates what processing the personality routine is expected to perform, as a bit mask. The possible actions are described below.

`exceptionClass`
: An 8-byte identifier specifying the type of the thrown exception. By convention, the high 4 bytes indicate the vendor (for instance HP\0\0), and the low 4 bytes indicate the language. For the C++ ABI described in this document, the low four bytes are C++\0.

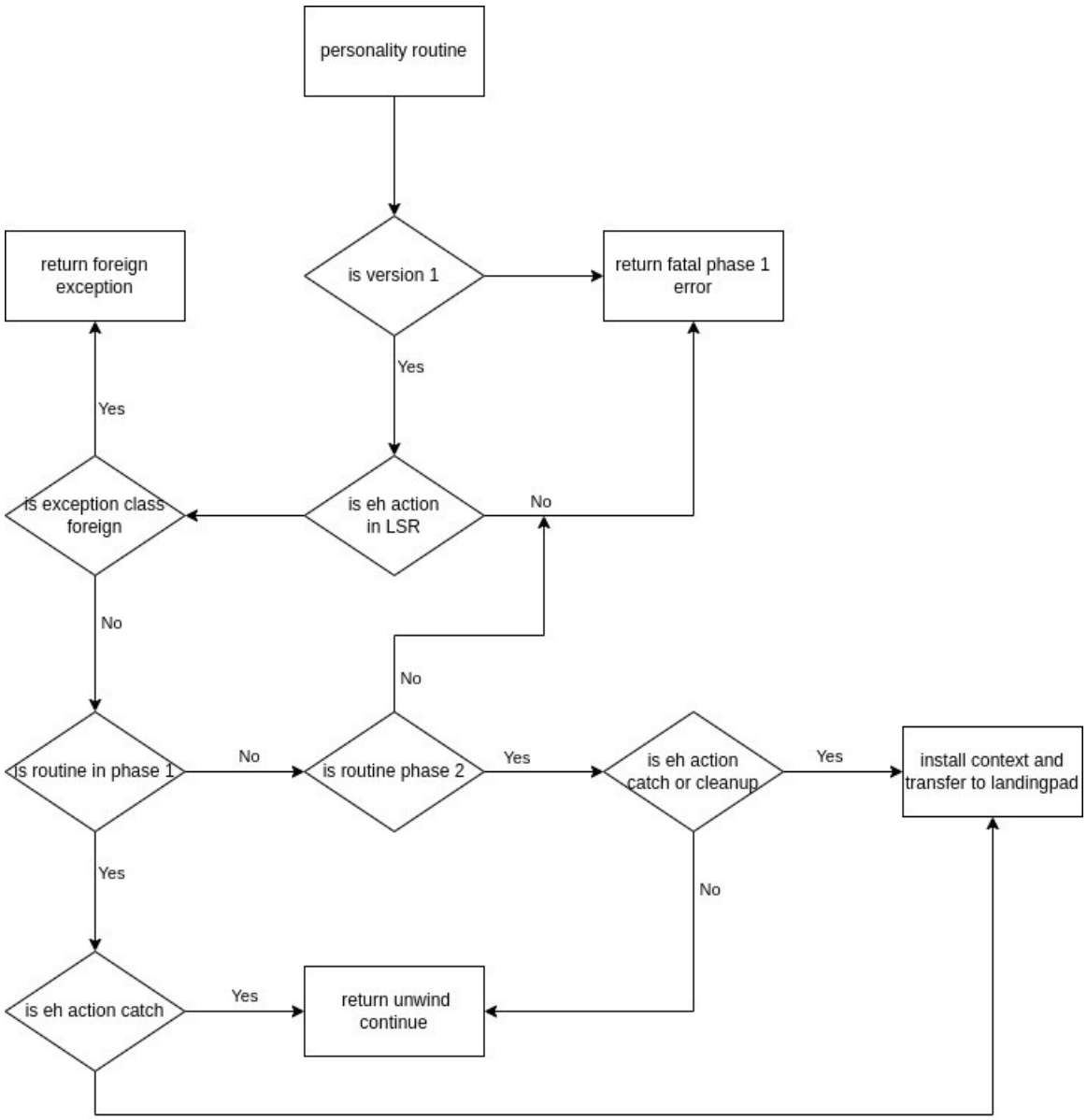
`exceptionObject`
: The pointer to a memory location recording the necessary information for processing the exception according to the semantics of a given language (see the *Exception Header* section above).

`context`
: Unwinder state information for use by the personality routine. This is an opaque handle used by the personality routine in particular to access the frame's registers (see the *Unwind Context* section above).

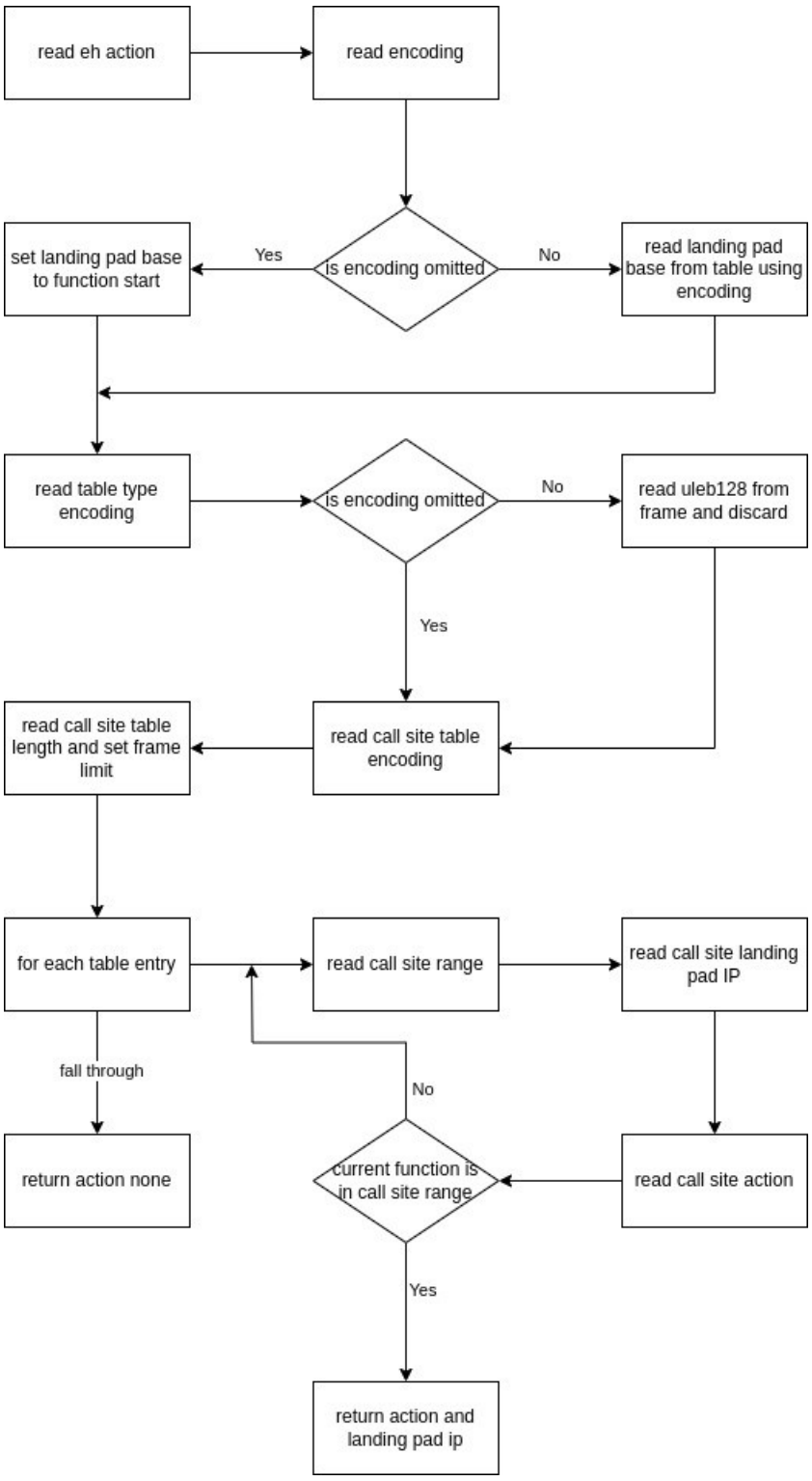
The return value from the personality routine indicates how further unwind should happen, as well as possible error conditions.

Reason code	description
Continue unwind	Continues unwind of the stack
Handler found	Exception Handler found in current frame, only valid in the search phase
Foreign Exception caught	This indicates that a different runtime caught this exception. Nested foreign exceptions, or rethrowing a foreign exception, result in undefined behaviour.
Fatal Phase 1 error	The personality routine encountered an error during phase 1, other than the specific error codes defined.
Fatal Phase 2 error	The personality routine encountered an error during phase 2, for instance a stack corruption.
Install Context	Handler is present at phase 2. The personality routine has installed the context and transfers to landing pad
End of stack	The stack has reached its end, no further unwinding should be performed.

The personality routine will first check the version of the unwind library to ensure compatibility. It will not support any version other then 1. It then checks for the exception class. If the exception class is not our exception class, this means that the exception is foreign and we will ignore it by returning. We will then check if the routine is at phase 1 (searching phase) or phase 2(clean up phase). In phase 1, if the eh action is a catch, return handler found, else return continue unwind. In phase 2, if the eh action is not none, we install the context of landing pad and transfer control to the landing pad, otherwise, return continue unwind.

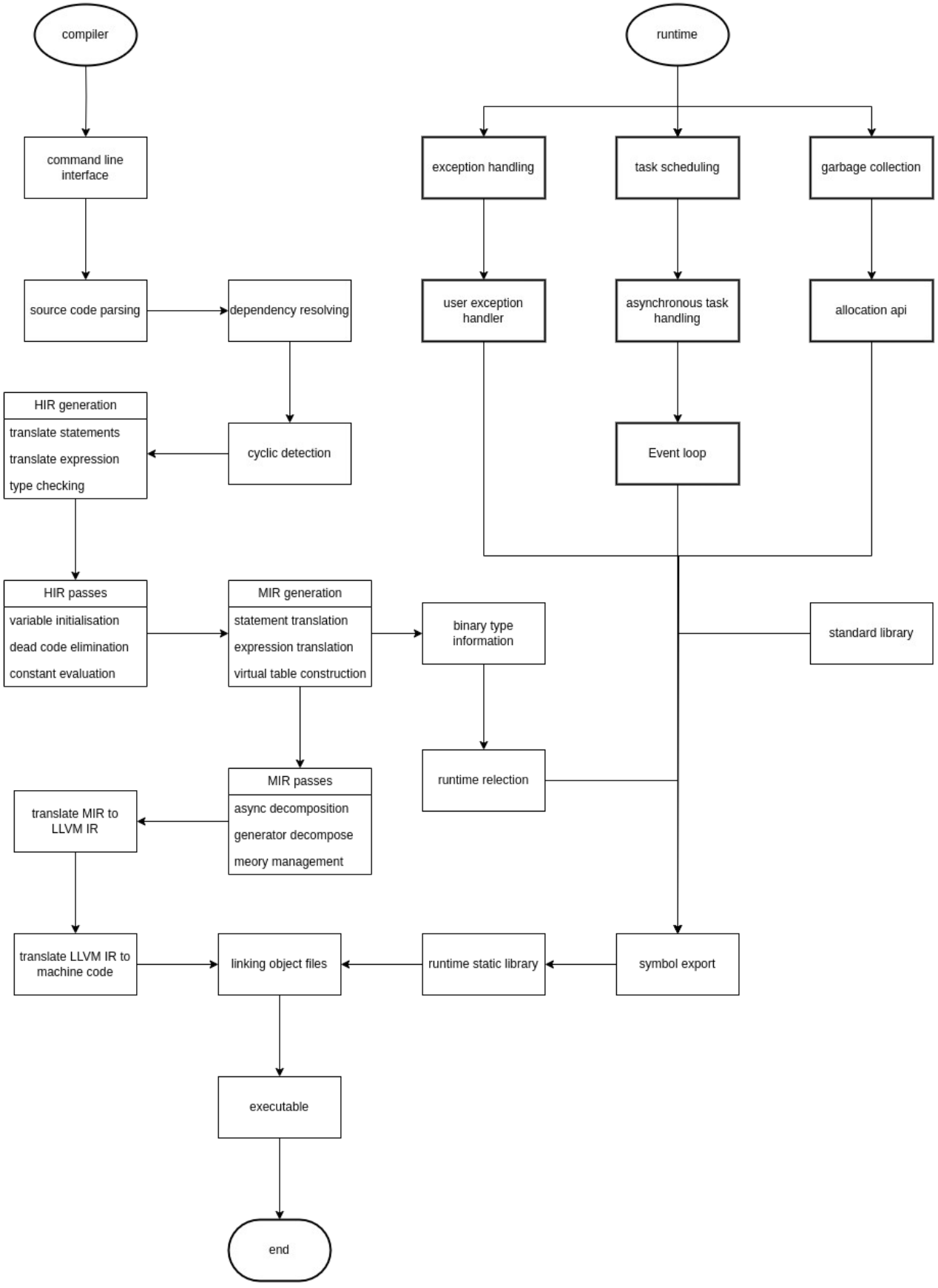


To find the EH action from the EH frame, we must look into the frame structurally. We first get an encoding from the beginning of the frame. This encoding will tell us whether the next field which is the base address of the landing pad. It then reads the type encoding and the type length. It then reads the call site encoding and the call site table length. It then loops through the frame to read call site ranges and its corresponding EH action. If an EH action is found, it is immediately returned. Otherwise, action None is returned.



2.10 Fitting the modules together

Below is a diagram showing how the modules all fit together:



As the diagram illustrated above, all the modules can be fitted together.

All the modules are essential to the compiler. The compiler would not function with any single module missing.

The compiler first enters the command line interface receiving commands from the user. It then parses the source code given by the user. It then tries to resolve the dependencies and also parse it. After parsing all the dependencies, cyclic detection algorithm is ran to check for cyclic dependencies. The AST generated by the parser is then translated into high-level IR by translating statements, expressions and performing type checks. Some passes then applies to the HIR code generated. This includes dead code elimination, variable initialisation check and constant evaluation.

The HIR is then translated into MIR by translating its statements and expression. Interfaces defined in HIR are translated into virtual table structures in MIR. The MIR code then goes through several passes including asynchronous function decomposition, generator function decomposition and memory management integration.

The MIR is then translated into LLVM IR and fed to the LLVM compiler backend. LLVM would generate machine code and write them into object files.

The runtime has three main components: exception handling, task scheduler and the garbage collector. These three components are independent of each other. However they are all included in the runtime because the compiler generated code depends on them. The three components of the runtime is exported as functions to the public symbols as a static library. The runtime make use of the binary type information generated by the compiler to perform type reflection for the user.

The compiler and the runtime is linked together at the object file linking phase. The runtime is pre-compiled as a static library and will be provided to the linker to link against user generated codes in object files. The final executable will include the runtime and the user executable code generated by the compiler.

2.11 Key variables and data structures

Type	name	Data type	description	justification
structure	Source map	Raw bytes	The source map is responsible for storing source files in text format.	The location of source code may be referenced later if warning or error is emitted.
structure	Parser	composite	The Parser is a structure that stores the source map, the parsed AST modules and the syntax errors found during parsing. It also stores metadata used while parsing.	The contextual data used during parsing has to be stored somewhere. The parsed modules must also be stored.
function	Parse string	/	This function takes in source codes as a string located somewhere in the source map. It tries to parse the source code into AST. Syntax error is emitted to the Parser structure when encountered.	Multiple source code file may be parsed in a project. This function can be reused to parse multiple files.
function	Resolve dependency	/	This function takes in a string and tries to figure out what the dependency is. If the dependency cannot be found from either local file or the internet, an error would be returned	Multiple dependencies may be required by a module. This function can be reused to import dependencies.
structure	Dependency graph	graph	This is a graph structure that stores the relationship between modules.	This is used to perform dependency analysis and cyclic dependency detection.

function	find cyclic dependency	/	This function is recursive. It performs a depth first search on the Dependency graph to find cyclic dependencies. An error with the dependency chain is returned if detected.	The DFS algorithm implemented is recursive. This function calls itself to search in child nodes.
structure	Symbol table	map	This structure stores symbols and their corresponding values. It is used to identify symbols in a programme.	The compiler can lookup for symbols in a symbol table during compilation.
alias	Variable ID	integer	This is a unique identifier that points to a variable symbol in the symbol table.	By storing identifiers instead of names, the memory usage can be reduced.
alias	Class ID	integer	This is a unique identifier that points to a class definition in the symbol table.	Same as above
alias	Interface ID	integer	This is a unique identifier that points to an interface definition in the symbol table.	Same as above
alias	Function ID	Integer	This is a unique identifier that points to a function definition in the symbol table.	Same as above
Structure	HIR program	composite	This structure stores all the parsed HIR modules and a global symbol table. It represents all of the user’s source code along with its dependencies.	The translated HIR modules have to be stored somewhere.
structure	HIR module	composite	This structure stores a list of its dependencies, all its statements and its exported symbols.	It stores information of a translated module.

structure	HIR statement	composite	This structure represents a statement in the source code. It is both a enum and a structure at the same time. It can store required information for the statement to function such as labels, condition expressions etc.	Statements in the source code should be represented in HIR.
structure	HIR expression	composite	This structure represents an expression in the source code. An expression is in fact a tree structure. An expression can be an operation or a value. The operands of an operation are also expressions.	Expressions in the source code must be represented in some way in HIR.
structure	HIR type	composite	This structure represents a type in Typescript. This is used to store type informations for values. This is also used to perform type checks	Types in Typescript must be represented in HIR for compiling purposes.
function	Construct union	/	This function takes in two HIR types and return a single type. In some case, one type may have override another. Otherwise a union type is returned.	Users may defined union types in source codes. A function to guide the construction for unions is needed.
function	Type compatible	/	This function takes two HIR types and return a boolean. It checks whether the right hand type can be assigned to the left hand type. If so, it returns true otherwise false. A different type can be compatible with each other for example an object and an interface.	This allows the compiler to perform type checking.

structure	HIR context	composite	This structure stores information about the translating source code. This is a temporary storage to store information about scopes during translation from AST to HIR	Scope information about source code is needed during translation.
structure	HIR scope	composite	This data structure stores information about a programming scope during translation. It stores information about the symbol bindings, scope labels and other data to assist the translation process.	Scope information about source code is needed during translation.
function	Hoist statement	/	This function performs statement hoisting	Hoisting must be done beforehand to met the Typescript specification
function	Hoist class	/	This function performs class hoisting.	This allows classes to be visible to the programme.
function	Hoist interface	/	This function performs interface hoisting.	This allows interfaces to be visible to the programme.
function	Translate module	/	This function translates a module. It will iterate on the module’s dependencies to make sure that they are translated. It will iterate over the module statements and translate them.	Modules of AST needed to be translated to HIR
function	Translate statement	/	This function translates an AST statement into an HIR statement.	Statements of AST must be translated to HIR.

function	Translate declare statement	/	This function translates declaration statements, this includes variables, classes, interfaces, enums and type alias declaration.	Statements of AST must be translated to HIR.
function	Translate block statement	/	This function will translate a block statement in AST into HIR. It will create a new scope in the context and translate its child statements. Upon exit, it will close the scope.	Statements of source code must be translated to HIR.
function	Translate while loop statement	/	This function translates a while loop statement.	Typescript have while loops
function	Translate for loop statement	/	This function translates a for loop statement	Typescript have for loops
function	Translate break statement	/	This function translates a break statement. It will check if current context is breakable.	Typescript have break statements
function	Translate conditional statements	/	This function translates a conditional statement	Typescript have conditional statements

2.12 testing plans

Testing would be done during the development phase and post-development phase.

Individual would have unit tests to make sure that every module works. The unit tests would have both valid and invalid test cases to make sure the components can function in both cases.

We do not have to perform any unit testing for the parser since we are using a third-party library that have been fully tested already.

After the modules are developed and put together, a set of black box testing would be performed to test the overall programme.

The black box test would be performed in two ways: From source code to HIR and from source code to MIR. This is because we fully trust the source code parser that is a third party.

Unit test: parsing configuration file

input	Justification	type
[project] name = "hello world" version = "0.1.0" [features] my_feature = []	User defined feature name should be accepted	Valid
[project] name = "hello world" version = "0.2.1" [lib] lib-type = “static”	lib_type is renamed into lib-type in serde	Valid
[project] name = "hello world" version = "0.2.1" [dependencies.my_lib] url = "mylib.com/lib" optional = true	User defined dependency name comes after dependencies	Valid
[project] name = "win" version = "0.1.1" [target.windows.dependencies.winapi] path = "winapi"	Target specific dependencies	Valid
[project] name = "win" version = 2 [profile.debug] opt-level = 3	The version must be a string	Invalid
name = "win" version = "0.1.1" [profile.release]	Missing the compulsory profile section	Invalid

Unit test: cyclic detection

Test case	expected	Justification	type
0 => (1, 2, 3), 1 => (5), 2 => (3, 4, 5), 3 => (4, 5), 4 => (), 5 => ()	No cycles detected	No cycle is presented.	Valid
0 => (1, 2), 1 => (2), 2 => (0, 1)	Cycles detected: 0 → 1 → 2 → 0	Simple cyclic dependencies.	Invalid
0 => (1, 2), 1 => (2), 2 => (3, 4), 3 => (1), 4 => ()	Cycles detected: 1 → 2 → 3 → 1	Multiple cycles	Invalid
0 => (), 1 => (), 2 => (3), 3 => (2)	Cycles detected: 2 → 3 → 2	Cyclic dependency not in root	Invalid
0 => (2, 3, 4, 5), 1 => (), 2 => (), 3 => (), 4 => (3, 2), 5 => (3, 2)	Cycles not detected	Shared dependencies	Valid
0 => (4, 5), 1 => (2), 2 => (), 3 => (1), 4 => (2, 5), 5 => (1, 3)	Cycles not detected	Complex dependency graph	Valid

Unit test: construction of union:

The reason that there is no invalid testing is because all types can be united together.

Input1: left	Input2: right	justification	Type
null	null	Same type	Valid
Any	undefined	Any type will override union	Valid
integer	number	Integer should be treated as number in a union	Valid
Number	integer	Reverse order of number and integer from the above test	Valid
integer	object	Testing random types together	Valid
Any Object	object	Any object should override object type	Valid
Any Object	interface	An interface may not be an object, they must form a union	Valid
Any Object	Any	Any will override any object	Valid
Any Object	symbol	Testing random types together	Valid
Null union interface	null	The left hand side is a union that contains right-hand side	Valid
Object union interface	object	If the object fulfils requirement of interface, the interface should override	Valid
Null union symbol union number	Null union Number	The left-hand side union includes the right-hand side union	Valid
number	bigint	Some random testing	Valid
interface	boolean	Some random testing	Valid

Unit test: conditional types

input	Expected output	justification	type
number extends number? number: string	number	Condition have the same type therefore should always be true	valid
number extends string? number: string	string	Number is not the same as string and therefore condition is false	valid
number extends any? number : string	number	Number is compatible with any and therefore condition is true	valid
{} extends Object? any: null	any	All object type inherits Object therefore condition is true	valid
null extends Object? any: null	null	Null is not an Object therefore condition is false	valid
number extends object | number ? number: null	number	The union contains number therefore condition is true	valid

Unit test: index access types

Object type	Index type	Expected output	Justification	Type
string[]	number	string	Number index type on an array should return element type	Valid
{a:number}	“a”	number	String literal index should return the property type	Valid
[null, string]	1	string	Number literal index should return the element type at index	Valid
any	number	error	Any type have no properties and should return an error	Invalid
{}	99n	error	Big integer cannot be used as index	Invalid
[]	{o:3}	error	Object type cannot be used as index	Invalid

Unit test: key of operator

input	expected	justification	type
{a:any}	"a"	Object has a single key	valid
{}	undefined	Object has no keys therefore undefined	valid
any	string | number | symbol	Any type have dynamic keys	valid
number	"toString" | "toLocaleString" | "toFixed" | "toExponential" | "toPrecision" | "valueOf"	Key of operator should return built in property strings	valid
undefined	never	Undefined type does not have any key.	valid
object	never	Any object does not have any property.	valid
[]	number	Array should have numeric keys	valid
[object, any]	0, 1	Tuple should have integer keys	valid

Unit test: translating function types

input	This type	parameters	Return type	Justification	type
(number) => number	any	number	number	Simple types	valid
(this: object, string) => undefined	object	string	undefined	The 'this' type is set.	valid
(string, any) => any	any	String, any	any	Multiple parameter	valid
(string, {a}: {a:number},) => any	any	{a:number}	any	Destructive parameter	valid
(a?:number) => number	any	Number | undefined	number	Optional parameter converts to union	valid
() => object	any	-	object	No parameter	valid
([a, b]:[null, number]) => string	any	[null, number]	string	Destructive parameter	valid

Unit test: translate variable declare

input	Justification	Type
`let a = 0;`	Compiler should able to reference initialiser type if no type annotation.	valid
`let a: string = 0;`	Number type is not assignable to string type.	valid
`let a = 0, b: string = "0"`	Multiple declarator	valid
`var a = 0;` `var a = 9;`	Variable with var declaration can be redeclared	valid
`var a: number = 6;` `var a: string = "6";`	Redeclared variable must have the same type	invalid
`let a = 0;` `let a = 8;`	‘let’ declared variables cannot be redeclared	invalid
`const a = 0;` `const a = 8;`	‘const’ declared variables cannot be redeclared	invalid
`var a;`	Variable declaration should have type annotation or initialiser	invalid
`const a: number;`	Constant declaration must have initialiser	invalid
`var [a, b] = [0, 9];`	Compiler should be able to reference element types in destructive assignment	valid
`var [a, b] = [0];`	Although initialiser only has one element, this is allowed because it is seen as an array instead of a tuple because there is no type annotation.	valid
`let [a, b]:[number, string] = [0, ""];`	Destructive assignment with type annotation.	valid

Unit test: translate for loop

input	Expected output	Justification	Type
for(;;){ }	for(;;){ }	Simple for loop	valid

for(let i=0;;){
}
	let var4:number
	var4=0 as number
	for (;;){
	}
		For loop with initialiservalid	valid

let i=0
for(;i<100;){
 	
}
	let var4:number
	var4=0 as number
	for (;;){
	if (!((var4)<(100 as number))){
	break
	}
	}
		For loop with break condition	valid

let i = 0;
for (;;i++){
 	
}
	let var4:number
	var4=0 as number
	for (;;var4++){
	}
		For loop with update expression	valid

for (var i=0;i<10;)
{
}
	var var4:number
	var4=0 as number
	for (;;){
	if (!((var4)<(10 as number))){
	break
	}
	}
		For loop with initialiser and break condition	valid

for (var i=0;i<10;i++){
 	
}
	var var4:number
	var4=0 as number
	for (;;var4++){
	if (!((var4)<(10 as number))){
	break
	}
	}
		For loop with initialiser, break condition and update expression	valid

Black box testing: from source code to HIR

This is a black box testing planned to test that HIR is correctly generated from source code.

input	justification	type

`for (let i in []){`
`  i += (99)`
`}`
	For-in loops are translated into simple loops with its iterator represented as a counter.	Valid

`for (let i of []){`
`  i += (99)`
`}`
	For-of loops are translated into simple loops with its iterator represented as a counter and an index access to the element.	Valid

`let i = 0;`
`while (i < 100){`
`  i++;`
`}`
	While loops are translated into simple loops with breaking condition compared at the beginning of iteration.	Valid

`let t = {i:0, u: "i"};` `t.o = 9;`	Testing type checks. Object in variable ‘t’ has no property ‘o’	Invalid

`interface A{`
`  a: number;`
`}`
 	
`interface B{`
`  a: string`
`}`
 	
`type U = A & B;`
	Syntax check. HIR should be able to declare interfaces. It should also be able to perform union construction. It should also be able to declare type aliases.	Valid

`class Vehicle{`
`  weight?: number;`
`}`
 	
`class Car extends Vehicle{`
`constructor(a:boolean){`
`  if (a){`
`  super()`
`  }`
`}`
`}`
	Testing HIR validation of constructor. Because a condition statement is used to initialise the parent class ‘Vehicle’, there may be a possibility that the parent class is not initialised before constructor returns.
	This should result in an error.
		Invalid

`let arr = [10, 80, 30, 90, 40];`	Testing array construction, compiler should be able to reference element types as number.	valid

`let a = 0;`
`for (let i = 0; i < 100;i++)`
`{`
`for (let j=0; j < 100;j++){`
`  a++;`
`}`
`}`
	Stacked loops	valid

`function recurring(){`
`  for (let i=0;i< 100;i++){`
`  recurring()`
`  }`
`}`
	Recurring function in a loop context	valid

`type recurring_func = typeof weird_recurring;`
 	
`function weird_recurring(f: (f:recurring_func) => undefined): undefined{`
`f(weird_recurring);`
`}`
 	
`weird_recurring(weird_recurring);`
	Cyclic type referencing	invalid

Block scope testing

input	Justification	type

try {
  (function(x) {
  try {
  let x = 'inner';
  throw 0;
  } finally {
  assert.sameValue(x, 'outer');
  }
  })('outer');
} catch (e) {}
	finally block let declaration only shadows outer parameter value 1	valid

(function(x) {
  try {
  let x = 'middle';
  {
  let x = 'inner';
  throw 0;
  }
  } catch(e) {
  } finally {
  assert.sameValue(x, 'outer');
  }
})('outer');
	finally block let declaration only shadows outer parameter value 2	valid

(function(x) {
  for (var i = 0; i < 10; ++i) {
  let x = 'inner' + i;
  continue;
  }
  assert.sameValue(x, 'outer');
})('outer');
	for loop block let declaration only shadows outer parameter value 1	valid

(function(x) {
  label: for (var i = 0; i < 10; ++i) {
  let x = 'middle' + i;
  for (var j = 0; j < 10; ++j) {
  let x = 'inner' + j;
  continue label;
  }
  }
  assert.sameValue(x, 'outer');
})('outer');
	for loop block let declaration only shadows outer parameter value 2	valid

(function(x) {
  label: {
  let x = 'inner';
  break label;
  }
  assert.sameValue(x, 'outer');
})('outer');
	nested block let declaration only shadows outer parameter value 1	valid

(function(x) {
  label: {
  let x = 'middle';
  {
  let x = 'inner';
  break label;
  }
  }
  assert.sameValue(x, 'outer');
})('outer');
	nested block let declaration only shadows outer parameter value 2	valid

var caught = false;
try {
  {
  let xx = 18;
  throw 25;
  }
} catch (e) {
  caught = true;
  assert.sameValue(e, 25);
  (function () {
  try {
  // NOTE: This checks that the block scope containing xx has been
  // removed from the context chain.
  assert.sameValue(xx, undefined);
  eval('xx');
  assert(false); // should not reach here
  } catch (e2) {
  assert(e2 instanceof ReferenceError);
  }
  })();
}
assert(caught);
	outermost binding updated in catch block; nested block let declaration unseen outside of block	valid

function f() {}
 	
(function(x) {
  try {
  let x = 'inner';
  throw 0;
  } catch(e) {
 	
  } finally {
  f();
  assert.sameValue(x, 'outer');
  }
})('outer');
	verify context in finally block 1	valid

function f() {}
 	
(function(x) {
  for (var i = 0; i < 10; ++i) {
    let x = 'inner';
    continue;
  }
  f();
  assert.sameValue(x, 'outer');
})('outer');
	verify context in for loop block 2	valid

function f() {}
 	
(function(x) {
  label: {
    let x = 'inner';
    break label;
  }
  f(); // The context could be restored from the stack after the call.
  assert.sameValue(x, 'outer');
})('outer');
	verify context in labelled block 1	valid

function f() {}
 	
(function(x) {
  try {
    let x = 'inner';
    throw 0;
  } catch (e) {
    f();
    assert.sameValue(x, 'outer');
  }
})('outer');
	verify context in try block 1	valid

3.13 Post development testing: running real world algorithms

Test case: binary search

Justification: binary search is a commonly used algorithm

Expected output: 5

```
function round(i: number): number{
    let rem = i % 1;
    let n = i - rem;
    if (rem >= 0.5){
        return n + 1
    }
    return n
}
function binarySearch(arr: number[], x: number): number
{
    let l = 0;
    let r = arr.length - 1;
    let mid: number;
    while (r >= l) {
        mid = l + round((r - l) / 2);
        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;
        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            r = mid - 1;

        // Else the element can only be present
        // in right subarray
    else
        l = mid + 1;
    }
    // We reach here when element is not
    // present in array
    return -1;
}
binarySearch([0, 8, 9, 10, 11, 12, 13, 14], 12);
```

Test case: ternary search

Justification: ternary search is a commonly used algorithm

Expected output: 6

```
function ternarySearch(l: number, r: number, key: number, ar: number[]):
number {
    if (r >= l) {

        // Find the mid1 and mid2
        let mid1 = l + round((r - l) / 3);
        let mid2 = r - round((r - l) / 3);

        // Check if key is present at any mid
        if (ar[mid1] == key) {
            return mid1;
        }
        if (ar[mid2] == key) {
            return mid2;
        }

        // Since key is not present at mid, check in which region it is present
        // then repeat the Search operation in that region
        if (key < ar[mid1]) {
            // The key lies in between l and mid1
            return ternarySearch(l, mid1 - 1, key, ar);
        }
        else if (key > ar[mid2]) {
            // The key lies in between mid2 and r
            return ternarySearch(mid2 + 1, r, key, ar);
        }
        else {
            // The key lies in between mid1 and mid2
            return ternarySearch(mid1 + 1, mid2 - 1, key, ar);
        }
    }

    // Key not found
    return -1;
}
let arr = [2, 3, 6, 9, 10, 11, 13, 17, 23];
let search_item = 13;
ternarySearch(0, arr.length - 1, search_item, arr)
```

Test case: fibonacci search

Justification: fibonacci search is a common search algorithm

data: [10, 22, 35, 40, 45, 50, 80, 82,85, 90, 100,235] search target: 235

Expected output: 11

```
function fibMonaccianSearch(arr: number[], n: number, x: number) :number {
   /* Initialize fibonacci numbers */
   let fibMMm2 = 0; // (m-2)'th Fibonacci No.
   let fibMMm1 = 1; // (m-1)'th Fibonacci No.
   let fibM = fibMMm2 + fibMMm1; // m'th Fibonacci
/* store the smallest Fibonacci Number greater than or equal to n */
   while (fibM < n) {
      fibMMm2 = fibMMm1;
      fibMMm1 = fibM;
      fibM = fibMMm2 + fibMMm1;
   }
   // Marks the eliminated range from front
   let offset = -1;
   while (fibM > 1) {
      // Check if fibMm2 is a valid location
      let i = Math.min(offset + fibMMm2, n-1);
      if (arr[i] < x) {
         fibM = fibMMm1;
         fibMMm1 = fibMMm2;
         fibMMm2 = fibM - fibMMm1;
         offset = i;
      }
      /* If x is less than the value at index fibMm2, cut the subarray after i+1 */
      else if (arr[i] > x) {
         fibM = fibMMm2;
         fibMMm1 = fibMMm1 - fibMMm2;
         fibMMm2 = fibM - fibMMm1;
      }
      /* element found. return index */
      else return i;
   }
   /* comparing the last element with x */
   if(fibMMm1 && arr[n-1] == x){
      return n-1
   }
   /*element not found. return -1 */
   return -1;
}
```

Test case: Bubble sort

Justification: bubble sort is a commonly used algorithm

Expected output: [5, 6, 43, 55, 63, 234, 235, 547]

```
function bubbleSort(arr: number[]) {
    for (var i = 0; i < arr.length; i++) {
    // Last i elements are already in place
        for (var j = 0; j < (arr.length - i - 1); j++) {
            // Checking if the item at present iteration
            // is greater than the next iteration
            if (arr[j] > arr[j + 1]) {
                // If the condition is true
                // then swap them
                var temp = arr[j]
                arr[j] = arr[j + 1]
                arr[j + 1] = temp
            }
        }
    }
    // Print the sorted array
    console.log(arr);
}
// This is our unsorted array
var arr = [234, 43, 55, 63, 5, 6, 235, 547];
// Now pass this array to the bblSort() function
bubbleSort(arr);
```

Test case: quick sort

Justification: quick sort is a commonly used algorithm

Expected output: [10, 30, 40, 80, 90]

```
function partition(arr: number[], low: number, high: number): number {
    let pivot = arr[high];
    let i = low - 1;
    for (let j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
            // Increment index of smaller element
            i++;
            // Swap elements
            [arr[i], arr[j]] = [arr[j], arr[i]];
        }
    }
    // Swap pivot to its correct position
    [arr[i + 1], arr[high]] = [arr[high], arr[i + 1]];
    return i + 1; // Return the partition index
}
function quickSort(arr: number[], low: number, high: number) {
    if (low >= high) return;
    let pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
}
let arr = [10, 80, 30, 90, 40];
quickSort(arr, 0, arr.length - 1);
```

Test case: insertion sort

Justification: insertion sort is a commonly used algorithm

expected input: [3, 3, 4, 4, 6, 9, 12]

```
// Function to implement insertion sort
function insertionSort(arr: number[]) {
    // Getting the array length
    let n = arr.length;
    // To store value temporarily
    let key: number = 0;
    // For iterations
    let j: number = 0;
    // Iterate array in forward direction
    for (let i = 0; i < n ; ++i) {
        key = arr[i];
        j = i - 1;

        // Iterate and swap elements in backward direction
        // till number is greater then the key
        for (j; j >= 0 && arr[j]>key; --j){
            arr[j+1]=arr[j];
        }
        // Swap the key to right position
        arr[j+1]=key;
    }
}

insertionSort([9,6,3,4,3,12,4])
```

Test case: simple hash

Justification: hashing algorithms are common in programmes

Expected output: -429545180

```
function simple_hash(arr: number[]){
    var hash = 0;
    for (var i = 0; i < arr.length; i++) {
        var char = arr[i];
        hash = ((hash<<5)-hash)+char;
        hash = hash & hash; // Convert to 32bit integer
    }
    return hash;
}

simple_hash([0,9,6,1,5,9,8,4]);
```

3.14 post development: usability testing

Interview

index	question	Justification
1.	Do you think the command line interface is easy to use	Ask the user how they feel about the command line interface in general
2.	Do you find the command line helper useful	Ask the user how they fell about the helper in general
3.	What changes do you think should be added to the appearance of the command line interface	Ask about what they think should be changed in the interface
4.	What changes do you think should be added to the functionality of the command line interface	Ask about what functionalities they would like to be included in the interface
5.	Is the description clear in the command line helper	Ask about whether they find the descriptions clear.
6.	Do you think the error messages are clear	Ask about the formatting of the error messages
7.	Do you think the warning messages are clear	Ask about the formatting of the warning messages
8.	What do you think about the appearance of the warning and error messages	Ask about the appearance of formatting of messages and how to improve them

Part 3
Implementation and Testing

3.1 Parsing

3.1.1 Parsing configuration file

The crate serde and toml is used for parsing configuration files. By combining these two crates, the parse can be done easily just by defining a structure. We use the derive macro provided by the serde crate and the deserialiser provided by the toml crate to parse toml configuration file.

The derive macro will automatically detect structure declarations and generate a suitable method to represent the structure in a generic representation. The TOML crate supports serialising and de-serialising from the serd format. This means that we do not have to write any code for parsing the toml files. The macro will generate for us.

The configuration structure provides the entry point to the configuration file

```
#[derive(Debug, Serialize, Deserialize)]
pub struct Config{
        pub project: ProjectConfig,
        #[serde(default)]
        pub lib: LibConfig,
        #[serde(default)]
        pub bin: BinConfig,
        #[serde(default)]
        pub dependencies: Dependencies,
        #[serde(default)]
        pub target: Target,
        #[serde(default)]
        pub features: Features,
        #[serde(default)]
        pub profile: ProfileConfig
}
```

the project section:

```rust
#[derive(Debug, Serialize, Deserialize)]
pub struct ProjectConfig{
        pub name: String,
        pub version: String,
        pub authors: Option<Vec<String>>,
        #[serde(rename="compiler-version")]
        pub compiler_version: Option<String>,
        #[serde(rename="ts-version")]
        pub ts_version: Option<String>,
        pub description: Option<String>,
        pub documentation: Option<String>,
        pub readme: Option<String>,
        pub homepage: Option<String>,
        pub repository: Option<String>,
        pub licence: Option<String>,
        #[serde(rename = "licence-file")]
        pub licence_file: Option<String>,
        pub keywords: Option<Vec<String>>,
        pub categories: Option<Vec<String>>,
        pub exclude: Option<Vec<String>>,
}
```

The lib section:

```rust
#[derive(Debug, Default, Serialize, Deserialize)]
pub struct LibConfig{
        #[serde(default)]
        pub test: bool,
        pub lib_type: Option<String>,
        pub features: Option<Vec<String>>
}
```

The bin section:

```rust
#[derive(Debug, Default, Serialize, Deserialize)]
pub struct BinConfig{
        #[serde(default)]
        pub test: bool,
        pub features: Option<Vec<String>>
}
```

Key/ value pair of dependencies:

```
pub type Dependencies = HashMap<String, Dependency>;
```

Configuration of each dependency:

```
#[derive(Debug, Serialize, Deserialize)]
pub struct Dependency{
	pub path: Option<String>,
	pub url: Option<String>,
	pub git: Option<String>,
	pub version: Option<String>,
	pub features: Option<Vec<String>>,
	#[serde(default)]
	pub optional: bool,
}

#[derive(Debug, Default, Serialize, Deserialize)]
#[serde(default)]
pub struct Target{
	pub windows: TargetConfig,
	pub unix: TargetConfig,
	pub linux: TargetConfig,
	pub darwin: TargetConfig,
	pub macos: TargetConfig,
	pub ios: TargetConfig,
	pub freebsd: TargetConfig,
	pub openbsd: TargetConfig,
	pub redox: TargetConfig,
	pub android: TargetConfig,
	pub x86: TargetConfig,
	pub x86_64: TargetConfig,
	pub arm: TargetConfig,
	pub aarch64: TargetConfig,
	pub riscv: TargetConfig,
	pub wasm32: TargetConfig,
}

#[derive(Debug, Default, Serialize, Deserialize)]
#[serde(default)]
pub struct TargetConfig{
	pub dependencies: Dependencies,
	pub features: Option<Vec<String>>
}
```

Key/ value pair of custom features implemented as a hashmap:

```
pub type Features = HashMap<String, Vec<String>>;
```

The profile section:

```
#[derive(Default, Debug, Serialize, Deserialize)]
#[serde(default)]
pub struct ProfileConfig{
        #[serde(default)]
        pub debug: Profile,
        #[serde(default)]
        pub release: Profile
}

#[derive(Debug, Default, Serialize, Deserialize)]
pub struct Profile{
        #[serde(rename="opt-level")]
        pub opt_level: Option<u8>,
        pub debug: Option<bool>,
        pub lto: Option<bool>,
        pub incremental: Option<bool>
}
```

3.1.2 Testing configuration file parsing

input	output	reason	pass
[project] name = "hello world" version = "0.1.0" [features] my_feature = []	[project] name = "hello world" version = "0.1.0" [features] my_feature = []	User defined feature name should be accepted	Yes
[project] name = "hello world" version = "0.2.1" [lib] lib-type = “static”	[project] name = "hello world" version = "0.2.1" [lib] lib-type = “static”	lib_type is renamed into lib-type in serde	Yes

[project] name = "hello world" version = "0.2.1" [dependencies.my_lib] url = "mylib.com/lib" optional = true	[project] name = "hello world" version = "0.2.1" [dependencies.my_lib] url = "mylib.com/lib" optional = true	User defined dependency name comes after dependencies	Yes
[project] name = "win" version = "0.1.1" [target.windows.dependencies.winapi] path = "winapi"	[project] name = "win" version = "0.1.1" [target.windows.dependencies.winapi] path = "winapi"	Target specific dependencies	Yes
[project] name = "win" version = "0.1.1" [profile.debug] opt-level = 3	[project] name = "win" version = "0.1.1" [profile.debug] opt-level = 3	Setting opt-level on debug profile	Yes
name = "win" version = "0.1.1" [profile.debug] opt-level = 3	error	Missing section profile	Yes
[project] name = "project" version = 2	error	The field ‘version’ must be a string	Yes
[project] name = "win" version = "0.1.1" [lib] path = “./myfile”	error	Section ‘lib’ does not have field ‘path’	Yes

With the above testing all passing, I can conclude that there are no bugs in parsing configuration file. This is mostly because the parser is generated using macros and no direct code is written to parse the file.

3.1.3 Parsing source code

The structure ParsedModule is returned after parsing a module.

```
#[derive(Debug)]
pub struct ParsedModule{
	/// the canonicalised name
	pub path: PathBuf,
	/// the unique module identifier
	pub id: ModuleId,
	/// dependencies of the module
	pub dependencies: Vec<ModuleId>,
	/// AST of the parsed module
	pub module: swc_ecmascript::ast::Module,
}
```

The structure ParsedProgram contains all the modules parsed.

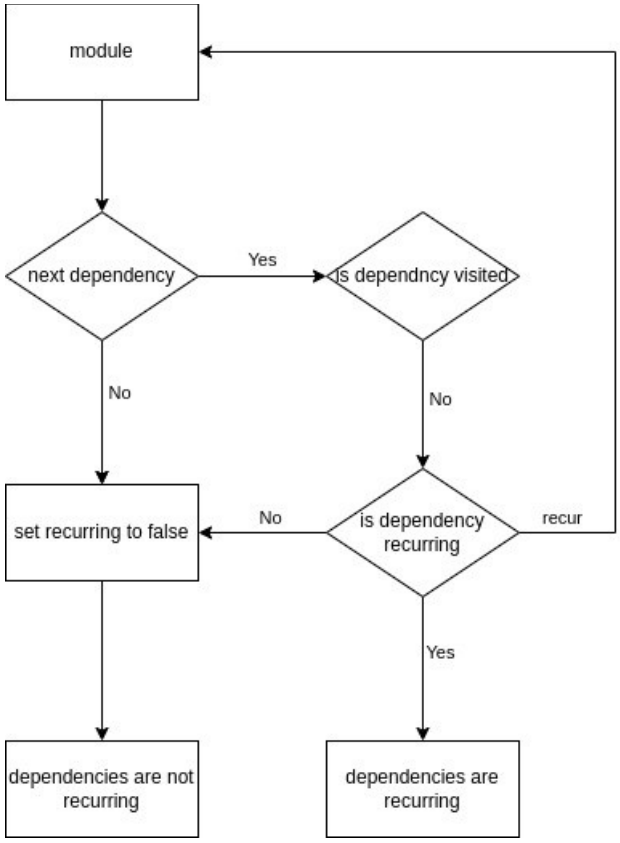
```
/// stores every module in a program
#[derive(Debug)]
pub struct ParsedProgram{
	/// hash map stores the unique id and the parsed module
	pub modules: HashMap<ModuleId, ParsedModule>,
}
```

The structure Parser is used when parsing. It contains a source map and the modules being parsed. The source map is used to store source files and debug information.

```
/// a state structure that stores a source code map and the parsed modules
#[derive(Default)]
pub struct Parser{
	/// stores source code files and their relative paths
	src_map: swc_common::SourceMap,
	/// stores the parsed modules
	modules: HashMap<ModuleId, ParsedModule>
}
```

3.1.4 Cyclic detection

Cyclic detection is done after parsing modules. The following methods is implemented for struct Parser.



```rust
fn check_cyclic_dependency(&self) -> Result<(), String> {
	// allocate visited stack
	let mut visited = Vec::with_capacity(self.modules.len());
	// allocate recurring stack
	let mut rec_stack = Vec::with_capacity(self.modules.len());

	// set all value to false
	visited.resize(self.modules.len(), false);
	rec_stack.resize(self.modules.len(), false);

	// loop over every module
	for id in self.modules.keys() {
		// not visited
		if !visited[id.0]
		&& self.is_cyclic_until(*id, &mut visited, &mut rec_stack) {
			// cyclic dependency detected
			let mut msg = "cyclic dependency detected: ".to_string();

			// format the error message
			// find any module that is recurring
			for (i, recurring) in rec_stack.into_iter().enumerate() {
				// is recurring,
				if recurring {
					msg.push_str(
						&self
						.modules
						.get(&ModuleId(i))
						.unwrap()
						.path
						.to_string_lossy(),
					);
					msg.push_str(" -> ");
				}
			}
			/// return the error
			return Err(msg);
		}
	}
	return Ok(());
}
```

The function below is a recurring function to perform depth first search. Dependencies of the module are call upon if not already visited. The length of slice of **visited** and **rec_stack** must be the same. The **rec_stack** and **visited** of the current module will be set to true when traversed. The **rec_stack** is set to false once all dependencies are traversed and no cycles are detected indicating that the child nodes of current module does not contain cycles and so as the node.

```
/// a recurring function that loops through every node
fn is_cyclic_until(&self, id: ModuleId, visited: &mut [bool], rec_stack: &mut
[bool]) -> bool {
        // node not visited
        if !visited[id.0] {
                // set visited
                visited[id.0] = true;
                // set recurring
                rec_stack[id.0] = true;

                // visit every dependency
                for dep in &self.modules.get(&id).unwrap().dependencies {
                        // dependency not visited, check its dependencies
                        if !visited[dep.0]
                        && self.is_cyclic_until(*dep, visited, rec_stack) {
                                /// cyclic detected
                                return true;
                        } else if rec_stack[dep.0] {
                                // dependency is recurring, cyclic detected
                                return true;
                        }
                }
        }
        // set recurring to false on exit
        rec_stack[id.0] = false;
        /// not cyclic
        return false;
}
```

3.1.5 Testing Cyclic detection

To test the validity of cyclic detection, a set of tests cases are carried out. A macro rule is defined to ease the construction of test case.

```
#[cfg(test)]
macro_rules! test_case {
   ($($id:expr => ($($dep:expr),*)),*) => {
      // construct Parser
      {let mut case = Parser{
         src: Default::default(),
         modules: Default::default()
      };
      $(
         // insert id to hashmap
         case.modules.insert(
            // module id
            ModuleId($id),
            // a dummy module
            ParsedModule{
               // Path is same as module id
               path: PathBuf::from(stringify!($id)),
               // the module id
               id: ModuleId($id),
               // add the dependencies
               dependencies: vec![$(ModuleId($dep)),*],
               // dummy ast
               module: swc_core::ecma::ast::Module{
                  span: Default::default(),
                  body: Vec::new(),
                  shebang: None
               }
            }
         );
      )*
      case}
   };
}
```

The above macro accepts inputs with the form **id => (dependencies)** and generates a parser with dummy modules represented with the provided id.

input	output	Pass
0 => (1, 2, 3), 1 => (5), 2 => (3, 4, 5), 3 => (4, 5), 4 => (), 5 => ()	No cycles detected	Yes
0 => (1, 2), 1 => (2), 2 => (0, 1)	Cycles detected: 0 → 1 → 2 → 0	Yes
0 => (1, 2), 1 => (2), 2 => (3, 4), 3 => (1), 4 => ()	Cycles detected: 1 → 2 → 3 → 1	Yes
0 => (), 1 => (), 2 => (3), 3 => (2)	Cycles detected: 2 → 3 → 2	Yes
0 => (2, 3, 4, 5), 1 => (), 2 => (), 3 => (), 4 => (3, 2), 5 => (3, 2)	Cycles not detected	Yes
0 => (4, 5), 1 => (2), 2 => (), 3 => (1), 4 => (2, 5), 5 => (1, 3)	Cycles not detected	Yes

The test have been conducted. With all the testing above passing the test, I can conclude that there are no bugs in the cyclic detection implementation.

3.2 HIR type representations

In HIR, types are represented using a enum. User defined types are stored on a table and a unique id is generated corresponding to the user defined type.

To ensure the uniqueness of the IDs, a global counter is used. The counter is updated atomically every time an ID is created.

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct ClassId(pub(super) usize);

impl ClassId {
	pub fn new() -> Self {
		static COUNT: AtomicUsize = AtomicUsize::new(0);
		Self(COUNT.fetch_add(1, Ordering::SeqCst))
	}
}
```

The above function is also implemented for InterfaceID, EnumID, FunctionID, AliasID and VarID.

Function types are not stored on the table, instead, each type representation value maintains its own copy of a function type definition.

A function type is represented as follow:

```
#[derive(Debug, Clone, PartialEq, Eq, PartialOrd, Ord)]
pub struct FuncType {
	pub this_ty: Type,
	pub params: Vec<Type>,
	pub var_arg: bool,
	pub return_ty: Type,
}
```

the this_ty field corresponds to the ‘this’ binding in Typescript. In case of a constructor or a method, this_ty will be the class type where the function belongs.

The ‘var_arg’ field indicates whether the function accepts variable arguments. However variable arguments will not be implement in this project right now, it may happen in the future but we ignore this field for now.

The ‘return_ty’ may be a promise if function is an async function and may be an iterator if function is a generator.

The enum structure representing types are as follow:

```
#[repr(C)]
#[derive(Debug, Clone, PartialEq, Eq, PartialOrd, Ord)]
pub enum Type {
	Any,
	/// not number, string, boolean, bigint, symbol, null, or undefined.
	AnyObject,
	///
	Undefined,
	Null,
	Bool,
	Number,
	Int,
	Bigint,
	String,
	Symbol,
	Regex,
	Object(ClassId),
	Interface(InterfaceId),
	Function(Box<FuncType>),
	Enum(EnumId),
	Array(Box<Type>),
	Promise(Box<Type>),
	Map(Box<Type>, Box<Type>),
	Union(Box<[Type]>),
	Tuple(Box<[Type]>),
	Iterator(Box<Type>),

	Alias(AliasId),
	Generic(GenericId),
}
```

variant	description
Any	Any value, same as an interface without properties
Any Object	Any object, could be any value except number, string, boolean, bigint, symbol, null or undefined
Undefined	Undefined type, only accepts undefined value
Null	Null type, only accepts null value
Bool	Boolean type, only accepts true or false
Number	Number, a floating point number.
Integer	32bit signed integer type. This type is used when number declared in source code is certain to be integer at compile time. This type cannot be declared by user explicitly. It is implemented to speed up arithmetic performance.
BigInt	Big integer, it is implemented as a signed 128bit value.
String	String. It is represented using a structure of null terminated pointer and a usize length value.
Symbol	A symbol is a unique representation of property key. It is implemented using a 64bit hash value.
Regex	A regular expression.
Object	An object is an instance of a class.
Interface	A dynamic type that accepts any type fulfilling the requirements.
Function	A function type. Maybe a closure or a raw function
Enum	A Enum is an integer value that represents a state.
Array	An Array has dynamic length. Type of each element must be the same.
Tuple	Tuple has fixed number of elements. However, each element can have a different type.
Iterator	An Iterator follows the ECMA iterator protocol. It is essentially an interface with ‘next’ method.
Alias	An alias type represents a possible unknown type during translation. This type is only used during translation and will not be precent outside HIR generation
Generic	A generic type represents an unknown type during translation. It is replaced by user provided type arguments during translation. Generics will not be precent outside HIR generation.

3.2.1 representing classes

In a class, static properties are interpreted as global variables. Static methods are interpreted as global functions. Property Description describes the properties of a property.

Class representation is implemented as follow:

```
#[derive(Clone)]
pub struct PropertyDesc {
	pub ty: Type,
	pub readonly: bool,
	pub initialiser: Option<Expr>,
}

#[derive(Default, Clone)]
pub struct ClassType {
	pub name: String,

	pub extends: Option<ClassId>,
	pub implements: Vec<InterfaceId>,

	/// class may not have constructor
	pub constructor: Option<(FunctionId, FuncType)>,

	/// static properties are just global variables
	pub static_properties: HashMap<
		PropName,
		(VariableId, Type)>,
	/// static methods are just static functions
	pub static_methods: HashMap<
		PropName,
		(FunctionId, FuncType)>,
	/// static generic methods are just generic functions
	pub static_generic_methods: HashMap<
		PropName, (FunctionId,)>,

	pub properties: HashMap<PropName, PropertyDesc>,
	pub methods: HashMap<PropName, (FunctionId, FuncType)>,
	/// not used
	pub generic_methods: HashMap<PropName, (FunctionId,)>,
}
```

3.2.2 representing interfaces

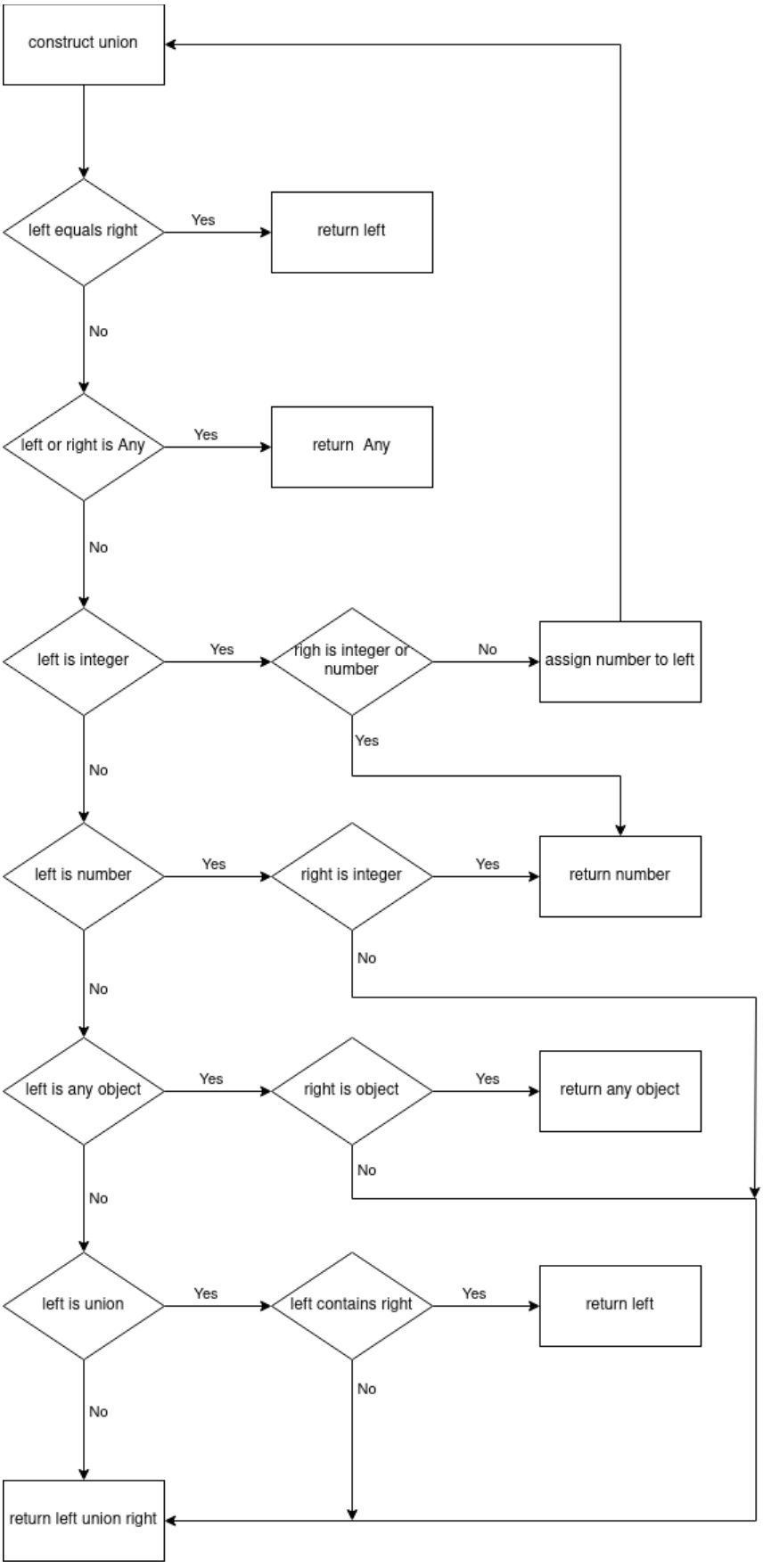
Interfaces is a virtual type that may actually be any type that fulfils its requirements. The optional field unlike a class, indicates whether a property is required.

It is implemented as follow:

```
/// descriptor of an interface property
#[derive(Debug, Clone)]
pub struct InterfacePropertyDesc {
	/// type
	pub ty: Type,
	/// is read only
	pub readonly: bool,
	/// is property optional
	pub optional: bool,
}

/// descriptor of an interface method
#[derive(Debug, Clone)]
pub struct InterfaceMethod {
	/// is method readonly
	pub readonly: bool,
	/// is method optional
	pub optional: bool,
	/// params of method
	pub params: Vec<Type>,
	/// return type of method
	pub return_ty: Type,
}

/// an interface definition
#[derive(Debug, Default)]
pub struct InterfaceType {
	/// name of the interface, only for debugging purpose
	pub name: String,

	/// classes that extend interface
	pub extends: Vec<ClassId>,
	/// interfaces implemented by interface
	pub implements: Vec<InterfaceId>,
	/// properies of this interface
	pub properties: HashMap<PropName, InterfacePropertyDesc>,
	/// methods of this interface
	pub methods: HashMap<PropName, InterfaceMethod>,
}
```

3.2.3 constructing union types

Union types are types that is possibly one of the type in the list. It is essentially an interface but with additional guarantees against its possibility of types.



```rust
pub fn union(self, other: Type) -> Type {
	if self == other {
		return self;
	}
	if self == Type::Any || other == Type::Any {
		return Type::Any;
	}
	if (self == Type::Number || other == Type::Number)
		&& (self == Type::Int || other == Type::Int){
		return Type::Number;
	}
	if other == Type::Int {
		return self.union(Type::Number);
	}
	if self == Type::Int {
		return Type::Number.union(other);
	}
	match &self {
		Type::Any => return self,
		Type::AnyObject => if other.is_object(){
			return Type::AnyObject
		} else{
			return Type::Union(Box::new([self, other]))
		},
		Type::Union(u) => {
			if u.contains(&other) {
				return self;
			}
			let mut v = Vec::with_capacity(u.len() + 1);
			for ty in u.iter() {
				v.push(ty.clone())
			}
			if let Type::Union(u) = other{
				for ty in u.iter() {
					v.push(ty.clone());
				}
			} else{
				v.push(other);
			}
			v.sort();
			return Type::Union(v.into_boxed_slice());
		}
		_ => Type::Union(Box::new([self, other])),
	}
```

3.2.4 Testing union constructions

Input1: left	Input2: right	output	pass
null	null	null	Yes
Any	undefined	Any	Yes
integer	number	number	Yes
Number	integer	number	Yes
integer	object	Number | object	Yes
Any Object	object	Any Object	Yes
Any Object	interface	Any Object | interface	Yes
Any Object	Any	Any	Yes
Any Object	symbol	Any Object | symbol	Yes
Null | interface	null	Null | interface	Yes
Object | interface	object	Object | interface	Yes
Null | symbol | number	Null | Number	Null | symbol | Number	Yes
number	BigInt	Number | BigInt	Yes
interface	boolean	Interface | boolean	Yes
Literal Boolean	boolean	Literal boolean | boolean	No
Literal integer	number	Literal integer | number	No
Literal number	number	Literal number | number	No
Literal string	string	Literal string | string	No

During testing, I have found out that literal types are not treated as its base type and the function returns a union. I have solved the problem by adding codes that converts literal types into their base types.

After implementing the above solution, I have run the tests again:

input	input	output	Pass
Literal Boolean	boolean	boolean	Yes
Literal integer	number	number	Yes
Literal number	number	number	Yes
Literal string	string	string	Yes

And now they are running correctly as expected.

3.3 HIR translating types

3.3.1 translate Typescript type annotation

The source code can be referenced at native-ts-hir/transform/types.rs

This function takes in the Transformer and the AST node as argument. Its body is implemented as a single switch that branches to other translation functions.

```
pub fn translate_type(&mut self, ty: &swc::TsType) -> Result<Type>
```

branches:

condition	branch to	result
Array type	translate_type	Array type of Type
Conditional type	translate_conditional_type	Type
Function type	translate_func_type	Function type
Constructor type	/	Error
Import type	/	Error
Index Access type	translate_index_access_ty	Map type
Infer type	/	Error
Keyword type	translate_keyword_type	Type
Literal type	/	Error
Mapped type	/	Error
Optional type	translate_type	Type union undefined
Parenthesized type	translate_type	Type
Rest type	/	Error
This type	/	Current context ‘this’ type
Tuple type	translate_tuple_type	Tuple type
Type literal	/	Error
Type operator	translate_type_operator	Type
Type Predicate	translate_type_predicate	Type
Type Query	translate_type_query	Type
Type reference	translate_type_ref	Type

3.3.2 translating conditional type

Translate type is called to translate the element type.

```
/// the type to be tested
let check_ty = self.translate_type(&c.check_type)?;
/// the constrain to be checked against
let extends_ty = self.translate_type(&c.extends_type)?;
/// the type to return when condition is false
let false_ty = self.translate_type(&c.false_type)?;
/// the type to return when condition is true
let true_ty = self.translate_type(&c.true_type)?;

/// check that test type matches constrain
let condition = self.type_check(c.span, &check_ty, &extends_ty).is_ok();

if condition{
        return Ok(true_ty)
} else{
        return Ok(false_ty)
}
```

Testing translation of conditional types

input	output	justification	pass
number extends number? number: string	number	Condition have the same type therefore should always be true	Yes
number extends string? number: string	string	Number is not the same as string and therefore condition is false	Yes
number extends any? number : string	number	Number is compatible with any and therefore condition is true	Yes
{} extends Object? any: null	any	All object type inherits Object therefore condition is true	Yes
null extends Object? any: null	null	Null is not an Object therefore condition is false	Yes
number extends object | number ? number: null	number	The union contains number therefore condition is true	Yes

As shown above, all the tests have passed, we can therefore conclude that there is no error in translating conditional types.

3.3.3 translate index access type

This function returns a Map type that uses an index type and a object type. The read-only property of the index access type can be specified by the user. However, we will not be supporting it as it simply prevents user from mutating the map. It does not affect accuracy.

```
/// a read only map cannot be modified
/// we do not support this right now but it does not affect accuracy
if map.readonly {
        // todo!()
}

/// the type use as index to access elements
let index_ty = self.translate_type(&map.index_type)?;
/// the type stored in the map as values
let value_ty = self.translate_type(&map.obj_type)?;

/// return the map type
return Ok(Type::Map(Box::new(index_ty), Box::new(value_ty)));
```

Testing translation of index access type

input	output	Justification	pass
string[number]	Map<number, string>	random	Yes
null[any]	Map<any, null>	random	Yes
object[string]	Map<string, object>	random	Yes

As shown above, all the tests have passed, I can therefore conclude that there is no error in translating index access types.

However, I have later found out that I have misunderstand the meaning of an index access type. I thought that an index access type is a map type. However it has a completely different meaning.

Index access type is a type that looks up a specific property on another type. It references the type of that property and takes it as its own.

Therefore I reimplemented the function:

```
// the index access type
let index_ty = self.translate_type(&i.index_type)?;
// the object type being accessed
let value_ty = self.translate_type(&i.obj_type)?;

match &value_ty {
    // array types can use number type as index access type
    Type::Array(elem) => match &index_ty {
        // number types
        Type::Number | Type::LiteralNumber(_) | Type::Int | Type::LiteralInt(_) => {
            // return a clone of element type
            return Ok(elem.as_ref().clone());
        }
        // otherwise fall through
        _ => {}
    },
    // tuple types can use number type as index access type
    Type::Tuple(elem) => match &index_ty {
        // number type with unknown index
        Type::Number | Type::Int => {
            // return a union with possible element types
            return Ok(Type::Union(elem.clone()));
        }
        // otherwise fall through
        _ => {}
    },
    // fall through
    _ => {}
}

// create a property name out of literal types
let key = match index_ty {
    // integer
    Type::LiteralInt(i) => PropName::Int(i),
    // number
    Type::LiteralNumber(i) => PropName::Int(i as i32),
    // string
    Type::LiteralString(s) => PropName::String(s.to_string()),
    // not a supported index access type
    _ => {
        return Err(Error::syntax_error(
            i.index_type.span(),
            format!("type '' cannot be used as index access type"),
        ))
    }
```

```
};

// find the property type
if let Some(ty) = self.type_has_property(&value_ty, &key, false) {
    // return property type
    return Ok(ty);
} else {
    // object type does not have the property
    return Err(Error::syntax_error(
        i.index_type.span(),
        format!("type '' has no property '{}'", key),
    ));
}
```

Testing index access type

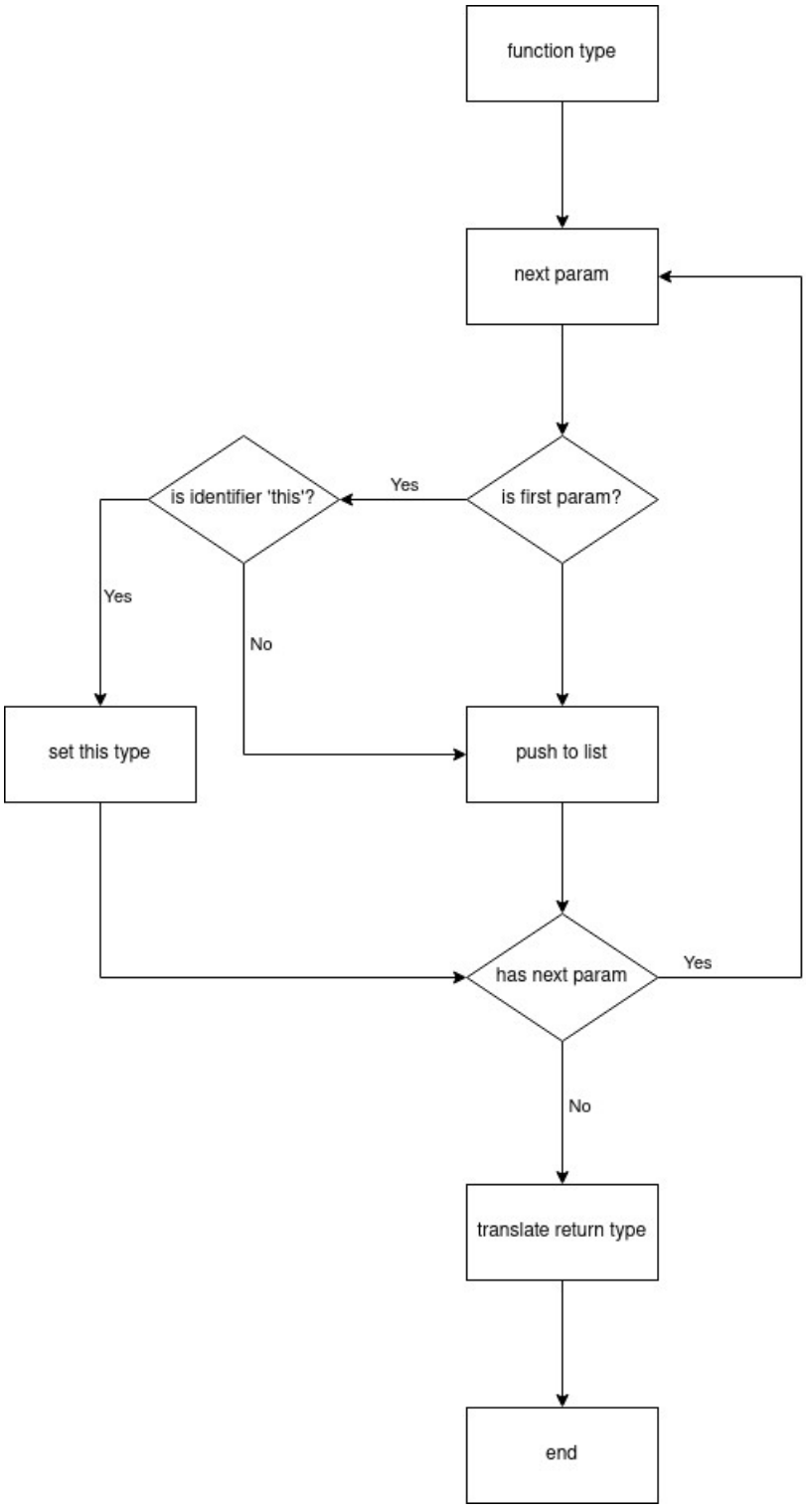
Object type	Index type	output	Justification	pass
string[]	number	string	Number index type on an array should return element type	Yes
{a:number}	“a”	number	String literal index should return the property type	Yes
[null, string]	1	string	Number literal index should return the element type at index	yes
any	number	error	Any type have no properties and should return an error	Yes
{}	99n	error	Big integer cannot be used as index	Yes
[]	{o:3}	error	Object type cannot be used as index	Yes

As shown above, all the test have passed. Therefore I have conclude that the translation of index access types is correct.

3.3.4 translate function type

A function type annotation represents a function. The argument types and return type must be translated using translate_type.

The control flow is as follow:



```
// ‘this’ type default to any
let mut this_ty = Type::Any;
// vector to store parameter types
let mut params = Vec::new();

// translate parameters
for (i, p) in func.params.iter().enumerate() {
    if let swc::TsFnParam::Ident(id) = p {
        // parameter must have type annotation
        if id.type_ann.is_none() {
            // return an error
            return Err(Error::syntax_error(
                id.span,
                "missing type annotation"
            ));
        }
        // translate type annotation of parameter
        let mut ty = self.translate_type(
            &id.type_ann.as_ref().unwrap().type_ann
        )?;
        // optional type
        if id.optional {
                ty = ty.union(Type::Undefined);
        }
        // explicit ‘this’ type
        if id.sym.as_ref() == "this" && i == 0 {
                this_ty = ty;
                continue;
        }
        // push parameter to vector
        params.push(ty);
    } else {
        // function type should not be destructive
        return Err(Error::syntax_error(
            p.span(), "destructive params not allowed",
        ));
    }
}

// translate return type
let return_ty = self.translate_type(&func.type_ann.type_ann)?;
```

Testing translation of function type

A test have been conducted to test the translation of function type.

input	This type	parameters	Return type	Justification	pass
`(number) => number`	any	number	number	Simple types	Yes
`(this: object, string) => undefined`	object	string	undefined	The ‘this’ type is set.	Yes
`(string, any) => any`	any	String, any	any	Multiple parameter	Yes
`(string, {a}: {a:number}, ) => any`	/	/	/	Destructive parameter not supported	No
`(a?:number) => number`	any	Number | undefined	number	Optional parameter converts to union	Yes
`() => object`	any	-	object	No parameter	Yes
`([a, b]:[null, number]) => string`	/	/	/	Destructive parameter not supported	No

As shown above the test have all passed, except the two tests that have included unsupported features.

In the future, I would like to support destructive patterns for function parameters.

What happened	Action taken	Justification
Destructive parameter not supported	None	Destructive parameter are not supported for now. It is not an essential feature and has no emergency to implement it. It may cause confusion during development and therefore not implemented.

3.3.6 translate optional type

Optional types are alias for union between a type and undefined type. We simply translate the type and return a union.

```
// translate the target type
let ty = self.translate_type(&t.type_ann)?;

// construct and return the union
return Ok(ty.union(Type::Undefined));
```

3.3.7 translate parenthesized type

A parenthesized type is simply a syntax sugar for a normal type annotation. We simply translate the inner type annotation by calling the function translate_type.

The parenthesized type simply exist the differentiate the priority of type operations, it has no real meaning in any context.

3.3.8 translate keyword type

A keyword type is a Typescript syntax keyword that annotates a built-in type. Each keyword is mapped to a fixed concrete type illustrated as follow:

keyword	Result type
any	Any
bigint	BigInt
boolean	Boolean
intrinsics	*Error
never	Undefined
null	Null
number	Number
object	Any Object
string	String
symbol	Symbol
undefined	Undefined
unknown	Any
void	Undefined | null

3.3.9 translate tuple type

A tuple type has a fixed length. Unlike an array type, each element in a tuple can have a different type.

No testing is required for this function as it is straight forward and simple.

```
// a vector to store all element types
let mut tys = Vec::new();
// loop through each element
for i in &t.elem_types {
    // translate element type
    let ty = self.translate_type(&i.ty)?;
    // push element type to vector
    tys.push(ty);
}

// return tuple type
return Ok(Type::Tuple(tys.into_boxed_slice()));
```

Testing translate tuple type

input	output	Justification	Pass
[]	[]	Tuple without elements	Yes
[number]	[number]	Tuple with one element	Yes
[number, string]	[number, string]	Tuple with two element	Yes
[number, string, undefined]	[number, string, undefined]	Tuple with three elements	Yes
[number, string, undefined, object]	[number, string, undefined, object]	Tuple with four elements	Yes
[number, string, undefined, object, bigint[]]	[number, string, undefined, object, bigint[]]	Tuple with five elements	Yes
[number, string, undefined, object, bigint[], {}]	[number, string, undefined, object, bigint[], {}]	Tuple with six elements	Yes

3.3.10 translate type operator

A type operator extracts or inject properties to a type. There is currently three variants of operator in the Typescript specification.

operator	supported	description
Key of	Yes	The `key of` operator takes an object type and produces a string or numeric literal union of its keys.
Read only	*No	The read only operator clarifies that all the properties in a field cannot be changed.
Unique	*No	The unique operator

Since read only and unique operator is not supported, we will only implement the key of operator. When we encounter a read only of unique operator, we simply return the inner type.

```
// translate the type
let ty = self.translate_type(&o.type_ann)?;
// vector for union type
let mut elem = Vec::new();

// loop through properties
for p in self.get_properties(&ty){
   match p{
      // a literal string
      PropName::Ident(id) |
      PropName::String(id) => elem.push(Type::LiteralString(id.into())),
      // a literal integer
      PropName::Int(i) => elem.push(Type::LiteralInt(i)),
      // private properties are not visible
      PropName::Private(s) => {},
      // a symbol
      PropName::Symbol(s) => elem.push(Type::Symbol),
   }
};

// return union
return Ok(Type::Union(elem.into()))
```

Testing type operator

A test have been conducted to test the translation of type operators.

input	expected	output	justification	pass
{a:any}	"a"	"a"	Object has a single key	Yes
{}	undefined	error	Object has no keys therefore undefined	No

During the test, we have found that we have not check the number of elements in a union. An empty union is not a valid type an will cause an error. We added code to check for that before returning the union.

```
// no properties
if elem.is_empty(){
    return Ok(Type::Undefined)
}
```

number[]	number	...	Key of an array includes the built in properties and number	No

When executing the next test, we have found that the union does not contain number type. This is because the method get_property does not return dynamic properties. To solve this I have added this code before the loop:

```
// add number to union if array
if let Type::Array(_) = &ty{
     elem.push(Type::Number);
}
```

any	string | number | symbol	undefined	Any can be dynamically accessed	No

The type any should be able to be dynamically accessed therefore the keys should be string | number | symbol. To solve this, I have added the following code:

```
// any can be dynamically accessed
if let Type::Any = &ty{
    elem.push(Type::String);
    elem.push(Type::Number);
    elem.push(Type::Symbol);
}
```

3.3.11 translate type predicate

A type predicate indicates the compiler that a function is a type guard. A type guard is some expression that performs a runtime check that guarantees the type in some scope [4]. Currently type guards are not supported.

Therefore the type predicate will result in a boolean type without further evaluation for now.

```
// any can be dynamically accessed
if let Some(ty) = &p.type_ann {
    // translate type
    let _ty = self.translate_type(&ty.type_ann)?;
    return Ok(Type::Bool);
} else {
    // type predicate must have type annotation
    return Err(Error::syntax_error(p.span, "missing type annotation"));
}
```

3.3.12 translating type query

A type query references a type of a variable. It looks up a binding using entity name. It then reference its type and return.

If the query target is not a function or variable, an error would be returned

Binding type	result
Generic Function	Error
Function	Function type
Variable	Type of variable
Using variable	Type
Class	Error
Generic class	Error
interface	Error
Generic interface	Error
Type alias	Error
Generic type alias	Error
Enum	Error
Namespace	Error

3.3.13 translate type reference

A type reference returns the type where the identifier is bind to.

When the binding of identifier is not a type, an error would be returned.

Binding	result
Generic Function	Error
Function	Error
Variable	Error
Using variable	Error
Class	Class type
Generic class	Class type
interface	Interface type
Generic interface	Interface type
Type alias	Type
Generic type alias	Type
Enum	Enum type
Namespace	Error

3.3.14 look up binding by entity name

Typescript type entity names may contain segments. In such case, further lookup must be performed on a module scale.

```
// lookup binding by entity name
fn find_binding(&mut self, entity_name: &swc::TsEntityName) ->
Option<Binding> {
    // match variant of entity name
    match entity_name {
        // an ident entity name
        swc::TsEntityName::Ident(id) => {
            // find binding from the current context
            return self.context.find(&id.sym).map(|b| b.clone())
        },
        // a chained entity name
        swc::TsEntityName::TsQualifiedName(q) => {
            // find the binding of module
            let left = self.find_binding(&q.left)?;

            match left {
                // only a module binding is allowed
                Binding::NameSpace(namespace) => {
                    // find the binding from module exports
                    return self.find_binding_from_module(
                        namespace,
                        &PropName::Ident(q.right.sym.to_string()),
                    );
                }
                // parent binding is not a module, no binding is found
                _ => return None,
            }
        }
    }
}
```

No further testing is required for this function as it is very straight forward.

3.3.15 attempts made to support generic types

Generic types are types that have no solid definition during translation. They represent a type unknown but constrained to an interface. During the process of **iterative development** of project, several attempts has been made to implement support for generic types.

The **first attempt** was to treat every generic type as an interface. This attempt has been a failure due to the behaviour of class not compatible with standard Typescript. Several Type operations and variable operations cannot be done in normal manner. Therefore it is removed from the project.

The **second attempt** was to separate generic type as a stand alone type and create copies of generic functions. Every time a generic type is declared, it’s constrains are registered into the current context. Type checks are not performed during the stage of translation but performed after HIR is fully constructed. A generic resolving pass is introduced to solve generics into concrete types. During the generic resolving pass, generic classes, methods and functions are copied. Generic types are replaced by a concrete type. A further type normalisation pass is used to simplify composite types. Type checks are then explicitly done in a separated pass. The solution however means that functions must be cloned the number of times it is called. Closures within functions must also be cloned every time its parent function is clone. This means that when a closure is declared in a function, the number of times its content being cloned is exponential. By implementing this solution, all type checks currently implemented must be stripped off. An early development has been made to implement this solution but it is later scrapped as it is too complicated.

The **third attempt** is to perform generic normalisation without translating types. This is done only with name bindings without any knowledge what the binding means. A copy of each variant is placed into the source code. However this method leads to confusion during translation and cannot accurately represent the original code. Therefore this attempt have been unsuccessful.

3.3.19 Adding support for literal types

Added: 19/1/2024

Literal types were not implemented when the project is in the first development cycles. However, it has been added later to better support a full set of typescript specification. Changes have been made to the original source code related to type checking. For instance, the type checking function has been modified so that literal types will be treated as their programme type. Several functions regarding to the translation of expressions such as binary operations, unary operations, update statements and literal expressions have been updated to supported literal types.

Five variants of literal types are implemented:

type	description
Literal Object	A literal object is an anonymous data structure that holds values in fields.
Literal number	A literal number is a type that is only compatible to a fixed floating point value.
Literal integer	A literal integer is a type that can only hold a fixed integer value.
Literal Big integer	A literal big integer is a type that can only hold a fixed big integer value.
Literal String	A literal string is a type that can only hold a fixed value of string.

3.4 HIR hoisting

According to MDN documents, “**Hoisting** refers to the process whereby the interpreter appears to move the *declaration* of functions, variables, classes, or imports to the top of their scope, prior to execution of the code.”[5]

Hoisting is not considered a part of the ECMA specification. However, it is considered a valid implementation of the language syntax. According to MDN, the following four behaviours are regarded as hoisting:

1. Being able to use a variable's value in its scope before the line it is declared. ("Value hoisting")
2. Being able to reference a variable in its scope before the line it is declared, without throwing a `ReferenceError`, but the value is always `undefined`. ("Declaration hoisting")
3. The declaration of the variable causes behaviour changes in its scope before the line in which it is declared.
4. The side effects of a declaration are produced before evaluating the rest of the code that contains it.

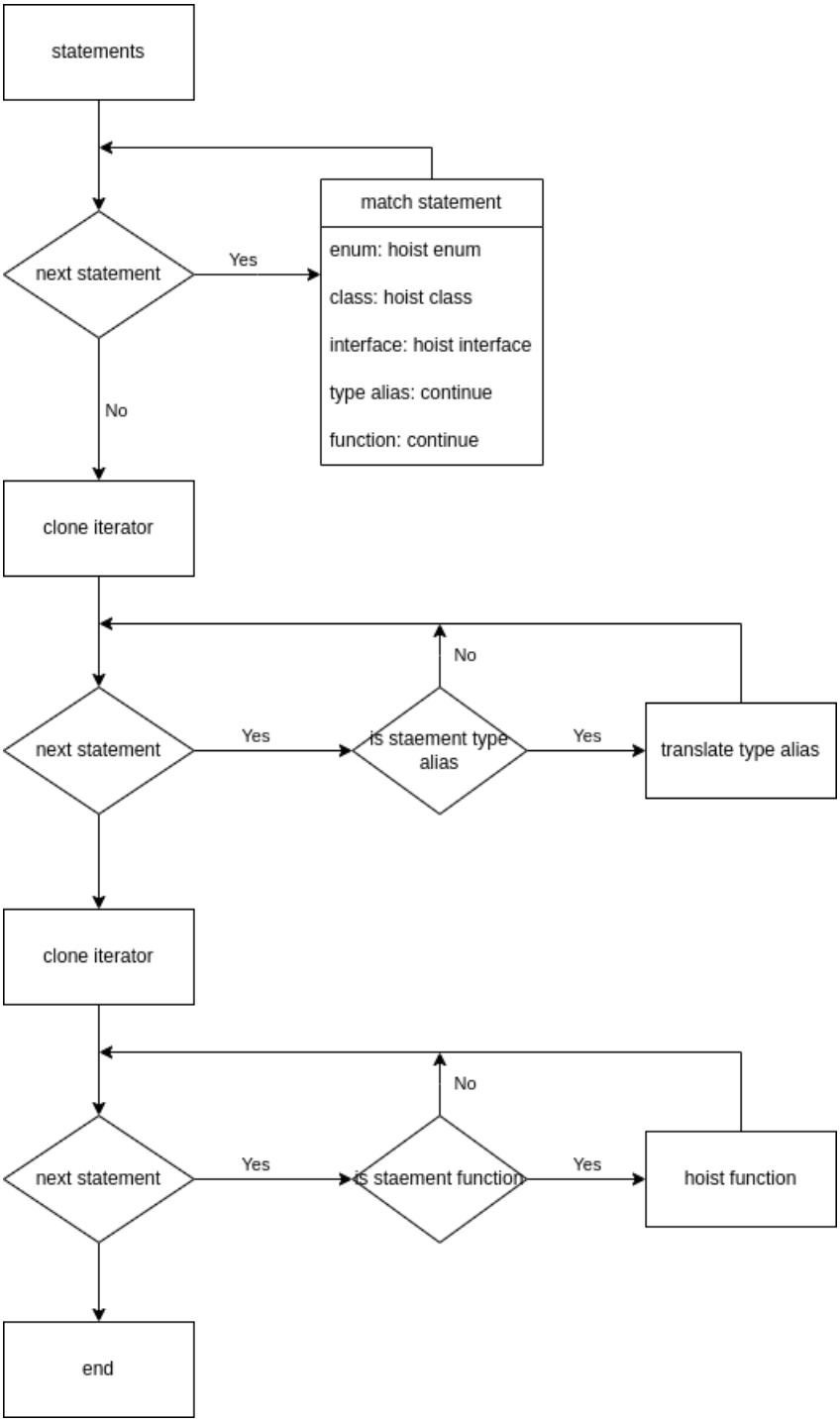
In my opinion, variable hoisting would usually cause confusion upon development as variables may be used before declaration. This type of behaviour usually causes logical errors that may not be notice during development time. Therefore to improve the validity of expressions of language, **Variable hoisting (value hoisting and reference hoisting) will not be implemented**. Variables must be declared and initialised before use just as how Rust behaves.

On the other hand, function, class, interface, type alias and enum hoisting are performed before translation of a scope.

3.4.1 Overall hoisting process

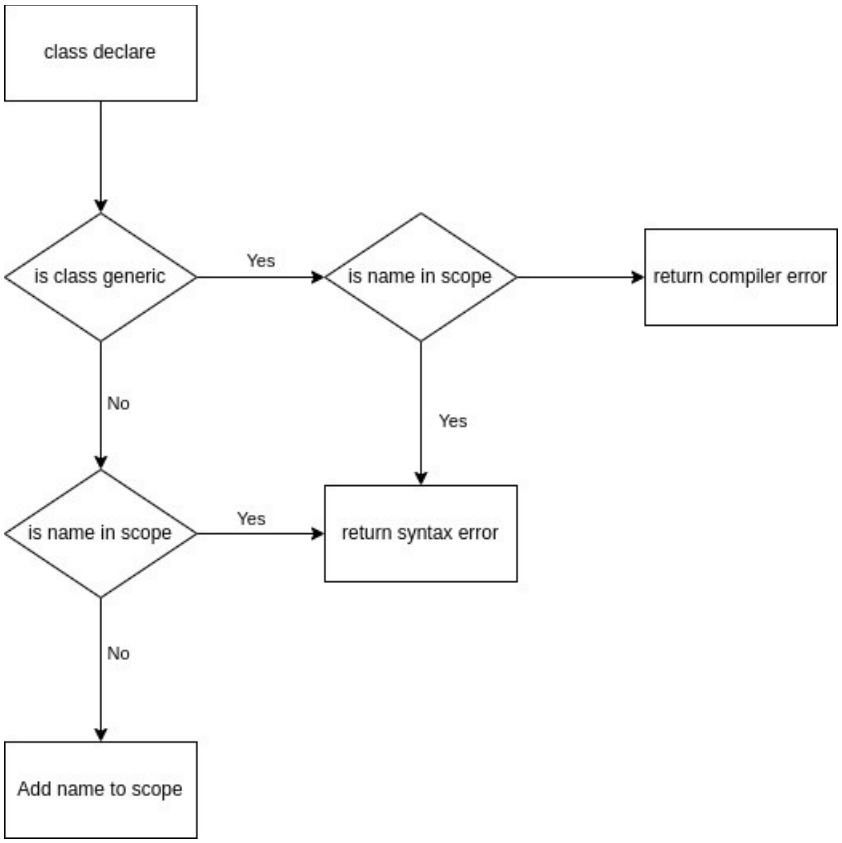
The hoisting process follows priority of declarations. Enums have the highest priority as they do not depend on any other user defined types. Class, interface and type alias may depend on each other and therefore are hoisted at the same time. Functions are hoisted last as it references on type definitions.

The hoist function is generic towards a value bounded by the Iterator trait with item as the reference to declaration statement. It is implemented as is because different part of the translation process may call on this function. This includes the global scope, function scope, block scope and any statements that creates a new scope.



3.4.2 hoisting classes

Class hoisting is performed during the hoisting process. The hoisting process of a Class involves adding the class name to symbol table and translating generics of a class. At the time this is written, Generics are not currently supported therefore will result to an error. The only concern would be to add the class to symbol table and register to the current scope.



classes would be further evaluated after hoisting of interfaces, enums and type aliases. The type of constructor, attributes and methods are translated. This allow users to construct objects, access attributes and call upon methods.

```
pub fn hoist_class(
    &mut self,
    name: Option<&str>,
    class: &swc::Class
) -> Result<()> {
    // create a new class id
    let id = ClassId::new();
    // check for generic paramaters
    if class.type_params.is_some() {
        // declare generic class
        if !self
            .context
            .declare(name.unwrap_or(""), Binding::GenericClass(id))
        {
            // identifier is already used
            return Err(Error::syntax_error(class.span, "duplicated identifier"));
        }
    } else {
        // declare the class
        if !self.context.declare(name.unwrap_or(""), Binding::Class(id)) {
            // identifier is already used
            return Err(Error::syntax_error(class.span, "duplicated identifier"));
        }
    }
    return Ok(());
}
```

Testing hoist class

Input	output	Justification	Pass
class MyClass{ }	/	Simple class declaration	Yes

class Car extends Vehicle{
 	
}
 	
class Vehicle{
}
	/	Class Vehicle referenced before declare	Yes

class Car{} class Car{}	Syntax Error: duplicated identifier	Duplicated declaration	Yes

3.4.3 hoisting interfaces

Interface hoisting is performed during the hoisting process. The hoisting process of an interface is basically the same as a class. The name of the interface is checked against the scope and any generics will be translated and checked against. Interfaces may also extend a class or to implement another interface.

Interfaces are further translated after hoisting of classes, enums and type aliases. The types of attributes and methods are translated. This allows type checks to be perform when translating statement.

```
pub fn hoist_interface(
    &mut self,
    name: Option<&str>,
    iface: &swc::TsInterfaceDecl,
) -> Result<()> {
    // create a new id for interface
    let id = InterfaceId::new();
    // a generic interface
    if iface.type_params.is_some() {
        // declare generic interface
        if !self
            .context
            .declare(name.unwrap_or(""), Binding::GenericInterface(id))
        {
            // identifier already used
            return Err(Error::syntax_error(iface.span, "duplicated identifier"));
        }
    } else {
        // declare interface
        if !self
            .context
            .declare(name.unwrap_or(""), Binding::Interface(id))
        {
            // identifier already used
            return Err(Error::syntax_error(iface.span, "duplicated identifier"));
        }
    }
    return Ok(());
}
```

Test hoisting of interfaces

Test input	output	Justification	Pass
`interface A{}`	/	Normal interface declaration	Yes
`interface A{}` `interface A{}`	error	Duplicated interface declaration	No

What happened	reason
Interface should be allowed to declare multiple times	Declaration merging should be performed by the compiler

Action	Justification
Added code to retrieve the current binding of the identifier if any. It the current binding is an interface, do not return error.	This is to allow declaration merging of interface in later stages of translation.

```
// allow declaration merging
if let Some(Binding::Interface(id)) = self.context.find(name){
    return Ok(())
}
```

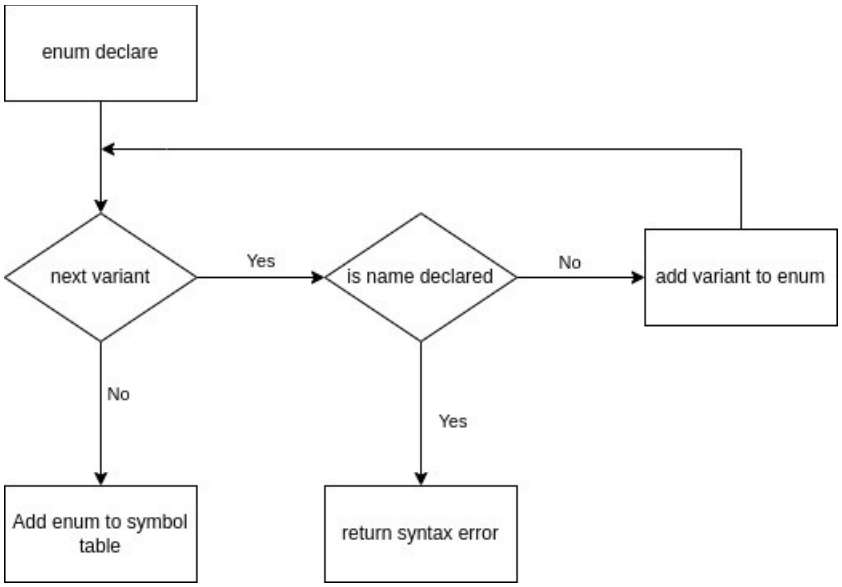
I rerun the test again

Test input	output	Justification	Pass
`interface A{}` `interface A{}`	/	Duplicated interface declaration	Yes

No error has been returned, the test has passed.

3.4.4 hoisting enums

Enum hoisting is performed during the hoisting process. The hoisting process of an enum translates the enum type definition. This allows users to construct variants of enum beforehand. The enum type allows user to declare literal, string or number type as its variant type. However, at this time when this is written, typed variants are not supported in order to reduce complexity. This will be implemented in the future.



3.4.5 hoisting type alias

A type alias is a name binding of a certain type defined by the user. Type aliases are alias of actual types. All class, interface and enum hoisting must be done before type alias hoisting. The type of the alias is translated. A unique type alias id is generated and registered to the symbol table.

When the hoisting process is finished, the symbol table is looked at to search for aliases. The aliases will then be replaced by its concrete type. This process is known as normalisation where alias types will not be present in the following translation process.

3.4.6 hoisting functions

Function hoisting is performed during the hoisting process. Function hoisting is performed last after all type hoisting are done. This is because function hoisting translate the types of parameters and return type of a function and return a function type. Function type depends on other type while other types does not depend on function types.

3.4.7 hoisting variables

Variable hoisting is currently not performed. User should declare variables before using it.

3.5 HIR translating statements

Statements are translated from AST to HIR. Some statements are simplified into multiple statements resulting in an overall decrease of number of variants of statements.

The entry point for translating statements is a function that uses a switch to branch to the corresponding statement translation function.

Below is a list of translation of Typescript statements.

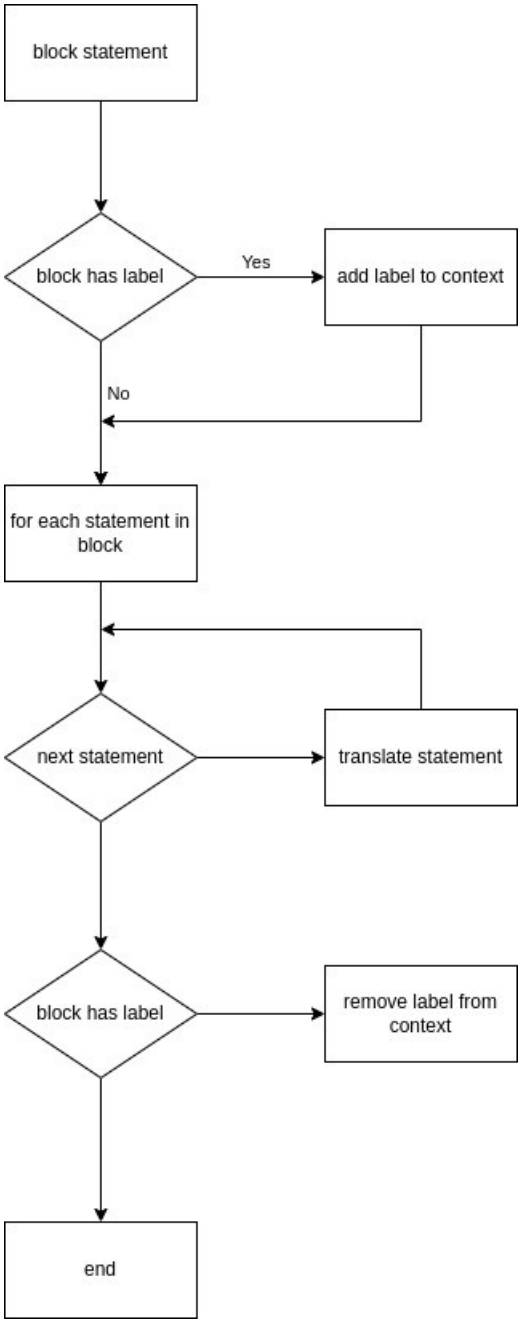
name	description
Block statement	A block statement creates a new scope and allows break statement to jump outside the scope from within.
Break statement	A break statement allows the programme to jump outside a scope. This includes loops, blocks and switches.
Continue statement	A continue statement allows the programme to skip the current iteration of loop and jump to the beginning of next iteration.
Debugger statement	Debugger statement allows users to call out the debugging panel. This is not supported and will result in an error
Empty statement	An empty statement has no content, no translation is done for this statement.
Declare statement	A declare statement defines symbols to which users may reference. This includes variables, classes and functions. The declare statement is hoisted beforehand.
Do...while statement	A do...while statement creates a loop. The content of the loop is executed first and the test value is checked against at last.
For statement	A for statement creates a loop. The condition is checked on each iteration. The content of loop is then executed.
For...in statement	A for...in statement creates a loop. The number of iteration depends on the number of element in the provided iterator. A variable binding is assigned from the counter on each iteration.

For..of statement	A for..of statement creates a loop. The number of iterations depends on the number of element in the provided iterator. A variable binding is created and assign from the current element in the iterator.
While statement	While statement creates a loop the condition for execution is checked at the beginning of each iteration.
If statement	An if statement is a conditional statement. The content of the block is executed when a condition is met. An if statement also have an optional else clause which allows the programme to execute when the condition does not met.
Return statement	A return statement returns a value from a function. This is only valid in a function scope.
Throw statement	A throw statement throws out an error to the outer scope. If no error handler is present, the programme will exit with the error.
Try statement	A try statement creates a error handler. Any error thrown within its scope would be caught by the given handler. The catch clause is executed if an error is present. The try statement also allows an optional finalising block that would be executed no matter what happens.
Switch statement	Switch statement matches the target value. A switch contains test cases that compares the case to the target value. The switch case is executed if value matches. Fall through to next switch case happens when no break is given. A default case would be executed if no cases matches.
Expression statement	An expression statement executes an expression and discards its result.

3.5.1 translate block statement

Block statements are statements that opens up a new scope. A scope contains variable and other declarations that will be unavailable when out of scope. An additional feature of an ECMA block statement is the tag that allows break statements to refer to the target block. By adding a tag to a block statement, breaks are allowed to break away from a block and jump behind the block.

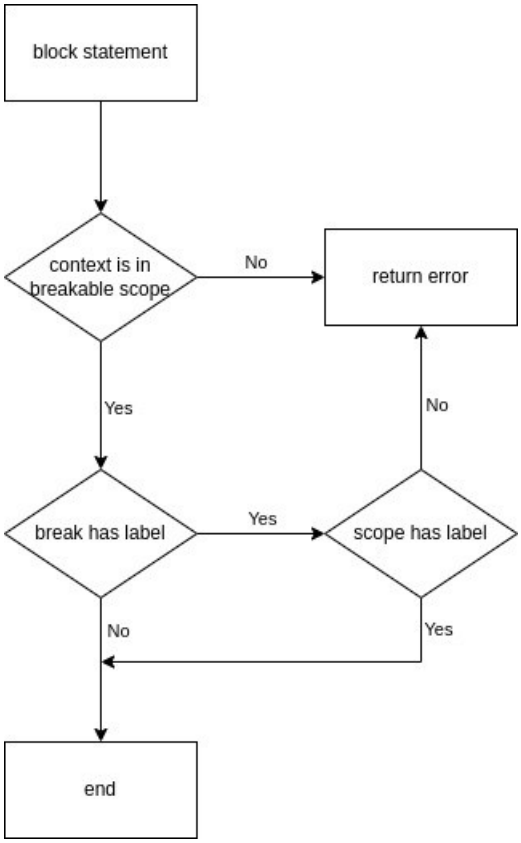
We first check if a label is presented. If a label is presented, it is added to the translator's label register so that break statements within the block can check against the label. A new scope is then added to the current context. All he statements are hoisted before translation. After translating every statement within the block, the scope is closed and the label is removed if any.



```rust
pub fn translate_block_stmt(
    &mut self,
    block: &swc::BlockStmt,
    label: Option<&str>,
) -> Result<()> {
    // true if the label is shared with parent scope
    let mut old_break = false;
    // insert the label
    if let Some(label) = label {
        // insert label while checking if label already exist
        old_break = !self.break_labels.insert(label.to_string());
        // start of block
        self.context.func().stmts.push(Stmt::Block {
            label: label.to_string(),
        });
    }
    // open a new scope
    self.context.new_scope();
    // hoist the statements
    self.hoist_stmts(block.stmts.iter())?;

    // translate all the statements
    for stmt in &block.stmts {
        self.translate_stmt(stmt, None)?;
    }

    // close the scope
    self.context.end_scope();

    // remove the label
    if let Some(label) = label {
        // only remove if label is not shared
        if !old_break {
            self.break_labels.remove(label);
        }
        // end of block
        self.context.func().stmts.push(Stmt::EndBlock);
    }

    return Ok(());
}
```

3.5.2 translate break statement

Break statements are only valid within a loop, a block and a switch. Checking for the scope in context is therefore required as it is not always valid. If a label if provided to the break, it is also checked against the context for validation of label.




```
// check for label
if let Some(label) = &b.label {
    if !self.break_labels.contains(label.sym.as_ref()) {
        return Err(Error::syntax_error(label.span, "undefined label"));
    }
}
// push break stmt
self.context
    .func()
    .stmts
    .push(Stmt::Break(label.map(|l| l.to_string())));
```

No testing is required for this function as it is straight forward.

3.5.3 translate continue statement

The `continue` statement terminates execution of the statements in the current iteration of the current or labelled loop, and continues execution of the loop with the next iteration.

In contrast to the `break` statement, `continue` does not terminate the execution of the loop entirely, but instead jump to the front of the loop.

The `continue` statement can include an optional label that allows the programme to jump to the next iteration of a labelled loop statement instead of the innermost loop. In this case, the `continue` statement needs to be nested within this labelled statement.

A `continue` statement, with or without a following label, cannot be used at the top level of a script, module, function's body, or static initialization block, even when the function or class is further contained within a loop.

The translation process is the same as the `break` statement.

```
// check for label
if let Some(label) = &c.label {
    // no label found
    if !self.continue_labels.contains(label.sym.as_ref()) {
        return Err(Error::syntax_error(label.span, "undefined label"));
    }
}
// push continue stmt
self.context
    .func()
    .stmts
    .push(Stmt::Continue(label.map(|l| l.to_string())))
```

No testing is required for this function as it is straight forward.

3.5.4 translate declare statement

A declaration statement declares a name binding to a specific type of entity. This includes variables, functions, classes, interfaces, enums and type aliases. The declaration statement must be hoisted before translation happens. The hoisting process of the statement would have generated a corresponding identifier on the symbol table that other statements can reference.

For interface, enum and type alias declaration, these types have already been translated due to the need of hoisting, further evaluation is not required and therefore return immediately.

For classes, the inner body of the class is not yet translated and will be translated by calling translate class. (See 3.3.16 translating classes) The id would first be retrieved using the binding identifier. And the class would be translated. A class declaration statement is then injected to the current context function.

For variables, see 3.5.5 translating variable declare.

For functions, the function body is translated, see 3.6.8 translating functions.

```rust
pub fn translate_decl(&mut self, decl: &swc::Decl) -> Result<()> {
    match decl {
        // class declare
        swc::Decl::Class(c) => {
            let id = self.context.get_class_id(&c.ident.sym);
            self.translate_class(id, c.ident.sym.to_string(), &c.class)?;
            self.context.func().stmts.push(Stmt::DeclareClass(id));
        }
        // function declare
        swc::Decl::Fn(f) => {
            let id = self.context.get_func_id(&f.ident.sym);
            self.translate_function(id, None, &f.function)?;
            self.context.func().stmts.push(Stmt::DeclareFunction(id));
        }
        swc::Decl::TsEnum(_) => {
            // do nothing
        }
        swc::Decl::TsInterface(_) => {
            // do nothing
        }
        swc::Decl::TsModule(_) => {
            // do nothing
        }
        swc::Decl::TsTypeAlias(_) => {
            // do noting
        }
        // using variable declare
        swc::Decl::Using(u) => {
            self.translate_using_decl(u)?;
        }
        // other variable declare
        swc::Decl::Var(decl) => {
            self.translate_var_decl(decl)?;
        }
    }
    return Ok(());
}
```

No testing is needed for this function, it is just a function that wraps against the switch.

3.5.5 translating variable declare

There are four variables types that can be declared, **`var`**, **`let`**, **`const`** and **`using`**. Each type of variable declaration behaves differently.

The `var` statement declares function-scoped or globally-scoped variables, optionally initializing each to a value. `var` declarations, wherever they occur in a script, are processed before any code within the script is executed. Declaring a variable anywhere in the code is equivalent to declaring it at the top. This also means that a variable can appear to be used before it's declared. This behaviour is called *hoisting*, as it appears that the variable declaration is moved to the top of the function, static initialization block, or script source in which it occurs.

The `let` declaration declares re-assignable, block-scoped local variables, optionally initializing each to a value. It is similar **`var`**, however when compared to **`var`** declaration, it has the following restriction:

- **`let`** declarations can only be accessed after the place of declaration is reached. It is not hoisted.
- **`let`** declarations do not create properties on `globalThis` when declared at the top level of a script.
- **`let`** declarations cannot be redeclared by any other declaration in the same scope.

The `const` declaration declares block-scoped local variables. The value of a constant can't be changed through reassignment using the assignment operator, but if a constant is an object, its properties can be added, updated, or removed.

The **`using`** declaration declares non-assignable, block-scoped variables. It must be initialised with a value and cannot be changed through reassignment. It cannot be captured by a closure and is only visible within its local scope. The dispose method is called once variable drops out of scope whether or not an error has been thrown.

A variable declaration statement main contain multiple declarators. This means that declaring multiple variables with different initialiser is allowed in a single statement. These declarators shares a single variable type either var, let, const or using. The declarators are allowed to have destructive patterns such as object pattern and array pattern. These patterns decomposes the initial value into multiple variable by accessing its fields. However this representation is complex and is not suitable for an IR code. Therefore we will have to decompose destructive patterns into simple assignment expressions. Destructive pattern is not allowed when in **`using`** declaration.

To simplify the process we first implement a function that loops through each declarator. It will translate the initialiser and call to translate the declaration pattern.

```
/// variable declaration
pub fn translate_var_decl(&mut self, decl: &swc::VarDecl) ->
Result<Vec<VariableId>> {
    // vec to store newly created ids
    let mut ids = Vec::new();

    // loop through each declorator
    for d in &decl.decls {
        // initialiser
        let init = if let Some(init) = &d.init {
            // translate expression
            Some(self.translate_expr(&init, None)?)
        } else {
            None
        };

        // translate the variable declare with pattern and initialiser
        ids.extend_from_slice(
            &self.translate_pat_var_decl(decl.kind, &d.name, init, None)?
        )
    }
    // return ids
    return Ok(ids);
}
```

To translate a declaration pattern, we implement this function.

```
fn translate_pat_var_decl(
    &mut self,
    kind: swc::VarDeclKind,
    pat: &swc::Pat,
    init: Option<(Expr, Type)>,
    parent_ann: Option<(Type, Span)>,
) -> Result<Vec<VariableId>> {
    match pat {
        // simple variable
        swc::Pat::Ident(id) => Ok(vec![
            self.translate_ident_var_dec(kind, id, init, parent_ann)?
        ]),
        // destructive array pattern
        swc::Pat::Array(a) => self.translate_array_pat_decl(kind, a, init,
parent_ann),
        // destructive object pattern
        swc::Pat::Object(obj) => self.tranlslate_object_pat_decl(kind, obj, init,
parent_ann),
        // assignment pattern, default initialiser
        swc::Pat::Assign(a) => {
            // not supported
            return Err(Error::syntax_error(
                a.span,
                "invalid left-hand side assignment",
            ))
        }
        // todo: rest assignment
        swc::Pat::Rest(r) => {
            return Err(Error::syntax_error(
                r.dot3_token,
                "rest assignment not supportd",
            ))
        }
        // only allowed in for-in loop, these are explicitly handled
        swc::Pat::Expr(_)|
        swc::Pat::Invalid(_) => {
            return Err(Error::syntax_error(
                pat.span(),
                "invalid left-hand side assignment",
            ))
        }
    }
}
```

Test translate variable declare

input	Output	Justification	Pass
let a = 0;	let var4:number var4=0 as number	Compiler should able to reference initialiser type if no type annotation.	Yes
let a: string = 0;	Syntax Error	Number type is not assignable to string type.	Yes
let a = 0, b: string = "0"	let var4:number var4=0 let var5:string var5="0"	Multiple declarator	Yes
var a = 0; var a = 9;	var var4:number var4=0 as number var var5:number var5=9 as number	Variable with var declaration can be redeclared	Yes
var a: number = 6; var a: string = "6";	var var4:number var4=6 var var5:string var5="6"	Redeclared variable must have the same type	No

What happened	Reason
Test failed when testing variable redeclaration	Compiler did not correctly identify the variable, it did not perform type check on the redeclaration

Action	Justification
I have reviewed the code that allows variables to be redeclared. I have added code to perform type check when a variable is redeclared.	The reason compiler did not behave correctly is because it does not perform type check. By adding type check, variable redeclared can only have the same type.

I have rerun the test again

input	output	pass
var a: number = 6; var a: string = "6";	Syntax Error: Subsequent variable declarations must have the same type.	Yes

input	output	Justification	Pass
`let a = 0;` `let a = 8;`	Syntax Error: duplicated identifier	‘let’ declared variables cannot be redeclared	Yes
`const a = 0;` `const a = 8;`	Syntax Error: duplicated identifier	‘const’ declared variables cannot be redeclared	Yes
`var a;`	Syntax Error: missing type annotation	Variable declaration should have type annotation or initialiser	Yes
`const a: number;`	`const var0:number`	Constant declaration must have initialiser	No

What happened	Reason
Test failed when testing constant variable declaration without initialiser	Compiler did not check for an initialiser for a constant declaration.

Action	Justification
I have reviewed the code that declare variables. I have added code to check for an initialiser when the variable declare kind is constant.	The reason the test has not pass is because the compiler did not check for an initialiser. By adding code to check for initialiser when declare kind is constant, an error should be returned.

I have added this code:

```
// 'const' declarations must be initialized.
if kind == swc::VarDeclKind::Const && init.is_none(){
    return Err(Error::syntax_error(
        ident.span,
        "'const' declarations must be initialized."
    ))
}
```

I have rerun the test:

input	output	Pass
`const a: number;`	Syntax Error: 'const' declarations must be initialized.	Yes

Now an error has been returned, the test has passed.

input	output	Justification	Pass
var [a, b] = [0, 9]; 	__pushstack__([0,9]) var var4:number var4=__readstack__()[0] var var5:number var5=__readstack__()[1] __popstack__()	Compiler should be able to reference element types in destructive assignment	Yes
var [a, b] = [0]; 	__pushstack__([0]) var var4:number var4=__readstack__()[0] var var5:number var5=__readstack__()[1] __popstack__()	Although initialiser only has one element, this is allowed because it is seen as an array instead of a tuple because there is no type annotation.	Yes
let [a, b]:[number, string] = [0, ""]; 	Syntax Error: type number | string is not assignable to type number	Destructive assignment with type annotation.	No

What happened	Reason
Test failed when testing array destructive assignment with type annotation	The compiler treats the initialiser as an array instead of a tuple

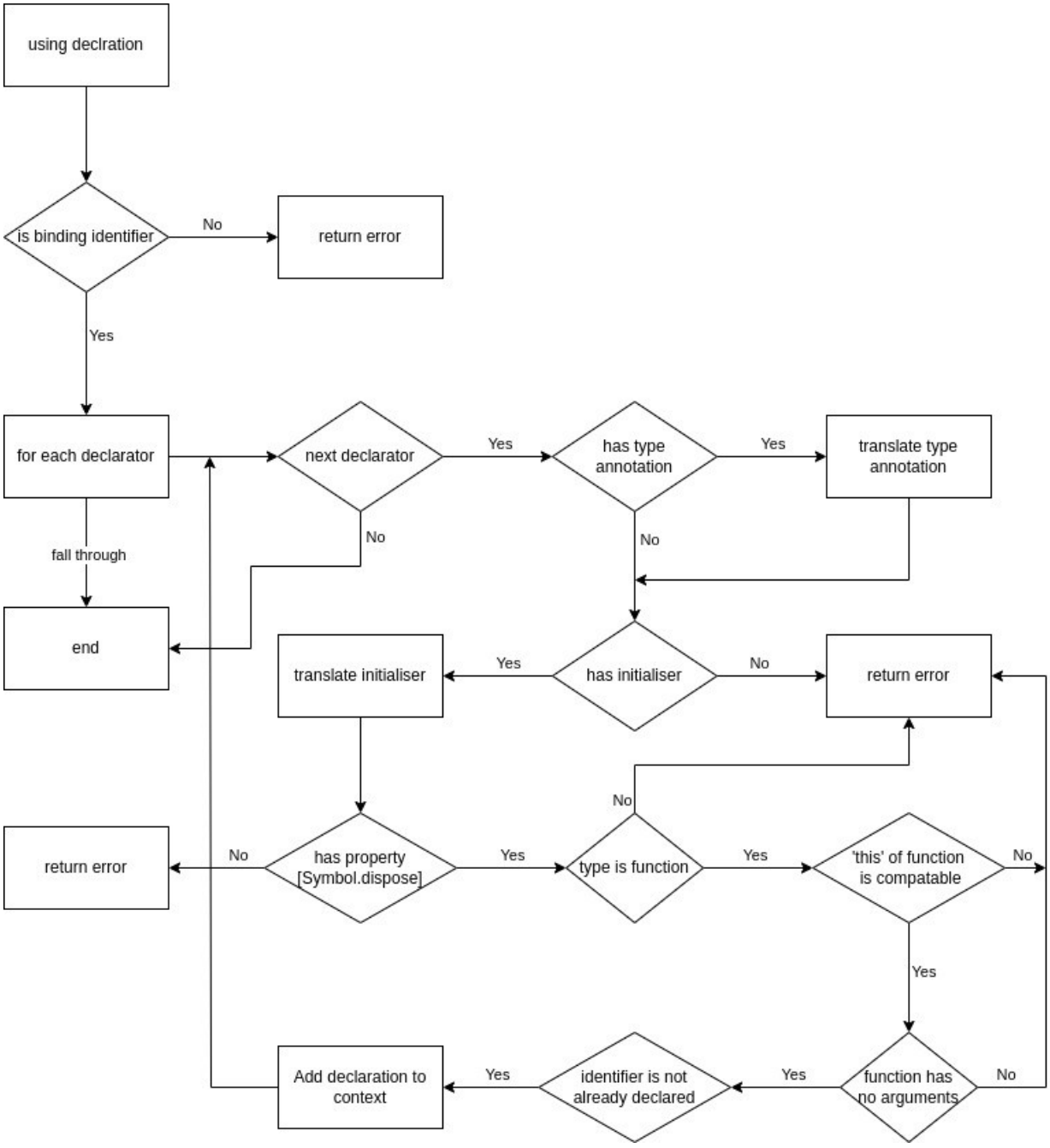
Action	Justification
I have reviewed the code for translating the initialiser. The initialiser is translated before the destructive pattern so no type hints were given to the expression. I have changed the sequence of translation by translating type annotation first and then translate initialiser with type hint.	By translating type annotation first and giving type hint to the expression, the array construction is treated as a tuple construction. The initialiser will return a tuple therefore fulfilling the type annotation's requirements.

I have rerun the test:

input	output	Pass
let [a, b]:[number, string] = [0, ""]; 	__pushstack__([0,""]) let var4:number var4=__readstack__()[0] let var5:string var5=__readstack__()[1] __popstack__()	Yes

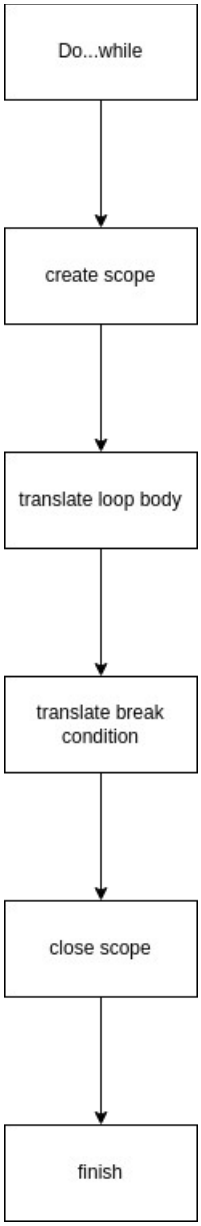
A using variable declaration must have an initialiser this is because using variables must be called upon the dispose method when it is out of scope. Using variables cannot be left uninitialised in any instance of the programme as there is a possibility that it may panic somewhere during the execution causing the value to be disposed.

To translate the using variable declaration, we loop through each declarator. We check that the binding pattern is an identifier, or else return an error. We then translate the type annotation if given. We translate the initial value and type check against its type making sure that it has the [Symbol.dispose] or [Symbol.asyncDispose] method.



3.5.7 translate do...while

Do...while loops are loops that executes until a condition is not met. The condition expression is evaluated and checked against after every iteration.



```rust
pub fn translate_do_while(&mut self, d: &swc::DoWhileStmt, label:
Option<&str>) -> Result<()> {

    // create new scope
    self.context.new_scope();
    // begin loop
    self.context.func().stmts.push(Stmt::Loop {
        label: label.map(|l| l.to_string()),
        update: None,
    });

    // translate blody
    self.translate_stmt(&d.body, None)?;
    // translate break condition
    let (test, _ty) = self.translate_expr(&d.test, Some(&Type::Bool))?;

   // insert condition branch to break if condition is false
    let func = self.context.func();
    // insert the statement
    func.stmts.push(Stmt::If {
       // insert condition
       test: Box::new(Expr::Unary {
          // break when false
          op: UnaryOp::LogicalNot,
          value: Box::new(test),
        }),
     });
    // insert break statement
    func.stmts.push(Stmt::Break(label.map(|l| l.to_string())));
    // end of conditional branching
    func.stmts.push(Stmt::EndIf);

    // end scope must go before end loop
    self.context.end_scope();
    // end loop
    self.context.func().stmts.push(Stmt::EndLoop);

    // return
    return Ok(());
}
```

Testing translate do...while statement

input	output	Justification	Pass

do{
 	
} while(true);
	for (;;){
	
	}
		Simple do..while statement	Yes

do { break; } while (false)	do { break; } while (false)	Break statement within do...while	Yes
do { continue; } while (false)	do { continue; } while (false)	Continue statement within do...while	Yes
label: do { break label; } while (false)	do { break; } while (false)	Break with label within do...while	No
label: do { continue label; } while (false)	do { continue; } while (false)	Continue statement with label within do...while	No

What happened	Reason
Test failed when labels are added to the statements	Labels are not properly translated in the function.

Action	Justification
I have reviewed the code that translates do...while statement. The code for handling labels are missing. I have therefore added code to the function to process labels.	Labels must be processed when translating loops to ensure accuracy of the program. By handling labels, break statements and continue statements in the loop can target specific context that the label is associated with.

After testing the translation function, I have found out that I forgot to implement translation for labels.

I have added these lines to the beginning of the function.

```
// true if the label for a breaking block already exist
let mut old_break = false;
// true if the label for a loop already exist
let mut old_continue = false;
if let Some(label) = label {
    // insert breakable label
    old_break = !self.break_labels.insert(label.to_string());
    // insert loop label
    old_continue = !self.continue_labels.insert(label.to_string());
}
```

I have added these lines to the end of the function:

```
// when label is present
if let Some(label) = label {
    // only remove label if it is not shared
    if !old_break {
        self.break_labels.remove(label);
    }
    // only remove label if it is not shared
    if !old_continue {
        self.continue_labels.remove(label);
    }
}
```

Now the translation function also handles label.

Input	Output	Justification	Pass
label: do { break label; } while (false)	Label: do { break label; } while (false)	Break with label within do...while	Yes
label: do { continue label; } while (false)	Label: do { continue label; } while (false)	Continue statement with label within do...while	Yes

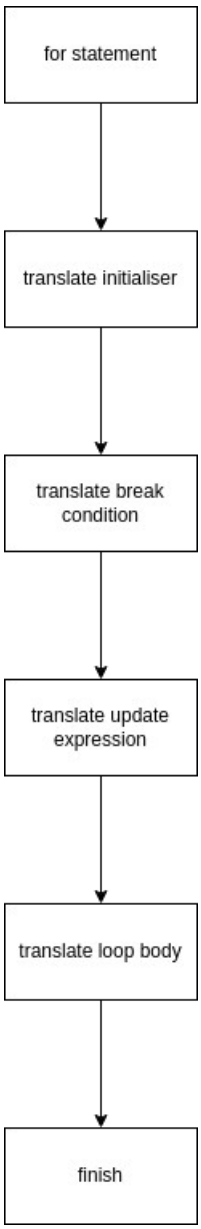
After running the tests on the interpreter, I have found out that the breaking conditions are not checked against if the user issues a continue statement in the loop. I have therefore rewrite the middle bit of the code so that the breaking condition is translated first. A field named end check is added to the loop statement.

So it now looks like this:

```
// translate break condition first
let (test, _ty) = self.translate_expr(&d.test, Some(&Type::Bool))?;

// create scope
self.context.new_scope();
// enter loop
self.context.func().stmts.push(Stmt::Loop {
    label: label.map(|l| l.to_string()),
    update: None,
    end_check: Some(Box::new(test))
});

// translate body
self.translate_stmt(&d.body, None)?;

// end scope must go before end loop
self.context.end_scope();
// end loop
self.context.func().stmts.push(Stmt::EndLoop);
```

3.5.7 translate for loop

The `for` statement creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement to be executed in the loop.



```rust
pub fn translate_for_stmt(&mut self, f: &swc::ForStmt, label: Option<&str>) ->
Result<()> {
    // is a label with same name exist
    let mut old_break = false;
    let mut old_continue = false;
    // register label
    if let Some(label) = label {
        old_break = !self.break_labels.insert(label.to_string());
        old_continue = !self.continue_labels.insert(label.to_string());
    }

    // open new context
    self.context.new_scope();
    // initialiser
    if let Some(init) = &f.init {
        match init {
            swc::VarDeclOrExpr::Expr(e) => {
                // translate expression
                let (e, _ty) = self.translate_expr(e, None)?;
                // push expression
                self.context.func().stmts.push(Stmt::Expr(Box::new(e)));
            }
            swc::VarDeclOrExpr::VarDecl(decl) => {
                // only hoist non var
                if decl.kind != swc::VarDeclKind::Var {
                    self.hoist_vardecl(decl)?;
                }
                // translate variable declare
                self.translate_var_decl(decl)?;
            }
        }
    }

    // translate update expression
    let update = if let Some(update) = &f.update {
        let (expr, _ty) = self.translate_expr(&update, None)?;
        Some(Box::new(expr))
    } else {
        None
    };

    // enter loop
    self.context.func().stmts.push(Stmt::Loop {
        label: label.map(|s| s.to_string()),
```

```
            update: update,
            end_check: None
        });

        // create new scope for loop body
        self.context.new_scope();

        // break if false
        if let Some(test) = &f.test {
            // translate break condition
            let (test, _ty) = self.translate_expr(&test, Some(&Type::Bool))?;
            let func = self.context.func();
            // break if not test
            func.stmts.push(Stmt::If {
                test: Box::new(Expr::Unary {
                    op: crate::ast::UnaryOp::LogicalNot,
                    value: Box::new(test),
                }),
            });
            func.stmts.push(Stmt::Break(None));
            func.stmts.push(Stmt::EndIf);
        }

        // translate body
        self.translate_stmt(&f.body, None)?;

        // close scope for loop body
        self.context.end_scope();
        // end loop
        self.context.func().stmts.push(Stmt::EndLoop);
        // close scope for for-head
        self.context.end_scope();

        // remove label
        if let Some(label) = label {
            if !old_break {
                self.break_labels.remove(label);
            }
            if !old_continue {
                self.continue_labels.remove(label);
            }
        }
        return Ok(());
    }
```

Testing translate for loop

input	output	Justification	pass
for(;;){ }	for(;;){ }	Simple for loop	Yes

for(let i=0;;){
}
	let var4:number
	var4=0 as number
	for (;;){
	}
		For loop with initialiser	Yes

let i=0
for(;i<100;){
 	
}
	let var4:number
	var4=0 as number
	for (;;){
	if (!((var4)<(100 as number))){
	break
	}
	}
		For loop with break condition	Yes

let i = 0;
for (;;i++){
 	
}
	let var4:number
	var4=0 as number
	for (;;var4++){
	}
		For loop with update expression	Yes

for (var i=0;i<10;)
{
}
	var var4:number
	var4=0 as number
	for (;;){
	if (!((var4)<(10 as number))){
	break
	}
	}
		For loop with initialiser and break condition	Yes

for (var i=0;i<10;i++){
 	
}
	var var4:number
	var4=0 as number
	for (;;var4++){
	if (!((var4)<(10 as number))){
	break
	}
	}
		For loop with initialiser, break condition and update expression	Yes

3.5.8 translate while loop

3.5.9 translate for...in loop

3.5.10 translate for...of loop

3.5.11 translate if statement

3.5.12 translate try statement

3.5.13 translate switch statement

3.6 HIR translate expression

Expressions are nodes that returns a value. This value can be created by different means for example variable read, a function call or a constant value. Expressions are essential to the source code's building block. They represent the majority of a programme. All functions that translate expression returns `Result<(Expr, Type)>`

The returned type is used for type checking on dependent expressions. An optional expected type is passed to the functions to guide the translation process.

3.8 Testing HIR translation: black box testing

HIR testing is done in two different manners. Integrated unit tests and black box testing. The integrated unit tests are documented in the above section for individual components. Boundary tests are not required in this design. The project is coded in a manner such that all access to arrays are done in an iterative loop using Rust's built in Iterator trait. It handles indexes automatically with strict memory access rules such that data in an array can only be mutably borrowed once at a time.

Some functions in the HIR translation stage does not implement an unit test. This is because their logic is straight forward and simple. The black box testing would be use to determine whether a bug occur. Error caused by these functions would be traced back during black box testing. If no errors were found in black box testing, these functions are most likely bugless.

test	description	Type
For in loop	Test transformation of for in loops into a counting loop	Valid
For of loop	Test transformation of for-of loops into a counting and indexing loop	Valid
While loop		Valid
Vehicle	Testing declaration of class and inheritance	Valid
Binary search	Test if a binary search algorithm can be correctly parsed	Valid

3.8.1 Testing for in loops

Testcase: translate for in loops

Justification: for in loops are decomposed into simple operations, correctness of this translation must be tested

input	output

for (let i in []){
i += (99)
}
	var var2:number
	var2=0
	var var0:number
	var0=[].length
	for (;;){
	if ((var0)===(var2)){
	break
	}
	var var3:number
	var3=var2++ as number
	var3+=99
	}

output is as expected. Logic of code is correct according to observation.

3.8.2 Testing for of loops

Testcase: translate for in loops

Justification: for of loops are decomposed into simple operations, correctness of this translation must be tested

input	output

for (let i of [0, 9, 8]){
i+=(99);
}
	var var2:number
	var2=0
	var var0:number[]
	var var1:number
	var0=[0 as number,9 as number,8 as number]
	var1=var0.length
	for (;;){
	if ((var1)===(var2)){
	break
	}
	var var3:number
	var3=var0[var2++]
	var3+=99
	}

output is as expected. Logic of code is correct according to observation.

3.8.3 Testing while loop

Testcase: translate while loop

Justification: for of loops are decomposed into simple operations, correctness of this translation must be tested

input	output	Pass

let i = 0;
 	
while (i < 100){
  i++;
}
	let var0:number
	var0=0 as number
	for (;;){
	  if (((var0)<(100 as number))){
	  break
	  }
	  var0++
	}
		No

What happened	reason
The breaking condition of the loop is incorrect	The break condition of the while loop should be when the comparison is false. However the produced IR code breaks when comparison is true.

Action	Justification
I have looked into the search code where while loop statement is translated. I have added a logical not operation in the breaking condition so that the loop will exit when condition is false.	The error is due to the breaking condition being inverted. By adding a logical not operation, the breaking condition is now inverted and correct.

I have rerun the test again:

input	output	Pass

let i = 0;
 	
while (i < 100){
  i++;
}
	let var0:number
	var0=0 as number
	for (;;){
	  if (!((var0)<(100 as number))){
	  break
	  }
	  var0++
	}
		Yes

Now the output of IR code is as expected, the test passes.

3.8.3 Testing invalid property

Justification: Testing type checks. Object in variable ‘t’ has no property ‘o’

input	output	Pass
let t = {i:0, u: "i"}; t.o = 9;	Syntax Error : type has no property 'o'	Yes

The test is as expected, an error have been returned.

3.8.4 Testing intersection interface

Justification: Syntax check. HIR should be able to declare interfaces. It should also be able to perform interface intersection. It should also be able to declare type aliases. When a conflicted intersection happens, a never type is placed in the field.

input	output	Pass

interface A{
  a: number;
}
 	
interface B{
  a: string
}
 	
type U = A & B;
	interface U{
	  a: never;
	}
		Yes

The output is as expected, The resulting interface has property ‘a’ with type ‘never’.

3.8.5 Testing constructor super call

Justification: Testing HIR validation of constructor. Because a condition statement is used to initialise the parent class ‘Vehicle’, there may be a possibility that the parent class is not initialised before constructor returns.

This should result in an error.

input	output	pass

class Vehicle{
  weight?: number;
}
 	
class Car extends Vehicle{
  constructor(a:boolean){
  if (a){
  super()
  }
  }
}
	Syntax Error: super must be called before returning from a constructor	Yes

The output is as expected, an error have been returned.

3.8.6 Testing array construction

Justification: Testing construction or arrays without type annotation. The compiler should be able to automatically reference types from elements

input	output	pass
`let arr = [10, 80, 30, 90, 40];`	Syntax Error: type ‘number’ cannot be assigned to type ‘0’	No

The test has failed by the compiler returning a syntax error.

What happened	Reason
Type check fails when creating an array with number	Literal numbers are not treated properly as number, this also applies to other literal type.

Action	Justification
Added codes to the translate array expression function. Types are converted to a concrete type during the translation of an array expression	The error is due to the function not being able to recognise literal types. It referenced a literal type as its element type preventing other numbers to be assigned to the array. By converting the literal type in the array expression into number type, the other elements will now be assignable to the array.

We rerun the test to validate the result

input	output	pass
`let arr = [10, 80, 30, 90, 40];`	`[10, 80, 30, 90, 40]`	Yes

Now that the compiler has successfully constructed an array, the test passes.

3.8.7 testing switch cases

Justification: switches in AST is translated into HIR switches, their accuracy must be tested

input	output	pass

switch (0){
  case 1:
  'case 1';
  break;
  default:
  'default case';
}
	Syntax error: type ‘0’ is not comparable to type ‘1’	Yes

When the test was designed, I did not expect an error from the compiler. Initially I thought that the test has failed. However, this behaviour also applies to the current Typescript language. This means that I have implemented something right by accident.

So another test is carried out:

input	output	pass

let i = 99;
switch (i){
  case 1:
  'case 1';
  break;
  default:
  'default case';
  case 7:
  'case 7';
  break;
}
	let var0:number
	var0=99 as number
	switch (var0){
	  case 1:
	  "case 1"
	  break;
	  case 7:
	  "case 7"
	  break;
	  default:
	  "default case"
	}
		Yes

As shown above, the compiler has rearranged the order of cases so that the default case would come last.

3.8.3 testing binary search

Test case: binary search

description: A binary search algorithm is implemented in Typescript is parsed and transformed to HIR. This is a black box testing to make sure the HIR can translate more complex programmes.

input	output

binarySearch([], 0)
 	
function binarySearch(arr: number[], x: number): number
{
let l = 0;
let r = arr.length - 1;
let mid: number;
while (r >= l) {
mid = l + (r - l) / 2;
// If the element is present at the middle
// itself
if (arr[mid] == x)
return mid;
// If element is smaller than mid, then
// it can only be present in left subarray
if (arr[mid] > x)
r = mid - 1;
// Else the element can only be present
// in right subarray
else
l = mid + 1;
}
// We reach here when element is not
// present in array
return -1;
}
	fun1([],0)
	function fun1(this:any, var0:number[], var1:number):number{
	var var2:number
	var2=0 as number
	var var3:number
	var3=(var0.length)-(1) as number
	var var4:number
	for (;;){
	if (!(var3)>=(var2)){
	break
	}
	var4=(var2)+((var3-var2)/2)
	if ((var0[var4])==(var1)){
	return var4
	}
	if ((var0[var4])>(var1)){
	var3=(var4)-1
	}
	else {
	var2=(var4)+(1 as number)
	}
	}
	return -(1)
	}

Output is as expected according to observation

Block scope testing

input	Justification	Pass

try {
  (function(x) {
  try {
  let x = 'inner';
  throw 0;
  } finally {
  assert.sameValue(x, 'outer');
  }
  })('outer');
} catch (e) {}
	finally block let declaration only shadows outer parameter value 1	Yes

(function(x) {
  try {
  let x = 'middle';
  {
  let x = 'inner';
  throw 0;
  }
  } catch(e) {
 	
  } finally {
  assert.sameValue(x, 'outer');
  }
})('outer');
	finally block let declaration only shadows outer parameter value 2	Yes

(function(x) {
  for (var i = 0; i < 10; ++i) {
  let x = 'inner' + i;
  continue;
  }
  assert.sameValue(x, 'outer');
})('outer');
	for loop block let declaration only shadows outer parameter value 1	Yes

(function(x) {
  label: for (var i = 0; i < 10; ++i) {
  let x = 'middle' + i;
  for (var j = 0; j < 10; ++j) {
  let x = 'inner' + j;
  continue label;
  }
  }
  assert.sameValue(x, 'outer');
})('outer');
	for loop block let declaration only shadows outer parameter value 2	Yes

(function(x) {
  label: {
  let x = 'inner';
  break label;
  }
  assert.sameValue(x, 'outer');
})('outer');
	nested block let declaration only shadows outer parameter value 1	Yes

(function(x) {
  label: {
  let x = 'middle';
  {
  let x = 'inner';
  break label;
  }
  }
  assert.sameValue(x, 'outer');
})('outer');
	nested block let declaration only shadows outer parameter value 2	Yes

var caught = false;
try {
  {
  let xx = 18;
  throw 25;
  }
} catch (e) {
  caught = true;
  assert.sameValue(e, 25);
  (function () {
  try {
  // NOTE: This checks that the block scope containing xx has been
  // removed from the context chain.
  assert.sameValue(xx, undefined);
  eval('xx');
  assert(false); // should not reach here
  } catch (e2) {
  assert(e2 instanceof ReferenceError);
  }
  })();
}
assert(caught);
	outermost binding updated in catch block; nested block let declaration unseen outside of block	No

What happened	reason
Block scope testing failed with syntax error “undefined function ‘eval’”	The eval function is not supported by this compiler

Action	Justification
Remove test from the test data	The evaluation feature is not supported. The eval function runs source code during runtime and is not compatible with static codes generated by the compiler.

Input	Justification	Pass

function f() {}
 	
(function(x) {
  try {
    let x = 'inner';
    throw 0;
  } catch(e) {
 	
  } finally {
    f();
    assert.sameValue(x, 'outer');
  }
})('outer');
	verify context in finally block 1	Yes

function f() {}
 	
(function(x) {
  for (var i = 0; i < 10; ++i) {
    let x = 'inner';
    continue;
  }
  f();
  assert.sameValue(x, 'outer');
})('outer');
	verify context in for loop block 2	Yes

function f() {}
 	
(function(x) {
  label: {
    let x = 'inner';
    break label;
  }
  f(); // The context could be restored from the stack after the call.
  assert.sameValue(x, 'outer');
})('outer');
	verify context in labelled block 1	Yes

function f() {}
 	
(function(x) {
  try {
    let x = 'inner';
    throw 0;
  } catch (e) {
    f();
    assert.sameValue(x, 'outer');
  }
})('outer');
	verify context in try block 1	Yes

3.9 HIR interpreter

Added: 24/2/2024

To aid the testing and validation of the HIR and its transformation, an interpreter is implemented. As the number and complexity of test case increases, it is very hard to validate intermediate codes one by one. The interpreter will be able to execute large batch of tests at the same time. It will be able to validate type information, and any other operation that must be contextually correct. We will then be able to focus on tests where the interpreter failed to execute.

The interpreter will be able to create a trace table upon failure to aid the debugging process. It will be focussing on the correctness of execution rather then performance. The implementation will be simple and readable so that any bugs can be fixed easier.

3.9.10 debugging the interpreter

Several bugs have been found when testing the interpreter. It is important that the interpreter to be bug free as it is the test ground for HIR. By examining the result and trace table from the interpreter in a failed test case, we determine if it is a HIR translation error or an interpreter error. This is done by recreating the executable HIR by hand and hand craft a trace table. By comparing hand calculated result of HIR and the interpreter generated trace table, we can determine if it is a HIR error or interpreter error.

If an interpreter error occurred, we rerun the HIR on the interpreter while creating a stack trace. By examining the stack trace and trace table reference we can pin point where the error occurred and what caused the error.

For instance, I have found four errors within the interpreter after running the test cases immediately after implementing the interpreter. The first one is that the programme cursor not being reset after each iteration in a loop causing the programme to break away from the loop even if break conditions are not met. This is fixed by redefining the cursor on each iteration and the problem has been fixed. The second problem is that the else clause of a conditional statement always executes. This is addressed using a flag variable to indicated whether if clause has been executed. If if-clause is executed and else-clause is present, the interpreter will jump cursor to end of else-clause. The third bug is that the scope of a try block does not exit normally. This is due to the lack of cursor jump after a try block. This is fixed by jumping cursor to end of try block after scope exits.

3.9.11 Fixing bugs found in HIR translation

After implementing and using the interpreter as the primary method to debug HIR, a lot of bugs previously unknown has been found. Below is a list of several major issues that have been found.

issue	description	solution
While loop exits before the number of loop expected.	The testing condition of the while loop is incorrect	Added a logical not operation for the breaking condition
Type check fails when creating an array with number	Literal numbers are not treated properly as number, this also applies to other literal type.	Literal types are converted to a concrete type during the translation of an array expression
Parameter of methods in a class cannot be found in scope resulting in syntax error.	The methods and static functions are not hoisted before translation.	A flag is added to check whether function is hoisted. If the function is not hoisted, its parameters are translated.
Default case in a switch is executed before a switch case	The order of switch cases and the default case is not sequential.	Reorder the translation of default case to be done after switch cases.
Variable load failed in error ‘constant variable is not writeable’	A variable write check is conducted when translating variable load.	Remove the write check.
Class types with conflicted attributes were allowed	The class types were not checked against when a class inherits attributes.	Add checks in place where classes were translated.
Variables declared within a for loop head drops along with the loop context	The declaration is treated as if it is within the loop.	Create a new scope specifically for the for loop head

3.10 HIR passes

The HIR passes are post translation processed that cannot be done during translation time. These operations are carried out after HIR is built. These passes mainly involves rule checks against the use of types, variables and statements as well as transformations to simplify the expression. Some of these passes optimises the HIR so that the code forwarded to MIR is optimal.

Below is a list of passes.

name	description
Class default initialiser pass	Creates default initialisers for attributes that have literal types or optional types. This is run before the initialisation pass so that these attributes will not result in an error.
Class constructor pass	Checks that the super constructor is called before ‘this’ is accessed. It also checks for the attributes to be initialised before access in a constructor. Accessing an uninitialised attribute is undefined behaviour and should result in an error.
Constant evaluation pass	Carries out constant evaluations. Expressions that can be computed at compile time is evaluated and replaced.
Dead code elimination pass	Dead code elimination. Eliminates codes that are not reachable in any sense. This reduces the complexity for the following passes.
Variable default initialiser pass	Creates default values for variables. This allows users to declare variable without an initialiser on variables with types that are obvious.
Variable initialisation pass	Checks if variables are accessed before it is assigned to a value. Accessing a variable that is not initialised is undefined behaviour and should result in an error.

3.16 The Runtime

The runtime and the standard library is written as a single module. They are tightly coupled and provide utility functions for the programme to work. The runtime provides core functionalities of the language while the standard library provides usability functions to the user.

The runtime includes a exception handling framework, an asynchronous scheduler and executer and other platform dependent routines. It is not dependent on the standard library.

The standard library provides usability functions for the user. It aims to deliver the ECMA specification and other non-platform dependent APIs wrapped around platform-dependent implementations. This includes networking, file system access etc.

3.16.1 Exception handling

The exception handling mechanism of the runtime implements the Itanium CXX ABI. A personality routine is defined by the runtime along with parsing of dwarf eh data in the language specific area.

We first define an exception class for our language. The exception class is an eight character long identifier. The first four byte indicates the vendor and the last four bytes indicates the language. Our exception class would be “LAM\0NATS”.

```
/// LAM\0NATS
pub const NATIVE_TS_EXCEPTION_CLASS: u64 = u64::from_ne_bytes(['L' as u8,
'A' as u8, 'M' as u8, '\0' as u8, 'N' as u8, 'A' as u8, 'T' as u8, 'S' as u8]);
```

We now have to find a way to decode the dwarf eh action in the language specific area. However this will be implemented later as it is quite complicated. We define a dummy function.

```
/// possible actions
pub enum EhAction{
    None,
    CleanUp(LandingPad),
    Catch(LandingPad),
}

fn get_eh_action() -> Option<EhAction>{
    todo!()
}
```

We now can define our personality routine. The personality routine takes in five parameters according to the Itanium ABI (7) :

`version`
: Version number of the unwinding runtime, used to detect a mis-match between the unwinder conventions and the personality routine, or to provide backward compatibility. For the conventions described in this document, `version` will be 1.

`actions`
: Indicates what processing the personality routine is expected to perform, as a bit mask. The possible actions are described below.

`exceptionClass`
: An 8-byte identifier specifying the type of the thrown exception. By convention, the high 4 bytes indicate the vendor (for instance HP\0\0), and the low 4 bytes indicate the language. For the C++ ABI described in this document, the low four bytes are C++\0.

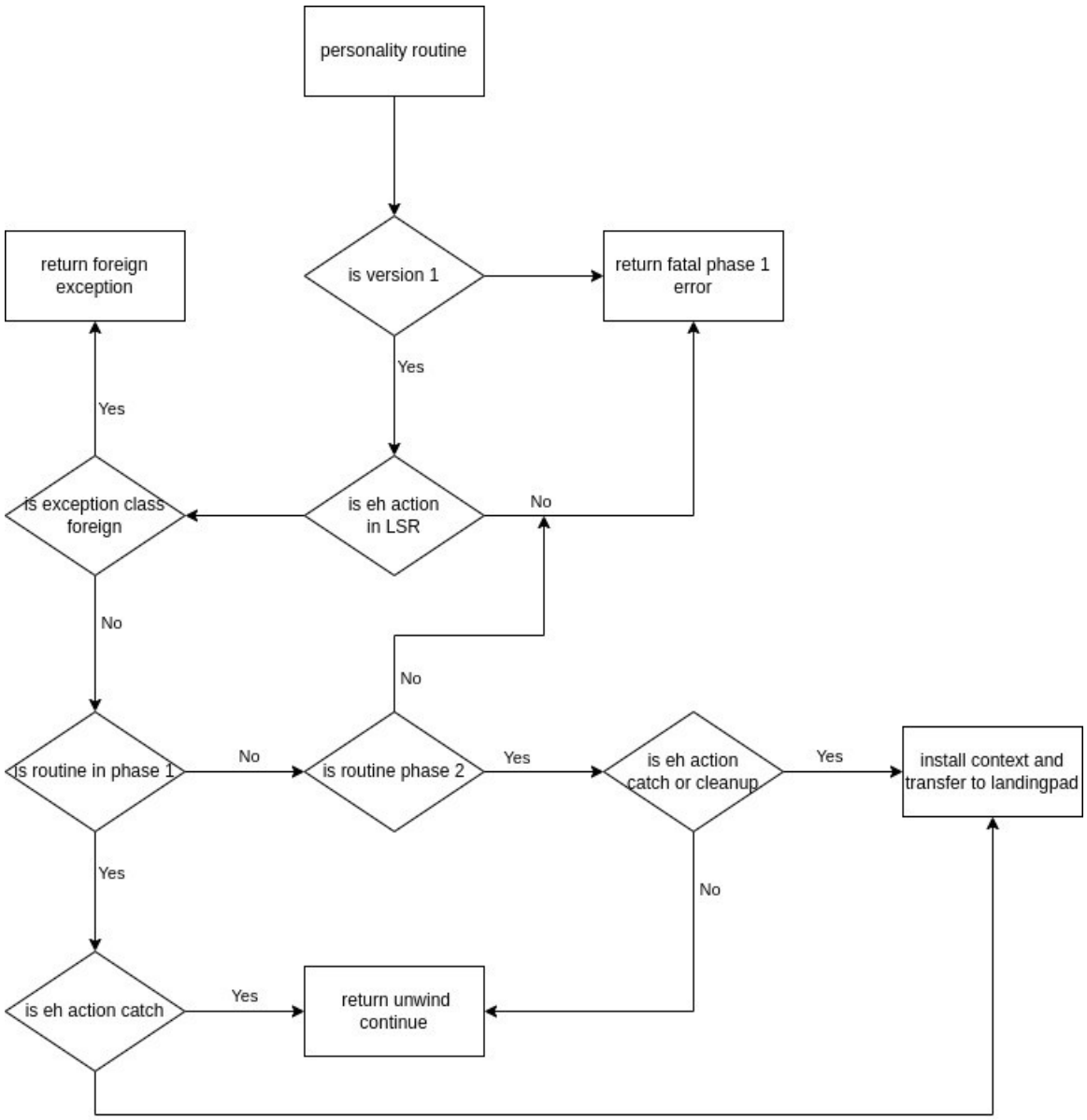
`exceptionObject`
: The pointer to a memory location recording the necessary information for processing the exception according to the semantics of a given language (see the *Exception Header* section above).

`context`
: Unwinder state information for use by the personality routine. This is an opaque handle used by the personality routine in particular to access the frame's registers (see the *Unwind Context* section above).

The return value from the personality routine indicates how further unwind should happen, as well as possible error conditions.

Reason code	code	description
Foreign Exception caught	1	This indicates that a different runtime caught this exception. Nested foreign exceptions, or rethrowing a foreign exception, result in undefined behaviour.
Fatal Phase 1 error	2	The personality routine encountered an error during phase 1, other than the specific error codes defined.
Fatal Phase 2 error	3	The personality routine encountered an error during phase 2, for instance a stack corruption.
End of stack	5	The stack has reached its end, no further unwinding should be performed.
Handler found	6	Exception Handler found in current frame, only valid in the search phase
Install Context	7	Handler is present at phase 2. The personality routine has installed the context and transfers to landing pad
Continue unwind	8	Continues unwind of the stack

The personality routine will first check the version of the unwind library to ensure compatibility. It will not support any version other then 1. It then checks for the exception class. If the exception class is not our exception class, this means that the exception is foreign and we will ignore it by returning. We will then check if the routine is at phase 1 (searching phase) or phase 2(clean up phase). In phase 1, if the eh action is a catch, return handler found, else return continue unwind. In phase 2, if the eh action is not none, we install the context of landing pad and transfer control to the landing pad, otherwise, return continue unwind.



```
// version of the unwind library must be 1
if version != 1{
    return UnwindReasonCode::FATAL_PHASE1_ERROR
}
// check the exception class
if exception_class != NATIVE_TS_EXCEPTION_CLASS{
   // ignore foreign exception
   return UnwindReasonCode::CONTINUE_UNWIND
}
// get the eh action from LSR
let action = match get_eh_action(){
    Some(action) => action,
    None => return UnwindReasonCode::FATAL_PHASE1_ERROR
};
// in the search phase, phase 1
if actions.contains(UnwindAction::SEARCH_PHASE){
   match action{
     // a handler is found
     NativeTsAction::Catch(_) => return UnwindReasonCode::HANDLER_FOUND,
     // no handler found
     _ => return     UnwindReasonCode::CONTINUE_UNWIND,
   }
}
match action{
   // no action is required
   NativeTsAction::None => return UnwindReasonCode::CONTINUE_UNWIND,
   // setup the context and transfer to landingpad
   NativeTsAction::Catch(landingpad) |
   NativeTsAction::CleanUp(landingpad) => {
       // set the ip to the landing pad
       unwinding::abi::_Unwind_SetIP(context, landingpad.ip);

       // define the registers
       #[cfg(target_arch = "x86_64")]
       let regs = (gimli::X86_64::RAX, gimli::X86_64::RDX);

       // forward the exception
       unwinding::abi::_Unwind_SetGR(context, regs.0.0 as _, exception);
       unwinding::abi::_Unwind_SetGR(context, regs.1.0 as _, 0);

       return UnwindReasonCode::INSTALL_CONTEXT
   }
}
```

After implementing the personality routine we have to implement a stack frame reader that reads the EH action as defined in the dwarf standard. The EH action code give a hint to the personality routine about what the stack frame is up to. It also provides information necessary to land on a landing pad.

Pointers are encoded in the dwarf EH frames. Its encoding is also stored on the frame therefore decoding the values is necessary. The encoding method is encoded in two 4bit values occupying one byte in total. The first four bits describes the format of pointer while the last four bit describes how the encoding should be applied.

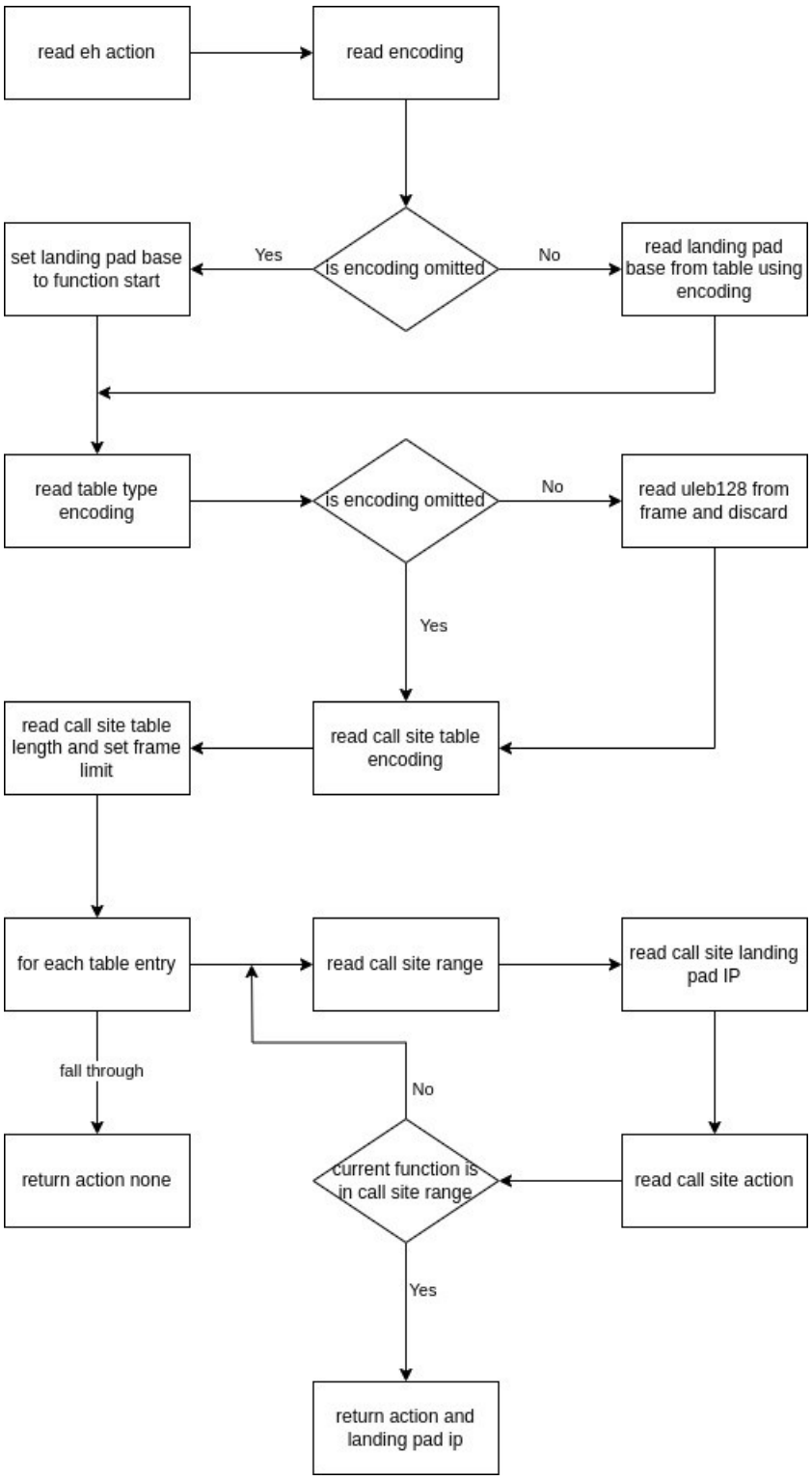
format	Value	description
absptr	0x00	The Value is a literal pointer whose size is determined by the architecture.
uleb128	0x01	Unsigned value is encoded using the Little Endian Base 128 (LEB128)
udata2	0x02	A 2 bytes unsigned value.
udata4	0x03	A 4 bytes unsigned value.
udata8	0x04	An 8 bytes unsigned value.
sleb128	0x09	Signed value is encoded using the Little Endian Base 128 (LEB128)
sdata2	0x0A	A 2 bytes signed value.
sdata4	0x0B	A 4 bytes signed value.
sdata8	0x0C	An 8 bytes signed value.

application	Value	description
Pc relative	0x10	Value is relative to the current program counter.
Text relative	0x20	Value is relative to the beginning of the .text section.
Data relative	0x30	Value is relative to the beginning of the .got or .eh_frame_hdr section.
Func relative	0x40	Value is relative to the beginning of the function.
aligned	0x50	Value is aligned to an address unit sized boundary.

We define a function to decode the pointer value stored in the frame.

```
fn parse_encoded_pointer(
    encoding: constants::DwEhPe,
    ctx: &UnwindContext<'_>,
    input: &mut StaticSlice,
) -> gimli::Result<Pointer> {
    if encoding == constants::DW_EH_PE_omit {
        return Err(Error::CannotParseOmitPointerEncoding);
    }

    let base = match encoding.application() {
        constants::DW_EH_PE_absptr => 0,
        constants::DW_EH_PE_pcrel => input.slice().as_ptr() as u64,
        constants::DW_EH_PE_textrel => _Unwind_GetTextRelBase(ctx) as u64,
        constants::DW_EH_PE_datarel => _Unwind_GetDataRelBase(ctx) as u64,
        constants::DW_EH_PE_funcrel => _Unwind_GetRegionStart(ctx) as u64,
        constants::DW_EH_PE_aligned => return
Err(Error::UnsupportedPointerEncoding),
        _ => unreachable!(),
};

let offset = match encoding.format() {
    constants::DW_EH_PE_absptr =>    input.read_address(size_of::<usize>()),
    constants::DW_EH_PE_uleb128 => input.read_uleb128(),
    constants::DW_EH_PE_udata2 => input.read_u16().map(u64::from),
    constants::DW_EH_PE_udata4 => input.read_u32().map(u64::from),
    constants::DW_EH_PE_udata8 => input.read_u64(),
    constants::DW_EH_PE_sleb128 => input.read_sleb128().map(|a| a as u64),
    constants::DW_EH_PE_sdata2 => input.read_i16().map(|a| a as u64),
    constants::DW_EH_PE_sdata4 => input.read_i32().map(|a| a as u64),
    constants::DW_EH_PE_sdata8 => input.read_i64().map(|a| a as u64),
    _ => unreachable!(),
}?;

let address = base.wrapping_add(offset);
    Ok(if encoding.is_indirect() {
        Pointer::Indirect(address)
    } else {
        Pointer::Direct(address)
    })
}
```

To find the EH action from the EH frame, we must look into the frame structurally. We first get an encoding from the beginning of the frame. This encoding will tell us whether the next field which is the base address of the landing pad. It then reads the type encoding and the type length. It then reads the call site encoding and the call site table length. It then loops through the frame to read call site ranges and its corresponding EH action. If an EH action is found, it is immediately returned. Otherwise, action None is returned.



Now that we have defined our personality routine, we must export a standard function to throw exceptions from the runtime to accommodate the user programme. This includes five functions:

```
extern fn __native_ts_allocate_exception(value: Any) -> *mut NativeTsException
```

This function allocates an exception with the given value as the user error. Unlike normal allocation, the memory allocate must be freed manually.

```
extern "C" fn __native_ts_free_exception(exception: &mut NativeTsException)
```

This function deallocate the exception. This should be called after the exception is caught.

```
pub extern "C" fn __native_ts_throw(exception: &mut NativeTsException)
```

This function provides an entry point for the programme to raise an exception. It will doe the following:

- Increment the uncaught exception flag by one
- set the exception class in the header to “LAM\0NATS”
- call _Unwind_RaiseException to raise exception.

```
pub extern "C" fn __native_ts_begin_catch(exception: &mut NativeTsException)
```

This function is called when a catch clause enters. It performs the following:

- increment the handler count by one
- decrement uncaught exception count by one
- push exception to a stack with caught exceptions

```
pub extern "C" fn __native_ts_end_catch()
```

This function is called when a catch clause exits. It performs the following:

- decrement the handler count by one on the most recent exception
- pop the exception from stack if its handler count goes to zero
- destroys the exception if handler count is zero and not being re thrown

Part 4
Evaluation

4.1 Testing for evaluation

4.1.1 testing real world algorithms

Test case: binary search

input	data	search	expected output
source code (see section 2)	[0, 8, 9, 10, 11, 12, 13, 14]	12	5

```
    Finished `test` profile [unoptimized + debuginfo] target(s) in 8.31s
     Running tests/binary_search.rs (target/debug/deps/binary_search-fe2a9dfc9cfaf87e)

running 1 test
fun2([0,8,9,10,11,12,13,14],12)
function fun1(this:any, var0:number):number{
    let var3:number
    var3=(var0)%(1)
    let var4:number
    var4=(var0)-(var3)
    if ((var3)>=(0.5)){
        return (var4)+(1)
    }
    return var4
}
function fun2(this:any, var1:number[], var2:number):number{
    let var5:number
    var5=0 as number
    let var6:number
    var6=(var1.length)-(1) as number
    let var7:number
    for (;;){
        if (!((var6)>=(var5))){
            break
        }
        var7=(var5)+(fun1(((var6)-(var5))/(2)))
        if ((var1[var7])==(var2)){
            return var7
        }
        if ((var1[var7])>(var2)){
            var6=(var7)-(1)
        }
        else {
            var5=(var7)+(1)
        }
    }    return -(1)
}

Ok(Number(5.0))
test binary_search ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

*  Terminal will be reused by tasks, press any key to close it.
```

The translated high-level IR is printed on the command prompt.

The result of translating source code into HIR is as expected.

The result of running the HIR in the interpreter has matched our expectation. The binary search algorithm works in HIR.

This test has demonstrated robustness of the compiler against real world application.

Test case: ternary search

input	data	search	expected output
source code (see section 2)	[2, 3, 6, 9,10,11,13,17,23]	13	6

```
running 1 test
function fun1(this:any, var0:number):number{
    let var5:number
    var5=(var0)%(1)
    let var6:number
    var6=(var0)-(var5)
    if ((var5)>=(0.5)){
        return (var6)+(1)
    }
    return var6
}
function fun2(this:any, var1:number, var2:number, var3:number, var4:number[]):number{
    if ((var2)>=(var1)){
        let var7:number
        var7=(var1)+(fun1(((var2)-(var1))/(3)))
        let var8:number
        var8=(var2)-(fun1(((var2)-(var1))/(3)))
        if ((var4[var7])==(var3)){
            return var7
        }
        if ((var4[var8])==(var3)){
            return var8
        }
        if ((var3)<(var4[var7])){
            return fun2(var1,(var7)-(1),var3,var4)
        }
        else {
            if ((var3)>(var4[var8])){
                return fun2((var8)+(1),var2,var3,var4)
            }
            else {
                return fun2((var7)+(1),(var8)-(1),var3,var4)
            }
        }
    }
    return -(1)
}
let var9:number[]
var9=[2,3,6,9,10,11,13,17,23]
let var10:number
var10=13 as number
fun2(0,(var9.length)-(1) as number,var10,var9)

Ok(Number(6.0))
test test_ternery_search ... ok
```

The translated high-level IR is printed on the command prompt.

The result of translating source code into HIR is as expected.

Output: 6.0

The result of running the HIR in the interpreter has matched our expectation. The ternary search algorithm works in HIR.

This test has demonstrated robustness of the compiler against real world application.

Test case: fibonacci search

input	data	search	expected output
source code (see section 2)	[10,22,35,40,45,50,80,82,85,90,100,235]	235	11

```
        }
        var5=var6
        var6=var7
        var7=(var5)+(var6)
    }    let var8:number
    var8=-(1)
    for (;;){
        if (!((var7)>(1 as number))){
            break
        }
        let var9:number
        var9=fun1((var8)+(var5),(var3)-(1))
        if ((var2[var9])<(var4)){
            var7=var6
            var6=var5
            var5=(var7)-(var6)
            var8=var9
        }
        else {
            if ((var2[var9])>(var4)){
                var7=var5
                var6=(var6)-(var5)
                var5=(var7)-(var6)
            }
            else {
                return var9
            }
        }
    }    if ((var6 as boolean)&&((var2[(var3)-(1)])==(var4))){
        return (var3)-(1)
    }
    return -(1)
}
let var10:number[]
var10=[10,22,35,40,45,50,80,82,85,90,100,235]
let var11:number
var11=var10.length as number
let var12:number
var12=235 as number
fun2(var10,var11,var12)

Ok(Number(11.0))
test test_fib_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out; finished in 0.00s
```

The translated high-level IR is printed on the command prompt.

The result of translating source code into HIR is as expected.

Output: 11.0

The result of running the HIR in the interpreter has matched our expectation. The fibonacci search algorithm works in HIR.

This test has demonstrated robustness of the compiler against real world application.

Test case: bubble sort

input	data	expected output
source code (see section 2)	[234, 43, 55, 63, 5, 6, 235, 547]	[5, 6, 43, 55, 63, 234, 235, 547]

```
warning: `native-ts-hir` (lib) generated 36 warnings (run `cargo fix --lib -p native-ts-hir` to apply 1 suggestion)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.27s
     Running tests/bubble_sort.rs (target/debug/deps/bubble_sort-ca39a522130526be)

running 1 test
function fun1(this:any, var0:number[]):number[]{
    var var1:number
    var1=0 as number
    for (;;){
        if (!((var1)<(var0.length as number))){
            break
        }
        var var2:number
        var2=0 as number
        for (;;){
            if (!((var2)<(((var0.length as number)-(var1))-(1)))){
                break
            }
            if ((var0[var2])>(var0[(var2)+(1)])){
                var var3:number
                var3=var0[var2]
                var0[var2]=var0[(var2)+(1)]
                var0[(var2)+(1)]=var3
            }
            var2++
        }        var1++
    }    return var0
}
var var4:number[]
var4=[234,43,55,63,5,6,235,547]
fun1(var4)

Ok(Array(RwLock { data: [Number(5.0), Number(6.0), Number(43.0), Number(55.0), Number(63.0), Number(234.0), Number(235.0), Number(547.0)]
 }))
test test_bubble_sort ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

 *  Terminal will be reused by tasks, press any key to close it.
```

The translated high-level IR is printed on the command prompt.

The result of translating source code into HIR is as expected.

Output: [5, 6, 43, 55, 63, 234, 235, 547]

The result of running the HIR in the interpreter has matched our expectation. The bubble sort algorithm works in HIR.

This test has demonstrated robustness of the compiler against real world application.

Test case: quick sort

input	data	expected output
source code (see section2)	[10, 80, 30, 90, 40]	[10, 30, 40, 80, 90]

```
   Compiling native-ts-hir v0.1.0 (/home/yc/Documents/GitHub/native-js/native-ts-hir)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 2.55s
     Running tests/sort.rs (target/debug/deps/sort-95a769174afab2f6)

running 1 test
function fun1(this:any, var0:number[], var1:number, var2:number):number{
    let var6:number
    var6=var0[var2]
    let var7:number
    var7=(var1)-(1)
    let var8:number
    var8=var1
    for (;;){
        if (!((var8)<=((var2)-(1)))){
            break
        }
        if ((var0[var8])<(var6)){
            var7++
            [var0[var7]=__pushstack__([var0[var8],var0[var7]])[0],var0[var8]=__popstack__()[1]]
        }
        var8++
    }    [var0[(var7)+(1)]=__pushstack__([var0[var2],var0[(var7)+(1)]])[0],var0[var2]=__popstack__()[1]]
    return (var7)+(1)
}
function fun2(this:any, var3:number[], var4:number, var5:number):undefined{
    if ((var4)>=(var5)){
        return undefined
    }
    let var9:number
    var9=fun1(var3,var4,var5)
    fun2(var3,var4,(var9)-(1))
    fun2(var3,(var9)+(1),var5)
}
let var10:number[]
var10=[10,80,30,90,40]
fun2(var10,0,(var10.length)-(1) as number)
var10

Ok(Array(RwLock { data: [Number(10.0), Number(30.0), Number(40.0), Number(80.0), Number(90.0)] }))
test test_quick_sort ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished in 0.00s

*  Terminal will be reused by tasks, press any key to close it.
```

The translated high-level IR is printed on the command prompt.

The result of translating source code into HIR is as expected.

Output: [10, 30, 40, 80, 90]

The result of running the HIR in the interpreter has matched our expectation. The quick sort algorithm works in HIR.

This test has demonstrated robustness of the compiler against real world application.

Test case: insertion sort

input: source code (see section 2)

input data: [9, 6, 3, 4, 3, 12, 4]

expected output: [3, 3, 4, 4, 6, 9, 12]

```
   Compiling native-ts-hir v0.1.0 (/home/yc/Documents/GitHub/native-js/native-ts-hir)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 2.43s
     Running tests/sort.rs (target/debug/deps/sort-95a769174afab2f6)

running 1 test
function fun1(this:any, var0:number[]):number[]{
    let var1:number
    var1=var0.length as number
    let var2:number
    var2=0 as number
    let var3:number
    var3=0 as number
    let var4:number
    var4=0 as number
    for (;;){
        if (!((var4)<(var1))){
            break
        }
        var2=var0[var4]
        var3=(var4)-(1)
        var3
        for (;;){
            if (!(((var3)>=(0 as number))&&((var0[var3])>(var2)))){
                break
            }
            var0[(var3)+(1)]=var0[var3]
            --var3
        }        var0[(var3)+(1)]=var2
        ++var4
    }    return var0
}
fun1([9,6,3,4,3,12,4])

Ok(Array(RwLock { data: [Number(3.0), Number(3.0), Number(4.0), Number(4.0), Number(6.0), Number(9.0), Number(12.0)] }))
test test_insertion_sort ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 3 filtered out; finished in 0.00s

 *  Terminal will be reused by tasks, press any key to close it.
```

The translated high-level IR is printed on the command prompt.

The result of translating source code into HIR is as expected.

Output: [3, 3, 4, 4, 6, 9, 12]

The result of running the HIR in the interpreter has matched our expectation. The insertion sort algorithm works in HIR.

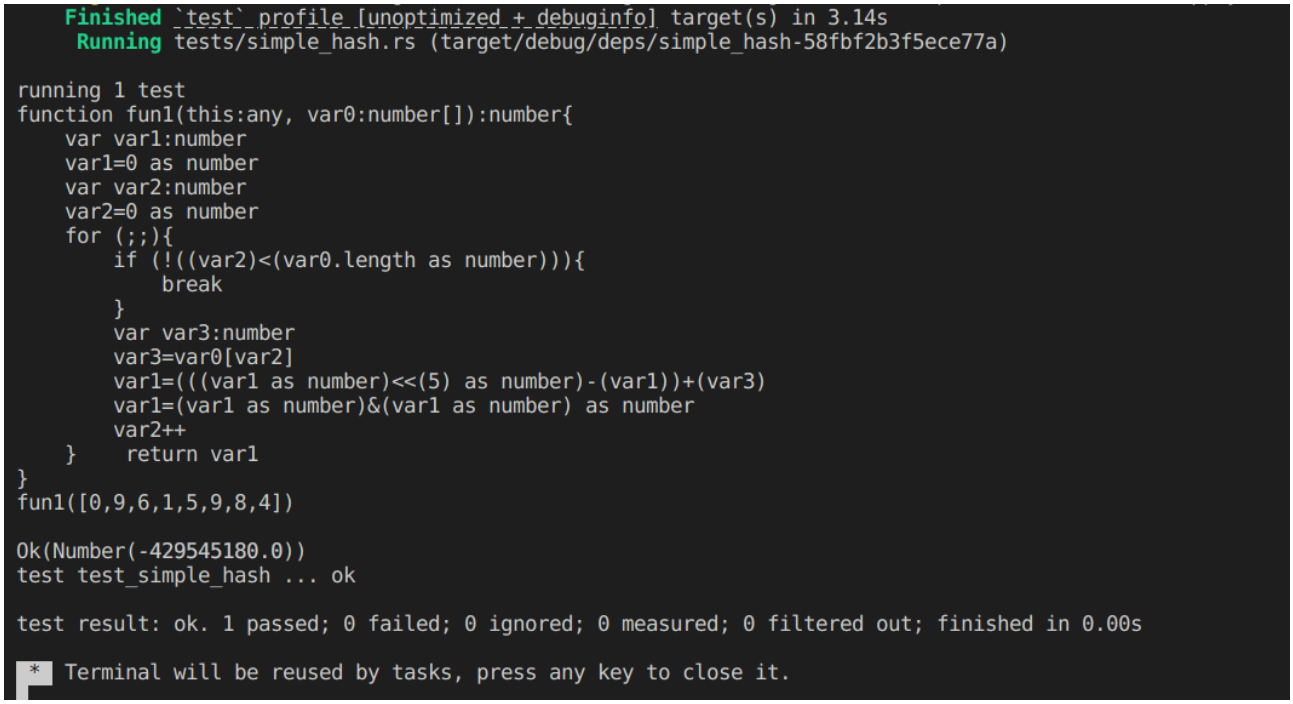
This test has demonstrated robustness of the compiler against real world application.

Test case: simple hash

input: source code (see section 2)

input data: [0, 9, 6, 1, 5, 9, 8, 4]

expected output: -429545180



The translated high-level IR is printed on the command prompt.

The result of translating source code into HIR is as expected.

Output: -429545180.0

The result of running the HIR in the interpreter has matched our expectation. The hash algorithm works in HIR.

This test has demonstrated robustness of the compiler against real world application.

4.2 Overview

This project has been overall successful archiving several criteria. The end result can successfully parse source code up to ECMA 2022 standard according to ECMA-262. It can translate most intended Typescript Syntax and carry out Type check following the Typescript standard.

Translating from HIR to MIR is however, only partially implemented and not yet tested. This is due to the time limitations given that the project is done in six months. More time is required to implemented this part.

Despite not being able to generating machine code, we can still make sure that the compiler worked as expected by implementing an interpreter that runs on HIR codes. The accuracy and reliability of the first stage has been tested and carried out by the interpreter.

The translation from AST to HIR is definitely the most complex and code intense section of this compiler as it incorporates all the heavy lifting work such as type checking and code analyse. With the HIR stage being implemented, this project have a cover rate of over 70%. The rest of the compiler implementation would very straight forward. This is because the translation from AST to HIR has filtered out all the edge cases and guaranteed consistency.

There are also several features that are missing in this project. Generic types are not supported and will fall to panic when compiled. This is due to the complexity posed by the nature of types in Typescript. The implementation of generic types will introduce significantly more uncertainties in the HIR translation process. This will not only affect the current type checking process. More pose translation analysis and normalisation on the HIR will be required. Generic types in HIR would mean that the concrete type is unknown during translation time. A second Type checking mechanism must be introduced to solve generics. The decision made during the iterative process of development is to not implement generic types for now but instead placing placeholders that will panic. Generic types will be implemented in the future when the solution is carefully considered.

4.3 Success criteria

Below is a list of criteria we have mentioned in section 1:

criteria	success	description
parse Typescript and ECMA source code up to 2023 edition	Yes	The parser used in this project is a third-party library that meets the standard.
identify syntax errors and report to user during parsing	Yes	Syntax error is detected by the library during parsing. If a syntax error occurs, the parsing process will stop. The error is then emitted to the command line interface.
identify modules of a project and its required dependencies	Yes	The identification of the dependencies are done after parsing. The dependency resolver can import source code files from either the local file system or from the internet using HTTP requests.
identify exported symbols of a module.	Yes	This process is done alone with module level hoisting where the global symbols are hoisted. The exported symbols are then registered to a table and can be referenced by other modules.
Able to reference and link modules	Yes	The modules are able to reference symbols from each other using global unique identifiers.
translate AST into HIR.	Yes	The AST tree can be transformed into HIR where expressions and statements are simplified. Some expressions however are not supported due to various reasons either them being obsolete or that the feature cannot be incorporated into static compilation. Some inaccuracy in the translate may occur. This will be sorted out in the future with more development time and more testing. Testing has been done in the translation. An interpreter has been implemented to further verify these testing where some bugs has been found in test cases that are not documented. This will be sorted in the future. It is somewhat successful in regards of overall translation with some details left to be addressed.
perform type checks and	Yes	The automatic conversion of types have been implemented alone the type checking process: when a

automatic type conversions in HIR.		type checks happens and the types are compatible yet not same with each other, the conversion expression is injected to the HIR.
normalise generic representations within AST.	No	Several attempts have been made to implement support for generic types. However none have been successful. This is a core feature of the Typescript language and must be implement to fully support the language and the standard library. With the experience gained in previous attempts, this will be implemented in the future.
perform type checks in HIR.	Yes	Type checking is done during the translation from AST to HIR. The type checking process is fully integrated into the traversal of the AST nodes while translating.
translate HIR into MIR.	Partial	Some progress has been made to translate from HIR into MIR. All the expressions can be translated by now. However some HIR statements are not yet implemented which means that the whole translation process is only partially implemented. This is due to time limit on the project. The translation process is fully tested and verified. This is mentioned in part 3.
represent types such as interface, generators and promises in MIR.	Yes	MIR is fully able to represent any type defined in HIR.
declare functions, structures and construct virtual tables in MIR.	Yes	MIR is fully able to declare functions in a given MIR context. The virtual tables are also constructable using type information built during the translation process.
integrate memory management strategy in MIR.	No	Since several passes in MIR is not yet implemented, The only memory management supported is the garbage collector. This will be addressed in the future
decompose async operations into lower level operations.	Yes	By defining the asynchronous framework api in the MIR, async operation can be decomposed into sections of function calls. An async task is a function with counter and heap allocated variables.

decompose generator operations into lower level operations.	No	The implementation detail of a generator has not yet been decided. There are loads of solutions of how to implement a generator. However, every single solution has its catch which make decisioning very hard. It is also due to time limit that this feature cannot be implemented.
translate MIR into LLVM IR.	No	Currently, the translation from HIR to MIR is only partially finished. Therefore the exact format of the generated MIR is not yet finalised. The translation from MIR to LLVM IR is therefore not yet implemented. However this can be easily implemented compared to the previous two translation process. This is because MIR and LLVM IR shares a similar structure in SSA format. The main difference between the two is that MIR is higher level with operations such as await and dynamic types such as interfaces. The translation is straight forward and will be implemented in the future.
compile LLVM IR into targeted machine code.	Yes	LLVM can compile LLVM IR into machine code.
link object files into executables.	Yes	LLVM ships with its own linker that can link object files into executables
link runtime to object files.	Yes	The static library binary compiled from the runtime is provided to the linker during object linking.
perform garbage collection during runtime.	Partial	A garbage collector has been written for the runtime. It is a conservative garbage collector that does not require any information from the compiler. However, the garbage collector would sometimes not be able to find addresses that are on the the call stack when running in optimised code. This causes segment fault errors to happen leading to a crash. For now, we have disabled garbage collection, the memory allocated in a programme will not be released. In the future, a rewrite of the garbage collector should be done in order to solve this problem.

provide built in functions in runtime.	No	This features is not yet implemented. This is due to several reasons. Generics are not yet implemented meaning that it cannot be utilised for now. It is also due to time limitations that it cannot be implemented.
perform type reflections during runtime.	Yes	Reflection of types are supported by the runtime. This is implemented by using static type information stored in the binary file that are generated during translation of MIR.
Able to handle exceptions during runtime.	Yes	Exception handling is supported by the runtime. The Itanium CXX ABI is implemented to accommodate debugging and foreign function compatibility.

4.3.1 Parsing source code

Source code parsing relies on a third party library named SWC. The SWC project follows the latest up to date version of ECMA and Typescript standard including experimental and not yet stable language features. The SWC project has included a range of tests. These test are provided by the ECMA group: test262. The test results are shown to be covering 96% of the specification. Given that modern browser such as chrome and Firefox has a coverage of 98%, the parser library has a very high coverage comparable to modern browsers.

4.3.2 identifying and resolving dependencies

Dependencies can be identified by the compiler. The dependencies are then imported from the local directory or from an online repository. The compiler is able to retrieve source codes from a local file or an online repository. This is tested and proven to work at section 3.1.7. In the future, a package manager may be introduced to be able to perform version control, packaging and distribution of the source code project.

Cyclic detection is implemented. The algorithm implemented to detect cycles has been tested and proven to work in section 3.1.5. In the future, cyclic detection may be removed. This is because cyclic dependencies can be resolved using more complicated solutions.

4.3.3 function, variable and type exports

Functions, variables and types can be exported from a module and imported by its dependent. This is done by module level hoisting in the HIR stage. All dependencies are hoisted and translated into HIR before its dependent. This allows Type checking to be performed on the fly. This is tested in section 3.7.2 and section 3.7.4. Importing and Exporting is therefore fulfilled.

4.3.4 Translating AST to HIR

The compiler is able to translate AST into HIR. HIR is a simplified version of the AST. The HIR reduces the total variants of nodes from 240 in AST to 60 in HIR. Most representation in AST can be represented in HIR except several expressions that are not supported. For instance, the non standard `argument` expression is not supported and will be identified as a variable.

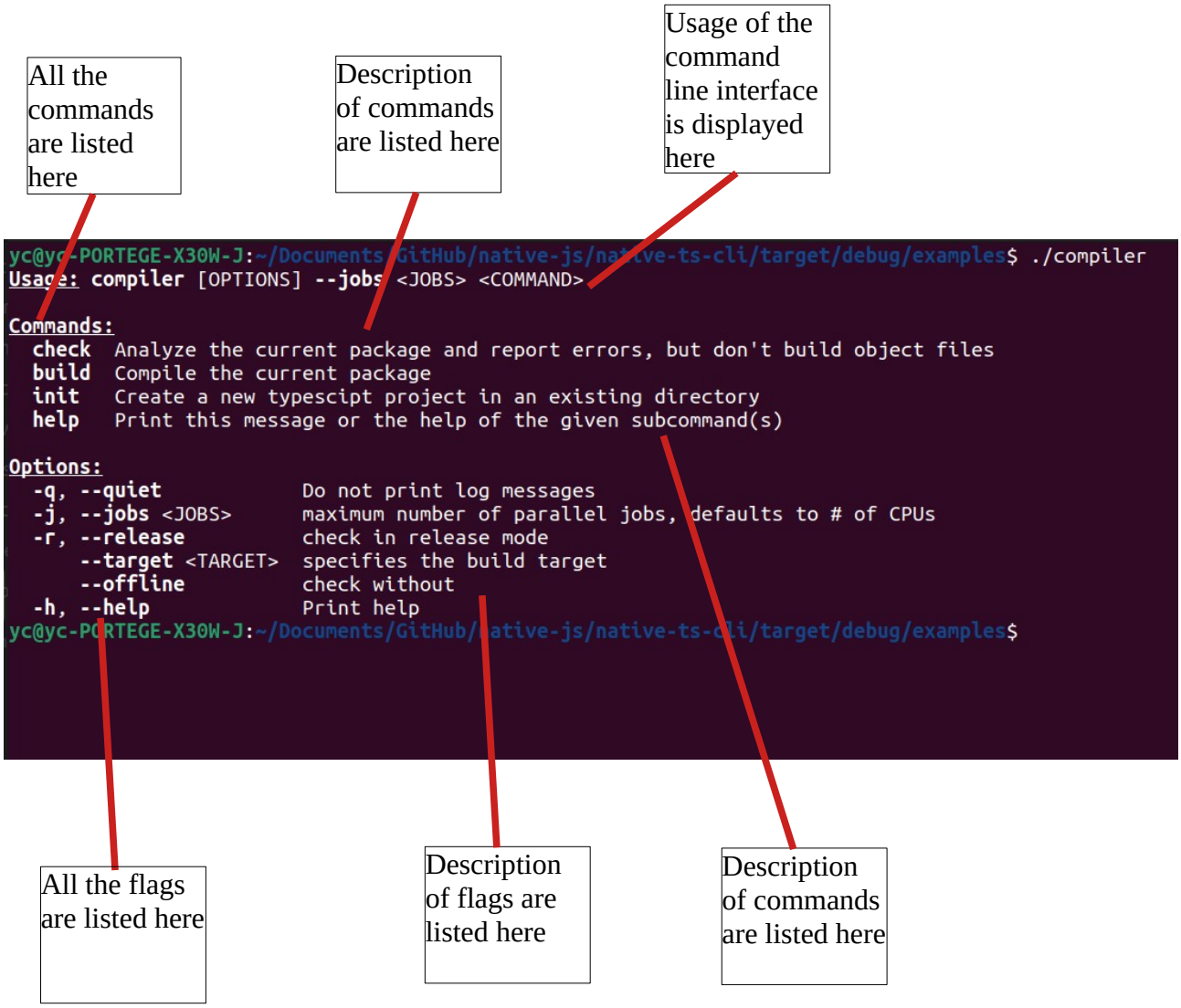
Unit tests were performed within each critical function of component. Most of these tests have been a success. A separated white box test is used to test each component individually. These tests are listed in section 3.3.16, 3.4.17, 3.5.17 and 3.6.17. These test have mostly passed meaning that the majority of component is working correctly. A black box test is implemented to test the overall translation process. This is described in section 3.8. Although some tests have failed when initially performed, the bugs were found and immediately fixed.

In testing for evaluation, real world algorithms written in Typescript is translated into HIR through the compiler and executed by the interpreter, the result of these testing have been successful. Several algorithm written in Typescript is fed to the compiler and translated into HIR. The HIR is then fed to the compiler to be executed and retrieves the result. Out of all the algorithms ran on the interpreter none have failed. These algorithms includes binary search, bubble sort, merge sort, quick sort and a simple hash algorithm. This has proven that the compiler is suitable for real world application

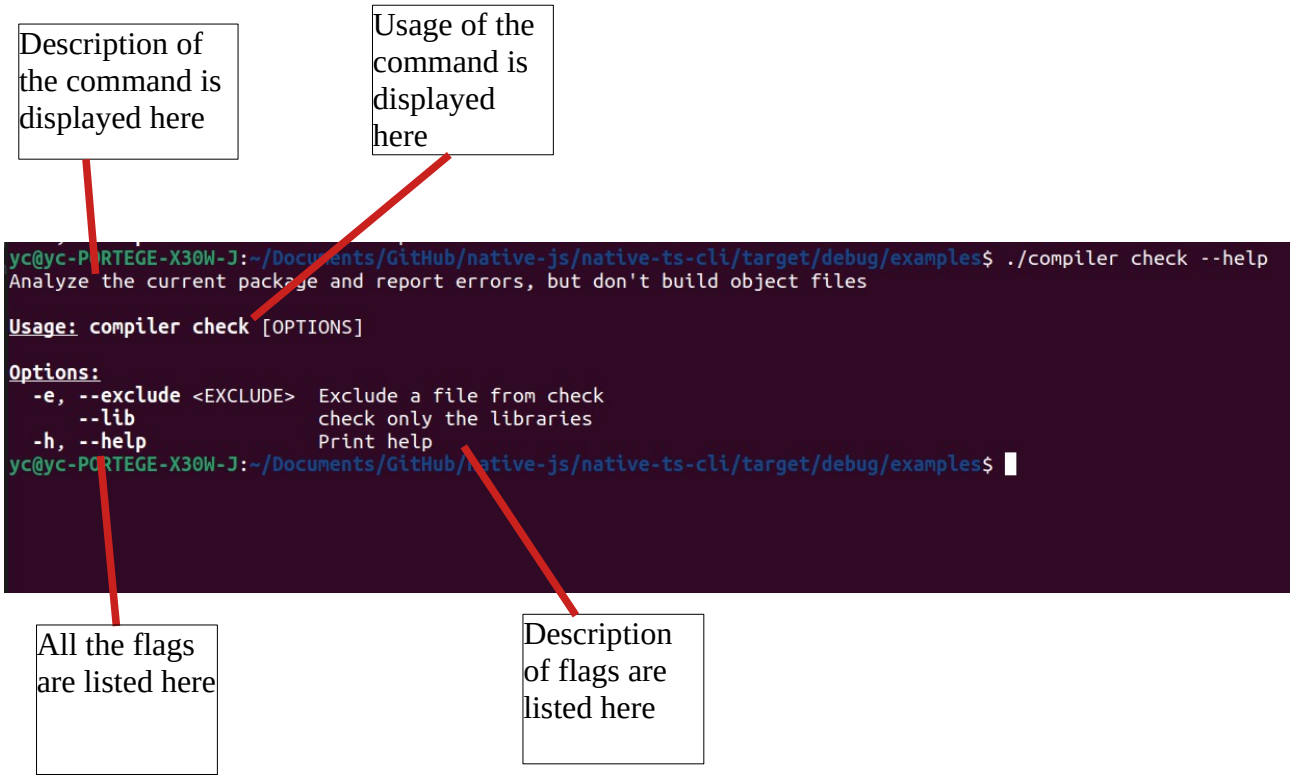
4.4 Usability features

The compiler provides a command line interface consist of only three commands. The three commands are: build, check and init. The command line is very easy to understand: `build` performs checking and compiles the source code into machine code, `check` performs checking and report any errors or warnings, `init` creates a project directory with a configuration file with default settings.

The command line also provides a helper to assist user.

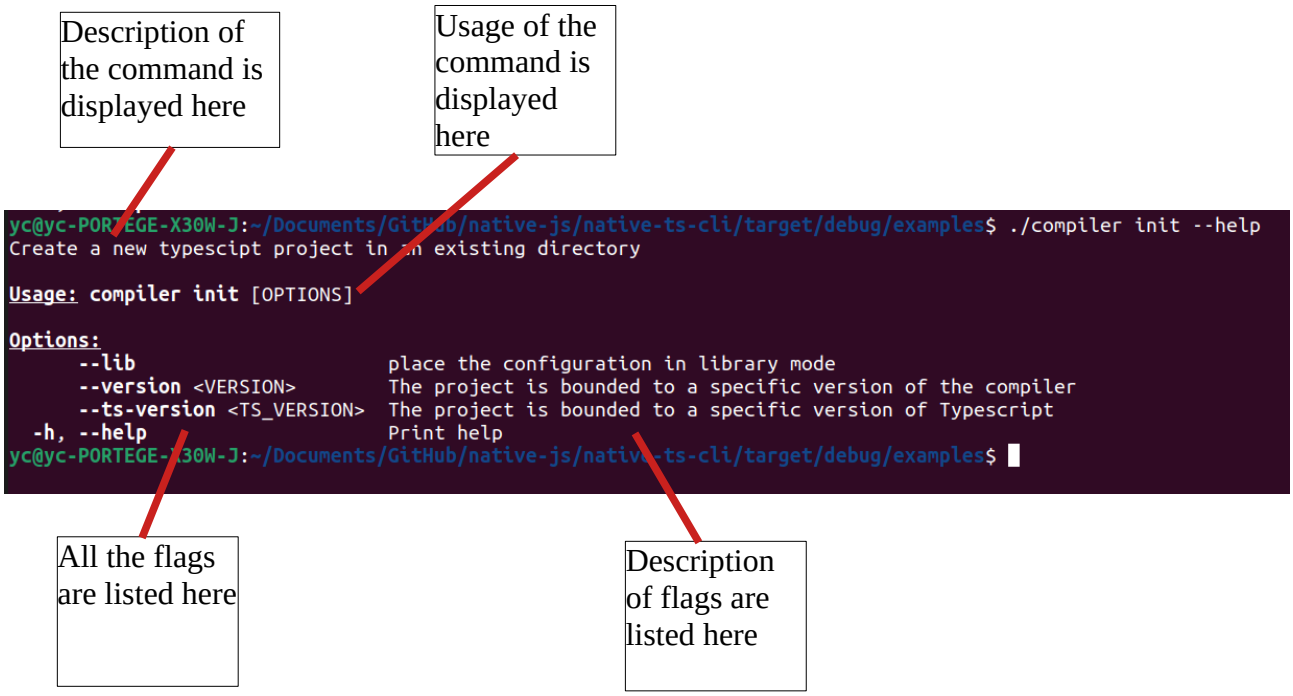


The check command helper



The build command helper

The init command helper



A complete view of the command line helpers is shown below:

```
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$ ./compiler
Usage: compiler [OPTIONS] --jobs <JOBS> <COMMAND>

Commands:
  check  Analyze the current package and report errors, but don't build object files
  build  Compile the current package
  init   Create a new typescipt project in an existing directory
  help   Print this message or the help of the given subcommand(s)

Options:
  -q, --quiet            Do not print log messages
  -j, --jobs <JOBS>      maximum number of parallel jobs, defaults to # of CPUs
  -r, --release          check in release mode
      --target <TARGET>  specifies the build target
      --offline          check without
  -h, --help             Print help
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$ ./compiler check --help
Analyze the current package and report errors, but don't build object files

Usage: compiler check [OPTIONS]

Options:
  -e, --exclude <EXCLUDE>  Exclude a file from check
      --lib                check only the libraries
  -h, --help               Print help
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$ ./compiler build --help
Compile the current package

Usage: compiler build [OPTIONS]

Options:
      --features <FEATURES>  the feature to turn on
  -h, --help                 Print help
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$ ./compiler init --help
Create a new typescipt project in an existing directory

Usage: compiler init [OPTIONS]

Options:
      --lib                      place the configuration in library mode
      --version <VERSION>        The project is bounded to a specific version of the compiler
      --ts-version <TS_VERSION>  The project is bounded to a specific version of Typescript
  -h, --help                     Print help
yc@yc-PORTEGE-X30W-J:~/Documents/GitHub/native-js/native-ts-cli/target/debug/examples$
```

Feature	Justification	met
Command line interface	A command line interface	Yes
Prints the usage format of the command line interface	Tells the user how to use the command line interface	Yes
Lists all the possible commands	Tells the user what commands they can use.	Yes
Display description for each command	Tells the user the function of each command	Yes
Display optional flags for the command line interface	Tells the user what options they have	Yes
List all the optional flags of the command line interface	Tells the user what flags they can add to the command line interface	Yes
Display the short name and long name of each flag	Tells the user how they can add flags to cli	Yes
Display whether a flag requires an argument	Tells the user if they should provide an argument for the flag	Yes
Display description for each flag in command line interface.	Tells the user about the purpose of the flags	Yes
Display helper for command when the help flag is set	Allow the user to find detail description of each command	Yes
Display description of command in command helper	Tells the user the functionality of the command	Yes
Display command usage	Tells the user how the command should be used	Yes
Display optional flags for the commands	Tells the user what flags can be added to their command	Yes
Display short name and long name for flags in command helper	Tells the user how to add flags to command	Yes
Display description for flags in command helper	Tells the user what the flag does to the command	Yes

Interview

index	question	Answers
1.	Do you think the command line interface is easy to use	- Yes - Yes
2.	Do you find the command line helper useful	- Yes - Yes
3.	What changes do you think should be added to the appearance of the command line interface	- None - None
4.	What changes do you think should be added to the functionality of the command line interface	- A test command to allow users to run tests - A command to run build scripts
5.	Is the description clear in the command line helper	- Yes - Yes, but some descriptions can be more detailed.
6.	Do you think the error messages are clear	- Yes - Yes
7.	Do you think the warning messages are clear	- Yes - Yes
8.	What do you think about the appearance of the warning and error messages	- Good, but a function trace can be added to the error to help debugging - Good, but syntax highlights can be added to the referenced source code.

feature	Justification	met	Solution
Display syntax errors to user	This allows user to debug their source code as they are using this compiler.	Yes	/
High-light and display the section of source code where the syntax error happens	This allows the user to easily locate the origin of source code where the syntax error happens. This helps developers to faster develop code.	Partially	Although source codes are referenced and displayed and also underlined, The source codes were not highlighted with colour. To do so, in the future a cross platform terminal user interface library would be used to display the errors so that they can be highlighted with colours.
Display the reason of error to the user when occurred	This allow user to better understand what kind of error has happened so that they can fix their source code.	Yes	/
Perform validation of source code without compiling into machine code	This allows users to debug their code without using much resources to create machine code. This is useful for user when writing source code.	Yes	/
Caching files from the internet	This reduces the use of network resources and speed up the parsing process. Recent files installed from the internet are cached and will not be downloaded again in a short period.	No	Currently, files downloaded from the network are not cached. They are instead downloaded every time a compilation takes place. The address this, a caching mechanism will be introduced to store files locally.

Emitting intermediate representation codes to user when requested	This allow users to better understand how their source code is processed and to debug their source code.	Yes	The HIR can be emitted into a file when user turns on a flag in the command line interface.
Presenting logs to user at each stage of the compilation process	This gives an indication to the user as where the compiler is up to so that user can estimate time required for the compiler to finish.	No	Currently, logs are not presented to the user. This is because this project has not yet finished. In the future, logs will be displayed to the user on each key stage of compilation through a cross platform terminal user interface library.

4.5 maintenance

The maintainability of this project is high. I have separated each stage of the compiler process into different modules. The difference process of each stage is also divided into different files so that the code can be easier referenced. Some modules are reused in the project so that repeat code is reduced.

Nearly all of the code are commented with detailed explanation. The codes are annotated to aid future maintenance of the system.

All variables and structures are appropriately named to make the code more readable and easier to debug. The structures are well defined to suit the problem and function intended. Variable names follows strict rule, only alphabets and underline can be used to define a variable. Global variables have names that are in capital letter so that they can be easily distinguished from local variables.

Unused variables and structures are removed from the source code so that the code is more readable.

The procedures and functions are written in a way such that they are as generic as possible so they can be reused. Interfaces(traits in Rust) are defined and used so that functions can be generic and can be reused with different structure as input.

The code follows a strict formatting rules. It follows the standard rust code formatting method. The cargo formatting utility is used to format the code so that the formatting is consistent across all codes. All indentations are automatically corrected by the formatter.

Document references to the code are automatically generated. It is automatically generated on lib.rs when a module is uploaded to cargo. These documents shows details about structures, traits and functions. They also generates description of them base on the annotated comments.

However, maintenance of the project on the backend side may have some limitations, this issue also applies to all LLVM based compilers. In order to build the compiler, a copy of the LLVM library must be present. On Linux, this can be done by installing it through package manager. However on Windows, the LLVM must be downloaded from the repository. A configuration tool of LLVM is required to link against LLVM when building. This tool however is not included in the pre-compiled LLVM distribution, therefore the whole LLVM project must be built from source code when building on windows. On windows, the version of Rust used to build the Typescript compiler must also be built by the same compiler as the LLVM is built, either MSVC or MinGW. A copy of custom built LLVM must therefore be maintained.

4.6 Limitations and potential improvements

Time limitations

The current capability of the compiler is very limited as its implementation is currently not complete. This is because of the time limitation has on implementation. This project has been developed in 6 months. The amount of time required to implement the solution is higher then our initial estimation.

Incomplete MIR stage

The implementation of MIR stage of the compiler is not yet finished. This is due to the time limitation this project has. More time is needed to finish the HIR stage then estimated. Because that the MIR stage is not fully implemented, the compile currently can only compile source codes into HIR. The compiler is not able to generate machine code because of this. However, HIR codes can still be executed through an interpreter making the code executable.

In the future, when the MIR stage is finished, users will be able to compile source code into machine code and execute them directly.

Missing Generic support

The compiler currently does not support generic types. Placing generic code in the source code will result in an error. This is because the implementation of generic support has been a failure.

In the future, the implementation for generic type support will be revised. This time a more sophisticated approach will be carefully planned and designed before implementing it. I will reference how other compilers process generic types and try to figure out a solution suitable for my compiler. Once this is implemented, users will be able to use generic types in their code.

Dynamic import of modules

Dynamic importing of modules is a feature of the ECMA standard. My compiler however is not able to support this. This is because the compiler is intended to compile source code into static machine codes. Dynamic importing relies on executing source codes during runtime through an interpreter. Since our compiler statically compiles code this is not supported.

There would be no action taken to address this issue. It is a limitation posed by the foundation of how the compiler is intended to be used. An error would be returned to the user if they try to perform dynamic importing in the source code.

No support for Eval

The ‘eval’ function is not supported. It is a feature in the early adoption of ECMA standard. However, this feature is not encouraged and only remained for compatibility of older source codes. Eval takes in source code and executes it in the parent context. The source code is executed in the interpreter. However its execution affects things such as variables out side of the function.

The fact that this compiler statically compiles source codes means that it is not possible to support such functionalities. Statically compiled codes cannot be mutated by source codes interpreted during runtime.

No action would be taken for this issue. It is simply not supported now and in the future.

Garbage collector

The current garbage collector implementation only partially functions. The garbage collector is able to allocate memory, perform generation separation, perform concurrent marking and able to free unused memory. However, it is currently not functioning because it misses pointers on the stack when conservative marking is performed. This means that it may free memory that are still in used causing the programme to malfunction.

I would fix this problem by referencing other garbage collector’s code to look for viable solutions. To do this, I will look into well tested and proven garbage collectors such as the oil pen garbage collector from v8 engine, the Boehm–Demers–Weiser garbage collector. I will reimplement the marking phase of conservative collection in the future.

LLVM compatibility

LLVM is known to have relatively weak API stability guarantees. Its API changes from every version of release. Therefore, updating LLVM version for the project in the future would not be easy. A lot of codes would have to be rewritten when the LLVM version is updated.

There is currently no plausible solution to this problem. We can only simply rewrite the codes that are required every time LLVM is updated.

Concurrent compilation

One feature I would like to add to the compiler in the future is to have concurrent compilation process. Currently, the compiler runs on a single thread. By adding concurrent processing of source code, the compilation speed can increase in order of magnitude when compiling large projects. This will make the compiler more efficient.

To implement this, a worker pool of threads should be created when the compiler starts. Each module of source code when loaded to the compiler is then distributed to the worker threads in the pool. The worker pool can minimalise overhead by performing the compilation task asynchronously.

Incremental compilation

One feature I would like to add to the compiler is incremental compilation. Incremental compilation reuses data generated in the previous iteration of compilation to reduce the amount of work required during compilation.

To implement this, the IR code produced during last iteration is stored in the file system. It is read when the compiler begins the compilation process.

4.8 Conclusion

Goods

- The compiler implementation is on track, no major issue is found in the programme.
- All the algorithm implemented works as expected.
- The Interpreter has proven that the translation process is accurate enough for real world applications such as binary search and other algorithms
- The command line interface is easy to understand.
- The compiler supports majority of the Typescript syntaxes

Bad

- The compiler is not finished
- Some hidden bugs exist, although this is minor.
- The compiler does not support generic types.
- The runtime library does not provide standard library.
- The garbage collector of the runtime does not function well.

References

1. *Rust Foundation (2023) The HIR (High-level IR) - Rust Compiler Development Guide*. Available at: https://rustc-dev-guide.rust-lang.org/hir.html (Accessed: 23 November 2023).

2. *Rust Foundation (2023) The MIR (Mid-level IR) - Rust Compiler Development Guide*. Available at: https://rustc-dev-guide.rust-lang.org/mir.html (Accessed: 23 November 2023).

3. The Go Authors (2023) Introduction to the Go compiler. Available at: https://github.com/golang/go/blob/master/src/cmd/compile/README.md (Accessed: 23 November 2023)

4. Typescript Document - Advance Types. Available at: https://www.typescriptlang.org/docs/handbook/advanced-types.html#using-type-predicates (Accessed: 28 December 2023)

5. MDN Web Docs Glossary: Definitions of Web-related terms: Hoisting. Available at: https://developer.mozilla.org/en-US/docs/Glossary/Hoisting (Accessed: 12 Feb 2024)

6. C++ ABI Summary: Itanium CXX ABI Avaliable at: https://itanium-cxx-abi.github.io/cxx-abi (Accessed 27 Feb 2024)

7. C++ ABI for Itanium: Exception Handling. Available at: http://refspecs.linux-foundation.org/abi-eh-1.21.html#base-personality (Accessed 27 Feb 2024)

8. Linux standard base core specification 4.0: Exception frames Available at: https://refspecs.linuxfoundation.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html (Accessed 27 Feb 2024)

9. Exception Handling in LLVM – LLVM 19.0.0git document Available at: https://llvm.org/docs/ExceptionHandling.html (Accessed 28 Feb 2024)

Part 5
Snapshot of source code

5.1 Source code: native-ts-parser

5.1.1 native-ts-parser/lib.rs

```rust
use std::collections::HashMap;
use std::io::Write;
use std::path::{Path, PathBuf};

use swc_core::common::{BytePos, Spanned};
pub use swc_core;

/// parser or configuration file
pub mod config;

/// unique id of module
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct ModuleId(usize);

/// a parsed module
#[derive(Debug)]
pub struct ParsedModule {
    /// the canonicalised name
    pub path: PathBuf,
    /// the unique id
    pub id: ModuleId,
    /// dependencies
    pub dependencies: Vec<ModuleId>,
    /// the ast of module
    pub module: swc_core::ecma::ast::Module,
}

/// a set of parsed modules
#[derive(Debug)]
pub struct ParsedProgram {
    pub modules: HashMap<ModuleId, ParsedModule>,
}

/// a temperary structure used to parse source code
#[derive(Default)]
pub struct Parser {
    /// source map holds the files and paths
    src: swc_core::common::SourceMap,
    /// the already parsed modules
    modules: HashMap<ModuleId, ParsedModule>,
}
```

```rust
impl Parser {
    pub fn new() -> Self {
        Self {
            src: swc_core::common::SourceMap::new(Default::default()),
            modules: HashMap::new(),
        }
    }
}

impl Parser {
    /// try to resolve dependency
    pub fn resolve_dependency(&self, base_path: &Path, name: &str) ->
Result<PathBuf, String> {
    // it is a local file
    if name.starts_with("./") {
        return Ok(base_path.join(name));
    }
    // it is a web file
    if name.starts_with("http://") || name.starts_with("https://") {
        log::info!("GET from {}", name);

        // try to get the file from server
        match ureq::get(name).call() {
            Ok(response) => {
                match response.into_string() {
                    // a response has been returned
                    Ok(body) => {
                        // creates a random file name
                        let tmp = std::env::temp_dir().join(
                            std::time::SystemTime::now()
                            .duration_since(std::time::UNIX_EPOCH)
                            .unwrap()
                            .as_nanos()
                            .to_string(),
                        );
                        // create the tmp file
                        let mut f = std::fs::File::create(&tmp).expect("faield to open
file");
                        // write the content to file
                        f.write_all(body.as_bytes()).expect("error writing file");

                        // return the tmp file path
                        return Ok(tmp);
                    }
```

```
                    // http error
                    Err(e) => return Err(e.to_string()),
                }
            }
            // connection error
            Err(e) => return Err(e.to_string()),
        };
    }

    // join the path and return
    match base_path.join(name).canonicalize() {
        Ok(path) => Ok(path),
        Err(e) => Err(e.to_string()),
    }
}

/// parses a string file
pub fn parse_str(mut self, name: String, src: String) -> Result<ParsedProgram,
String> {
    // register the string source to source map
    let file = self
        .src
        .new_source_file(swc_core::common::FileName::Custom(name), src);

    // parse the file
    self.parse_file(PathBuf::new(), &file.src, file.start_pos, file.end_pos)?;

    // check if any cyclic dependency occoured
    self.check_cyclic_dependency()?;

    // return the parsed program
    return Ok(ParsedProgram {
        modules: self.modules,
    });
}

// parse a file from the gven path
pub fn parse(mut self, main: PathBuf) -> Result<ParsedProgram, String> {
    // parse file as a module
    self.parse_module(main)?;

    // check if any cyclic dependencies occoured
    self.check_cyclic_dependency()?;
```

```
    // return the parsed program
    return Ok(ParsedProgram {
        modules: self.modules,
    });
}

// parse a file as module with a given path
fn parse_module(&mut self, path: PathBuf) -> Result<ModuleId, String> {
    // path must be canonicalised
    let path = match path.canonicalize() {
        Ok(p) => p,
        Err(e) => return Err(e.to_string()),
    };

    // module is already parsed
    if let Some(m) = self.modules.values().find(|m| m.path == path) {
        return Ok(m.id);
    }

    // load the file
    let file = match self.src.load_file(&path) {
        Ok(file) => file,
        // file read error
        Err(e) => return Err(e.to_string()),
    };

    // parse the file
    return self.parse_file(path, &file.src, file.start_pos, file.end_pos);
}

/// parse the file
fn parse_file(
    &mut self,
    path: PathBuf,
    input: &str,
    start: BytePos,
    end: BytePos,
) -> Result<ModuleId, String> {
    let input = swc_core::common::input::StringInput::new(&input, start, end);

    let mut parser = swc_core::ecma::parser::Parser::new(
swc_core::ecma::parser::Syntax::Typescript(swc_core::ecma::parser::TsConfig::
default()),
        input,
```

```
    None,
);

// parse the module
let re = parser.parse_typescript_module();

let mut module = match re {
    Err(e) => {
        // lookup the position
        let loc = self.src.lookup_char_pos(e.span_lo());
        // format error
        return Err(format!(
            "{}:{}:{}: {}",
            loc.file.name,
            loc.line,
            loc.col_display,
            e.kind().msg()
        ));
    }
    Ok(m) => m,
};

let mut dependencies = Vec::new();

// loop through statements and fin dependencies
for item in &mut module.body {
    if let swc_core::ecma::ast::ModuleItem::ModuleDecl(m) = item {
        match m {
            // import from
            swc_core::ecma::ast::ModuleDecl::Import(i) => {
                let p = self.resolve_dependency(&path, &i.src.value)?;
                i.src.raw = None;
                i.src.value = p.to_string_lossy().into();
                // parse the dependency
                let id = self.parse_module(p)?;
                dependencies.push(id);
            }
            // export from
            swc_core::ecma::ast::ModuleDecl::ExportAll(e) => {
                let p = self.resolve_dependency(&path, &e.src.value)?;
                e.src.raw = None;
                e.src.value = p.to_string_lossy().into();
                // parse the dependency
                let id = self.parse_module(p)?;
```

```
                dependencies.push(id);
            }
            // export from
            swc_core::ecma::ast::ModuleDecl::ExportNamed(n) => {
                if let Some(src) = &mut n.src {
                    let p = self.resolve_dependency(&path, &src.value)?;
                    src.raw = None;
                    src.value = p.to_string_lossy().into();
                    // parse the dependency
                    let id = self.parse_module(p)?;
                    dependencies.push(id);
                }
            }
            _ => {}
        }
    }
}

let id = self.modules.len();

// insert into map
self.modules.insert(
    ModuleId(id),
    ParsedModule {
        path: path,
        id: ModuleId(id),
        dependencies,
        module: module,
    },
);

return Ok(ModuleId(id));
}

// check if a cyclic dependency chain occoured
fn check_cyclic_dependency(&self) -> Result<(), String> {
if self.modules.len() == 0 {
return Ok(());
}
// allocate visited stack
let mut visited = Vec::with_capacity(self.modules.len());
// allocate recurring stack
let mut rec_stack = Vec::with_capacity(self.modules.len());
```

```
// set all value to false
visited.resize(self.modules.len(), false);
rec_stack.resize(self.modules.len(), false);

// loop over every module
for id in self.modules.keys() {
// not visited
if !visited[id.0] && self.is_cyclic_until(*id, &mut visited, &mut rec_stack) {
// cyclic dependency detected
let mut msg = "cyclic dependency detected: ".to_string();

// format the error message
// find any module that is recurring
for (i, recurring) in rec_stack.into_iter().enumerate() {
// is recurring
if recurring {
msg.push_str(
&self
.modules
.get(&ModuleId(i))
.unwrap()
.path
.to_string_lossy(),
);
msg.push_str(" -> ");
}
}

return Err(msg);
}
}

return Ok(());
}

// recurring function to find cycles
fn is_cyclic_until(&self, id: ModuleId, visited: &mut [bool], rec_stack: &mut
[bool]) -> bool {
// not visited
if !visited[id.0] {
// set visited
visited[id.0] = true;
// set recurring
rec_stack[id.0] = true;
```

```
// visit every dependency
for dep in &self.modules.get(&id).unwrap().dependencies {
// dependency not visited
if !visited[dep.0] && self.is_cyclic_until(*dep, visited, rec_stack) {
return true;
} else if rec_stack[dep.0] {
// dependency is recurring
return true;
}
}
}
// set recurring to false
rec_stack[id.0] = false;
return false;
}
}
```

5.1.2 native-ts-parser/config.rs

```
use std::collections::HashMap;

use serde::{Deserialize, Serialize};

#[derive(Debug, Serialize, Deserialize)]
pub struct Config {
pub project: ProjectConfig,
#[serde(default)]
pub lib: LibConfig,
#[serde(default)]
pub bin: BinConfig,
#[serde(default)]
pub dependencies: Dependencies,
#[serde(default)]
pub target: Target,
#[serde(default)]
pub features: Features,
#[serde(default)]
pub profile: ProfileConfig,
}

#[derive(Debug, Serialize, Deserialize)]
pub struct ProjectConfig {
pub name: String,
pub version: String,
```

```
pub authors: Option<Vec<String>>,
#[serde(rename = "compiler-version")]
pub compiler_version: Option<String>,
#[serde(rename = "ts-version")]
pub ts_version: Option<String>,
pub description: Option<String>,
pub documentation: Option<String>,
pub readme: Option<String>,
pub homepage: Option<String>,
pub repository: Option<String>,
pub licence: Option<String>,
#[serde(rename = "licence-file")]
pub licence_file: Option<String>,
pub keywords: Option<Vec<String>>,
pub categories: Option<Vec<String>>,
pub exclude: Option<Vec<String>>,
}

#[derive(Debug, Default, Serialize, Deserialize)]
pub struct LibConfig {
#[serde(default)]
pub test: bool,
pub lib_type: Option<String>,
pub features: Option<Vec<String>>,
}

#[derive(Debug, Default, Serialize, Deserialize)]
pub struct BinConfig {
#[serde(default)]
pub test: bool,
pub features: Option<Vec<String>>,
}

pub type Dependencies = HashMap<String, Dependency>;

#[derive(Debug, Serialize, Deserialize)]
pub struct Dependency {
pub path: Option<String>,
pub url: Option<String>,
pub git: Option<String>,
pub version: Option<String>,
pub features: Option<Vec<String>>,
#[serde(default)]
pub optional: bool,
```

```
}

#[derive(Debug, Default, Serialize, Deserialize)]
#[serde(default)]
pub struct Target {
pub windows: TargetConfig,
pub unix: TargetConfig,
pub linux: TargetConfig,
pub darwin: TargetConfig,
pub macos: TargetConfig,
pub ios: TargetConfig,
pub freebsd: TargetConfig,
pub openbsd: TargetConfig,
pub redox: TargetConfig,
pub android: TargetConfig,
pub x86: TargetConfig,
pub x86_64: TargetConfig,
pub arm: TargetConfig,
pub aarch64: TargetConfig,
pub riscv: TargetConfig,
pub wasm32: TargetConfig,
}

#[derive(Debug, Default, Serialize, Deserialize)]
#[serde(default)]
pub struct TargetConfig {
pub dependencies: Option<Dependencies>,
pub features: Option<Vec<String>>,
}

pub type Features = HashMap<String, Vec<String>>;

#[derive(Default, Debug, Serialize, Deserialize)]
#[serde(default)]
pub struct ProfileConfig {
#[serde(default)]
pub debug: Profile,
#[serde(default)]
pub release: Profile,
}

#[derive(Debug, Default, Serialize, Deserialize)]
pub struct Profile {
#[serde(rename = "opt-level")]
```

```
pub opt_level: Option<u8>,
pub debug: Option<bool>,
pub lto: Option<bool>,
pub incremental: Option<bool>,
}

#[test]
pub fn test() {
let s = r#"
[project]
name = "win"
version = "0.1.1"

[profile.debug]
opt-level = 3
"#;

let config: Config = toml::from_str(s).expect("error");
println!("{}", toml::to_string_pretty(&config).unwrap());
}
```

5.2 Source code: native-ts-hir

5.2.1 native-ts-hir/lib.rs

```
/// HIR definitions
pub mod ast;
/// post transformation checks
mod checks;
/// utils
mod common;
/// symbol table data structure
mod symbol_table;
/// transforms AST to HIR
pub mod transform;

use std::sync::atomic::{AtomicUsize, Ordering};

/// a unique identifier for variables
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct VarId(usize);
```

```
impl VarId {
/// creates a new unique identifier
pub fn new() -> Self {
// static counter
static IDS: AtomicUsize = AtomicUsize::new(0);
// fetch and increment counter
return Self(IDS.fetch_add(1, Ordering::SeqCst));
}
}

/// variable kind, not included in `ast` because it is not part of ast.
///
/// this is only used during the translation process for syntax checks
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub enum VarKind {
/// `var` declare, can be redeclared but must be of same type
Var,
/// `let` declare, cannot be redeclared
Let,
/// `const` declare, readonly
Const,
/// `using` declare, readonly and owned by the scope.
/// destructor called when it goes out of scope
Using,
/// `await using` declare. Same as `using` but with async destructor
AwaitUsing,
}

/// implemented for convenience
impl From<native_ts_parser::swc_core::ecma::ast::VarDeclKind> for VarKind {
fn from(value: native_ts_parser::swc_core::ecma::ast::VarDeclKind) -> Self {
match value {
native_ts_parser::swc_core::ecma::ast::VarDeclKind::Const => Self::Const,
native_ts_parser::swc_core::ecma::ast::VarDeclKind::Let => Self::Let,
native_ts_parser::swc_core::ecma::ast::VarDeclKind::Var => Self::Var,
}
}
}

/// property name for attributes and methods
#[derive(Debug, Clone, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub enum PropName {
/// e.g. obj.prop
Ident(String),
```

```
/// e.g. obj.#prop
Private(String),
/// e.g. obj["prop"]
String(String),
/// e.g. obj[0]
Int(i32),
/// e.g. obj[Symbol.iterator]
Symbol(Symbol),
}

/// format propname
impl core::fmt::Display for PropName {
fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
match self {
Self::Ident(id) => f.write_str(&id),
Self::String(s) => {
f.write_str("\"")?;
f.write_str(s)?;
f.write_str("\"")
}
Self::Int(i) => {
let mut buf = native_js_common::itoa::Buffer::new();
f.write_str(buf.format(*i))
}
Self::Private(p) => {
f.write_str("#")?;
f.write_str(p)
}
Self::Symbol(s) => s.fmt(f),
}
}
}
```

```
/// Typescript builtin symbols
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub enum Symbol {
Iterator,
AsyncIterator,
```

```
Dispose,
AsyncDispose,
HasInstance,
IsConcatSpreadable,
Match,
MatchAll,
Replace,
Search,
Species,
Split,
ToPrimitive,
ToStringTag,
Unscopables,
}

impl core::fmt::Display for Symbol {
fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
f.write_str("Symbol.")?;

f.write_str(match self {
Self::AsyncDispose => "asyncDispose",
Self::AsyncIterator => "asyncIterator",
Self::Dispose => "dispose",
Self::HasInstance => "hasInstance",
Self::IsConcatSpreadable => "isConcatSpreadable",
Self::Iterator => "iterator",
Self::Match => "match",
Self::MatchAll => "matchAll",
Self::Replace => "replace",
Self::Search => "search",
Self::Species => "species",
Self::Split => "split",
Self::ToPrimitive => "toPrimitive",
Self::ToStringTag => "toStringTag",
Self::Unscopables => "unscopables",
})
}
}
```

5.2.2 native-ts-hir/symbol_table.rs

```
use std::collections::HashMap;

use crate::{
ast::{ClassType, EnumType, FuncType, Function, InterfaceType},
```

```
common::{ClassId, EnumId, FunctionId, InterfaceId},
};

/// symbol table stores all the descriptor of a module
pub struct SymbolTable {
/// external functions
pub external_functions: HashMap<String, FuncType>,
/// functions
pub functions: HashMap<FunctionId, Function>,
/// classes
pub classes: HashMap<ClassId, ClassType>,
/// interfaces
pub interfaces: HashMap<InterfaceId, InterfaceType>,
/// enums
pub enums: HashMap<EnumId, EnumType>,
}
```

5.2.3 native-ts-hir/common.rs

```
use core::sync::atomic::{AtomicUsize, Ordering};

/// function id
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct FunctionId(pub(super) usize);

impl FunctionId {
pub fn new() -> Self {
static COUNT: AtomicUsize = AtomicUsize::new(0);
Self(COUNT.fetch_add(1, Ordering::SeqCst))
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct ClassId(pub(super) usize);

impl ClassId {
pub fn new() -> Self {
static COUNT: AtomicUsize = AtomicUsize::new(0);
Self(COUNT.fetch_add(1, Ordering::SeqCst))
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct InterfaceId(pub(super) usize);
```

```
impl InterfaceId {
pub fn new() -> Self {
static COUNT: AtomicUsize = AtomicUsize::new(0);
Self(COUNT.fetch_add(1, Ordering::SeqCst))
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct VariableId(pub(super) usize);

impl VariableId {
pub fn new() -> Self {
static COUNT: AtomicUsize = AtomicUsize::new(0);
Self(COUNT.fetch_add(1, Ordering::SeqCst))
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct GenericId(pub(super) usize);

impl GenericId {
pub fn new() -> Self {
static COUNT: AtomicUsize = AtomicUsize::new(0);
Self(COUNT.fetch_add(1, Ordering::SeqCst))
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct AliasId(pub(super) usize);

impl AliasId {
pub fn new() -> Self {
static COUNT: AtomicUsize = AtomicUsize::new(0);
Self(COUNT.fetch_add(1, Ordering::SeqCst))
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct EnumId(pub(super) usize);

impl EnumId {
pub fn new() -> Self {
static COUNT: AtomicUsize = AtomicUsize::new(0);
Self(COUNT.fetch_add(1, Ordering::SeqCst))
```

```
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct ModuleId(pub(super) usize);

impl ModuleId {
pub fn new() -> Self {
static COUNT: AtomicUsize = AtomicUsize::new(0);
Self(COUNT.fetch_add(1, Ordering::SeqCst))
}
}
```

5.2.4 native-ts-hir/ast/mod.rs

```
pub mod expr;
pub mod format;
pub mod function;
pub mod stmts;
pub mod strict_typed;
pub mod types;
pub mod visit;

use std::collections::HashMap;

pub use expr::*;
pub use function::*;
pub use stmts::*;
pub use types::*;

use crate::{
common::{AliasId, ClassId, EnumId, FunctionId, InterfaceId, ModuleId,
VariableId},
symbol_table::SymbolTable,
PropName,
};

#[derive(Debug, Clone, PartialEq)]
pub enum ModuleExport {
Undefined,
/// a variable
Var(VariableId, Type),
/// a function
Function(FunctionId),
```

```
/// a class type
Class(ClassId),
/// an interface
Interface(InterfaceId),
/// an enum
Enum(EnumId),
/// a type alias
Alias(AliasId),
/// a namespace
NameSpace(ModuleId),
}

pub struct Module {
/// symbol table
pub table: SymbolTable,
/// the unique function id of the entry function
pub main_function: FunctionId,
/// default export
pub default_export: ModuleExport,
/// named exports
pub exports: HashMap<PropName, ModuleExport>,
}
```

5.2.5 native-ts-hir/ast/expr.rs

```
use native_ts_parser::swc_core::common::Span;

use crate::common::{ClassId, FunctionId, VariableId};
use crate::{PropName, Symbol};

use super::Type;

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub enum AssignOp {
/// `=`
Assign,
/// `+=`
AddAssign,
/// `-=`
SubAssign,
/// `*=`
MulAssign,
/// `/=`
DivAssign,
/// `%=`
```

```
ModAssign,
/// `<<=`
LShiftAssign,
/// `>>=`
RShiftAssign,
/// `>>>=`
ZeroFillRShiftAssign,
/// `|=`
BitOrAssign,
/// `^=`
BitXorAssign,
/// `&=`
BitAndAssign,

/// `**=`
ExpAssign,

/// `&&=`
AndAssign,

/// `||=`
OrAssign,

/// `??=`
NullishAssign,
}

impl AssignOp {
pub fn as_str(self) -> &'static str {
match self {
Self::AddAssign => "+=",
Self::AndAssign => "&&=",
Self::BitAndAssign => "&=",
Self::BitOrAssign => "|=",
Self::BitXorAssign => "^=",
Self::DivAssign => "/=",
Self::ExpAssign => "**=",
Self::LShiftAssign => "<<=",
Self::ModAssign => "%=",
Self::MulAssign => "*=",
Self::NullishAssign => "??=",
Self::RShiftAssign => ">>=",
Self::SubAssign => "-=",
Self::ZeroFillRShiftAssign => ">>>=",
```

```
Self::OrAssign => "||=",
Self::Assign => "=",
}
}
}

impl From<native_ts_parser::swc_core::ecma::ast::AssignOp> for AssignOp {
fn from(value: native_ts_parser::swc_core::ecma::ast::AssignOp) -> Self {
match value {
native_ts_parser::swc_core::ecma::ast::AssignOp::AddAssign =>
Self::AddAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::AndAssign =>
Self::AndAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::BitAndAssign =>
Self::BitAndAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::BitOrAssign =>
Self::BitOrAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::BitXorAssign =>
Self::BitXorAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::DivAssign => Self::DivAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::ExpAssign =>
Self::ExpAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::LShiftAssign =>
Self::LShiftAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::ModAssign =>
Self::ModAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::MulAssign =>
Self::MulAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::NullishAssign =>
Self::NullishAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::RShiftAssign =>
Self::RShiftAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::SubAssign =>
Self::SubAssign,
native_ts_parser::swc_core::ecma::ast::AssignOp::ZeroFillRShiftAssign => {
Self::ZeroFillRShiftAssign
}
native_ts_parser::swc_core::ecma::ast::AssignOp::Assign => Self::Assign,
native_ts_parser::swc_core::ecma::ast::AssignOp::OrAssign => Self::OrAssign,
}
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
```

```
pub enum BinOp {
Add,
Sub,
Mul,
Div,
Mod,
Exp,
EqEq,
EqEqEq,
NotEq,
NotEqEq,
Lt,
Lteq,
Gt,
Gteq,
RShift,
URShift,
LShift,
And,
Or,
BitOr,
BitXor,
BitAnd,
Nullish,
In,
}

impl BinOp {
pub fn as_str(self) -> &'static str {
match self {
Self::Add => "+",
Self::And => "&&",
Self::BitAnd => "&",
Self::BitOr => "|",
Self::BitXor => "^",
Self::Div => "/",
Self::EqEq => "==",
Self::EqEqEq => "===",
Self::Exp => "**",
Self::Gt => ">",
Self::Gteq => ">=",
Self::In => "in",
Self::LShift => "<<",
Self::Lt => "<",
```

```
Self::Lteq => "<=",
Self::Mod => "%",
Self::Mul => "*",
Self::NotEq => "!=",
Self::NotEqEq => "!==",
Self::Nullish => "??",
Self::RShift => ">>",
Self::Sub => "-",
Self::URShift => ">>>",
Self::Or => "||",
}
}
}

impl From<native_ts_parser::swc_core::ecma::ast::BinaryOp> for BinOp {
fn from(value: native_ts_parser::swc_core::ecma::ast::BinaryOp) -> Self {
match value {
native_ts_parser::swc_core::ecma::ast::BinaryOp::Add => Self::Add,
native_ts_parser::swc_core::ecma::ast::BinaryOp::BitAnd => Self::BitAnd,
native_ts_parser::swc_core::ecma::ast::BinaryOp::BitOr => Self::BitOr,
native_ts_parser::swc_core::ecma::ast::BinaryOp::BitXor => Self::BitXor,
native_ts_parser::swc_core::ecma::ast::BinaryOp::Div => Self::Div,
native_ts_parser::swc_core::ecma::ast::BinaryOp::EqEq => Self::EqEq,
native_ts_parser::swc_core::ecma::ast::BinaryOp::EqEqEq => Self::EqEqEq,
native_ts_parser::swc_core::ecma::ast::BinaryOp::Exp => Self::Exp,
native_ts_parser::swc_core::ecma::ast::BinaryOp::Gt => Self::Gt,
native_ts_parser::swc_core::ecma::ast::BinaryOp::GtEq => Self::Gteq,
native_ts_parser::swc_core::ecma::ast::BinaryOp::In => Self::In,
native_ts_parser::swc_core::ecma::ast::BinaryOp::InstanceOf => unreachable!
(),
native_ts_parser::swc_core::ecma::ast::BinaryOp::LShift => Self::LShift,
native_ts_parser::swc_core::ecma::ast::BinaryOp::LogicalAnd => Self::And,
native_ts_parser::swc_core::ecma::ast::BinaryOp::LogicalOr => Self::Or,
native_ts_parser::swc_core::ecma::ast::BinaryOp::Lt => Self::Lt,
native_ts_parser::swc_core::ecma::ast::BinaryOp::LtEq => Self::Lteq,
native_ts_parser::swc_core::ecma::ast::BinaryOp::Mod => Self::Mod,
native_ts_parser::swc_core::ecma::ast::BinaryOp::Mul => Self::Mul,
native_ts_parser::swc_core::ecma::ast::BinaryOp::NotEq => Self::NotEq,
native_ts_parser::swc_core::ecma::ast::BinaryOp::NotEqEq => Self::NotEqEq,
native_ts_parser::swc_core::ecma::ast::BinaryOp::NullishCoalescing =>
Self::Nullish,
native_ts_parser::swc_core::ecma::ast::BinaryOp::RShift => Self::RShift,
native_ts_parser::swc_core::ecma::ast::BinaryOp::Sub => Self::Sub,
```

```
native_ts_parser::swc_core::ecma::ast::BinaryOp::ZeroFillRShift =>
Self::URShift,
}
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub enum UpdateOp {
/// ++expr
PrefixAdd,
/// --expr
PrefixSub,
/// expr++
SuffixAdd,
/// expr--
SuffixSub,
}

#[derive(Debug, Clone)]
pub enum Callee {
Function(FunctionId),
Member { object: Expr, prop: PropNameOrExpr },
Expr(Expr),
Super(ClassId),
}

impl Callee {
pub fn is_member(&self) -> bool {
match self {
Self::Member { .. } => true,
_ => false,
}
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub enum UnaryOp {
LogicalNot,
BitNot,
Typeof,
Void,
Minus,
Plus,
}
```

```
impl UnaryOp {
pub fn as_str(self) -> &'static str {
match self {
Self::BitNot => "~",
Self::LogicalNot => "!",
Self::Minus => "-",
Self::Plus => "+",
Self::Typeof => "typeof",
Self::Void => "void",
}
}
}

#[derive(Debug, Clone)]
pub enum PropNameOrExpr {
PropName(PropName),
Expr(Box<Expr>, Type),
}

#[derive(Debug, Clone)]
pub enum Expr {
Undefined,
Null,
Bool(bool),
Int(i32),
Number(f64),
/// loads an i128
Bigint(i128),
/// loads a string
String(String),
/// loads a symbol
Symbol(Symbol),
Regex(),
/// function is static and is initialised
Function(FunctionId),
/// a closure captures variables
Closure(FunctionId),
/// read the this binding
This,
/// constructs an array
Array {
values: Vec<Expr>,
},
```

```
/// constructs a tuple
Tuple {
values: Vec<Expr>,
},
Object {
props: Vec<(PropName, Expr)>,
},
/// constructs a class
New {
class: ClassId,
args: Vec<Expr>,
},
/// calls a function
Call {
callee: Box<Callee>,
args: Vec<Expr>,
optional: bool,
},
/// returns a reference
Member {
object: Box<Expr>,
key: PropNameOrExpr,
optional: bool,
},
/// returns the value with member type
MemberAssign {
op: AssignOp,
object: Box<Expr>,
key: PropNameOrExpr,
value: Box<Expr>,
},
/// increments or decrements the property
MemberUpdate {
op: UpdateOp,
object: Box<Expr>,
key: PropNameOrExpr,
},
/// push value to stack
Push(Box<Expr>),
/// read value from top of stack
ReadStack,
/// pop value from stack
Pop,
/// returns the value with variable type
```

```
VarAssign {
op: AssignOp,
variable: VariableId,
value: Box<Expr>,
},
/// returns the loaded value
VarLoad {
span: Span,
variable: VariableId,
},
/// returns the value with
VarUpdate {
op: UpdateOp,
variable: VariableId,
},
/// binary operations
Bin {
op: BinOp,
left: Box<Expr>,
right: Box<Expr>,
},
/// unary operations
Unary {
op: UnaryOp,
value: Box<Expr>,
},
/// selects right if test is nullish else left
Ternary {
test: Box<Expr>,
left: Box<Expr>,
right: Box<Expr>,
},
/// performs expression and returns last value
Seq(Box<Expr>, Box<Expr>),
/// async wait
Await(Box<Expr>),
/// yields from generator
Yield(Box<Expr>),

/// cast a value to another type.
/// type of value must be compatable with Type
Cast(Box<Expr>, Type),
/// assertion that value is not null.
/// this may panic at runtime if value is null or undefined
```

```
AssertNonNull(Box<Expr>),
}
```

5.2.6 native-ts-hir/ast/function.rs

```
use std::collections::HashMap;

use crate::common::VariableId;

use super::{FuncType, GenericParam};
use super::{Stmt, Type};

pub struct VariableDesc {
pub ty: Type,
pub is_heap: bool,
pub is_captured: bool,
}

pub struct FunctionParam {
pub id: VariableId,
pub ty: Type,
}

pub struct Function {
pub this_ty: Type,
pub params: Vec<FunctionParam>,
pub return_ty: Type,

pub variables: HashMap<VariableId, VariableDesc>,
pub captures: Vec<(VariableId, Type)>,
pub stmts: Vec<Stmt>,
}

impl Function {
pub fn ty(&self) -> FuncType {
FuncType {
this_ty: self.this_ty.clone(),
params: self.params.iter().map(|p| p.ty.clone()).collect(),
var_arg: false,
return_ty: self.return_ty.clone(),
}
}
}

pub struct GenericFunction {
```

```
pub type_params: Vec<GenericParam>,

pub this_ty: Type,
pub params: Vec<FunctionParam>,
pub return_ty: Type,

pub variables: HashMap<VariableId, VariableDesc>,
pub stmts: Vec<Stmt>,
}
```

5.2.6 native-ts-hir/ast/stmt.rs

```
use crate::common::{ClassId, FunctionId, InterfaceId, VariableId};

use super::{Expr, Type};

pub enum Stmt {
/// declares a class type
DeclareClass(ClassId),
/// declares an interface type
DeclareInterface(InterfaceId),
/// declares a function
DeclareFunction(FunctionId),
/// declares a generic class
DeclareGenericClass(ClassId),
/// declares a generic interface
DeclareGenericInterface(InterfaceId),
/// declares a generic function
DeclareGenericFunction(FunctionId),
/// declares a variable
DeclareVar(VariableId, Type),
/// indicate variable is out of scope
DropVar(VariableId),
/// start of a block
Block { label: String },
/// end of a block
EndBlock,
/// jump if condition
If { test: Box<Expr> },
/// end if
EndIf,
/// else, must be after end if
Else,
/// end else
EndElse,
```

```
/// match a value
Switch(Box<Expr>),
/// a switch case
SwitchCase(Box<Expr>),
/// end of switch case
EndSwitchCase,
/// default case
DefaultCase,
/// end of default case
EndDefaultCase,
/// end of a switch
EndSwitch,
/// a loop
Loop { label: Option<String> },
/// end of loop
EndLoop,

Try,
EndTry,
Catch(VariableId, Box<Type>),
EndCatch,
Finally,
EndFinally,

/// break from a loop
Break(Option<String>),
///
Continue(Option<String>),

Return(Box<Expr>),
Throw(Box<Expr>),
Expr(Box<Expr>),
}
```

5.2.7 native-ts-hir/ast/types.rs

```
use std::collections::HashMap;

use crate::common::{AliasId, ClassId, EnumId, FunctionId, GenericId,
InterfaceId, VariableId};
use crate::{PropName, Symbol};

use super::Expr;

#[repr(C)]
```

```rust
#[derive(Debug, Clone, PartialEq, PartialOrd)]
pub enum Type {
/// any type, alias of a raw interface
Any,
/// not number, string, boolean, bigint, symbol, null, or undefined.
AnyObject,
/// undefined
Undefined,
/// null type
Null,
/// boolean
Bool,
LiteralBool(bool),
/// number, f64
Number,
LiteralNumber(f64),
/// interger, i32
Int,
LiteralInt(i32),
/// big integer, i128
Bigint,
LiteralBigint(i128),
/// string
String,
LiteralString(Box<str>),
/// symbol, represented as u64
Symbol,
/// regular expression object
Regex,
/// any object type
Object(ClassId),
LiteralObject(Box<[(PropName, Type)]>),
/// interface type
Interface(InterfaceId),
/// function type
Function(Box<FuncType>),
/// enum type
Enum(EnumId),
/// array type
Array(Box<Type>),
/// map type
Map(Box<Type>, Box<Type>),
/// union type
Union(Box<[Type]>),
```

```
/// tuple type
Tuple(Box<[Type]>),
/// a promise, returned by an async function
Promise(Box<Type>),
/// an iterator, alias of an interface
Iterator(Box<Type>),

/// an alias type, should not be present after normalisation
Alias(AliasId),
/// a generic type, a placeholder to be resolved
Generic(GenericId),
}

impl Eq for Type {}

impl Ord for Type {
fn cmp(&self, other: &Self) -> std::cmp::Ordering {
match self {
Self::LiteralNumber(n) => match other {
Self::LiteralNumber(i) => return n.total_cmp(i),
_ => {}
},
_ => {}
}
return self.partial_cmp(other).unwrap();
}
}

impl Type {
/// constructs a union with another type
pub fn union(self, other: Type) -> Type {
// same type, not a union
if self == other {
return self;
}
// any type does not require union
if self == Type::Any || other == Type::Any {
return Type::Any;
}

// number and integer can be converted into floating point
if (self == Type::Number || other == Type::Number)
&& (self == Type::Int || other == Type::Int)
{
```

```
return Type::Number;
}

// unions must not contain integer
if other == Type::Int {
return self.union(Type::Number);
}

// unions must not contain integer
if self == Type::Int {
return Type::Number.union(other);
}

// any object can contain any object
if self == Type::AnyObject && other.is_object() {
return Type::AnyObject;
}

// any object can contain any object
if other == Type::AnyObject && self.is_object() {
return Type::AnyObject;
}

// recuring, append union
if other.is_union() && !self.is_union() {
return other.union(self);
}

// match each case
match &self {
// integers are already solved
Type::Int => unreachable!(),
// any is solved
Type::Any => unreachable!(),
// already solved
Type::AnyObject => unreachable!(),
Type::Bigint
| Type::LiteralBigint(_)
| Type::Enum(_)
| Type::Function(_)
| Type::Array(_)
| Type::Bool
| Type::LiteralBool(_)
| Type::LiteralObject(_)
```

```
| Type::Interface(_)
| Type::Null
| Type::Number
| Type::LiteralNumber(_)
| Type::LiteralInt(_)
| Type::Object(_)
| Type::Promise(_)
| Type::Regex
| Type::String
| Type::LiteralString(_)
| Type::Symbol
| Type::Map(_, _)
| Type::Tuple(_)
| Type::Alias(_)
| Type::Generic(_)
| Type::Undefined
| Type::Iterator(_) => {
// simply return a union
Type::Union(Box::new([self, other]))
}
// self is already a union
Type::Union(u) => {
// union already contains type
if u.contains(&other) {
// retunrn the union unchanged
return self;
}

// allocate vec
let mut v = Vec::with_capacity(u.len() + 1);

// clone each element
for ty in u.iter() {
v.push(ty.clone());

// if element contains any, simply return any
if ty == &Type::Any {
return Type::Any;
}
}

// the other type isalso a union
if let Type::Union(u) = other {
// push element in other
```

```
for ty in u.iter() {
// only pushes if not already contained
if !v.contains(ty) {
// push element
v.push(ty.clone());

// if element is any, simply return any
if ty == &Type::Any {
return Type::Any;
}
}
}
} else {
// other is not union, push
v.push(other);
}

// sort the vec for convinience
v.sort();

// box the vec
return Type::Union(v.into_boxed_slice());
}
}
}

/// returns true if self is an object type
pub fn is_object(&self) -> bool {
match self {
Type::AnyObject
| Type::LiteralObject(_)
| Type::Array(_)
| Type::Function(_)
| Type::Map(_, _)
| Type::Object(_)
| Type::Promise(_)
| Type::Regex
| Type::Tuple(_)
| Type::Iterator(_) => true,
Type::Union(u) => u.iter().all(Self::is_object),
_ => false,
}
}
```

```rust
/// returns true if self is union
pub fn is_union(&self) -> bool {
match self {
Self::Union(_) => true,
_ => false,
}
}

/// returns true if self is an interface
pub fn is_interface(&self) -> bool {
match self {
Self::Any | Self::AnyObject | Self::Interface(_) | Self::Iterator(_) => true,
_ => false,
}
}

pub fn is_array(&self) -> bool {
match self {
Self::Array(_) => true,
_ => false,
}
}
}

/// a function type
#[derive(Debug, Clone, PartialEq, PartialOrd)]
pub struct FuncType {
/// the `this` param
pub this_ty: Type,
/// function params
pub params: Vec<Type>,
/// is function variable argument
pub var_arg: bool,
/// return type of function
pub return_ty: Type,
}

/// a generic function param
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct GenericParam {
pub id: GenericId,
pub name: String,
pub constrain: Option<InterfaceId>,
pub extends: Option<ClassId>,
```

```rust
}

/// an object property descriptor
#[derive(Debug, Clone)]
pub struct PropertyDesc {
/// type of property
pub ty: Type,
/// is property readonly
pub readonly: bool,
/// initialiser of property
pub initialiser: Option<Expr>,
}

/// a class definition
#[derive(Debug, Default, Clone)]
pub struct ClassType {
pub name: String,

/// parent class extends from
pub extends: Option<ClassId>,
/// interfaces implemented
pub implements: Vec<InterfaceId>,

/// class may not have constructor
pub constructor: Option<(FunctionId, FuncType)>,

/// static properties are just global variables
pub static_properties: HashMap<PropName, (VariableId, Type)>,
/// static methods are just static functions
pub static_methods: HashMap<PropName, (FunctionId, FuncType)>,
/// static generic methods are just generic functions
pub static_generic_methods: HashMap<PropName, (FunctionId,)>,

/// attributes of class
pub properties: HashMap<PropName, PropertyDesc>,
/// methods of class
pub methods: HashMap<PropName, (FunctionId, FuncType)>,
/// TODO: generic methods
pub generic_methods: HashMap<PropName, (FunctionId,)>,
}

/// descriptor of an interface property
#[derive(Debug, Clone)]
pub struct InterfacePropertyDesc {
```

```rust
/// type
pub ty: Type,
/// is read only
pub readonly: bool,
/// is property optional
pub optional: bool,
}

/// descriptor of an interface method
#[derive(Debug, Clone)]
pub struct InterfaceMethod {
/// is method readonly
pub readonly: bool,
/// is method optional
pub optional: bool,
/// params of method
pub params: Vec<Type>,
/// return type of method
pub return_ty: Type,
}

/// an interface definition
#[derive(Debug, Default)]
pub struct InterfaceType {
/// name of the interface, only for debugging purpose
pub name: String,

/// classes that extend interface
pub extends: Vec<ClassId>,
/// interfaces implemented by interface
pub implements: Vec<InterfaceId>,

/// properies of this interface
pub properties: HashMap<PropName, InterfacePropertyDesc>,
/// methods of this interface
pub methods: HashMap<PropName, InterfaceMethod>,
}

/// descriptor of an enum variant
#[derive(Debug, Clone)]
pub struct EnumVariantDesc {
/// name of the variant
pub name: PropName,
}
```

```rust
/// a enum type definition
#[derive(Debug, Clone)]
pub struct EnumType {
/// name of the enum, for debugging purpose only
pub name: String,
/// variants of the enum
pub variants: Vec<EnumVariantDesc>,
}

/// literal types
#[derive(Debug, Clone, PartialOrd)]
pub enum LiteralType {
/// string literal
String(Box<str>),
/// number literal
Number(f64),
/// integer literal
Int(i32),
/// symbol literal
Symbol(Symbol),
/// boolean literal
Bool(bool),
/// bigint literal
Bigint(i128),
}

// manual implementation of equals
impl PartialEq for LiteralType {
fn eq(&self, other: &Self) -> bool {
match self {
// use total compare for f64
Self::Number(n) => match other {
Self::Number(i) => n.total_cmp(i).is_eq(),
_ => false,
},
Self::Int(i) => {
if let LiteralType::Int(n) = other {
return i == n;
}
other.eq(&LiteralType::Number(*i as f64))
}
Self::String(s) => match other {
Self::String(n) => s == n,
```

```
_ => false,
},
Self::Bigint(i) => match other {
Self::Bigint(n) => i == n,
_ => false,
},
Self::Bool(b) => match other {
Self::Bool(p) => b == p,
_ => false,
},
Self::Symbol(s) => match other {
Self::Symbol(n) => s == n,
_ => false,
},
}
}
}

// total eq should work on both order
impl Eq for LiteralType {}

// manual implementation of order
impl Ord for LiteralType {
fn cmp(&self, other: &Self) -> std::cmp::Ordering {
match self {
Self::Number(n) => match other {
Self::Number(i) => return n.total_cmp(i),
_ => {}
},
_ => {}
};

return self.partial_cmp(other).expect("partial compare");
}
}
```

5.2.8 native-ts-hir/ast/format.rs

```
use crate::{
common::{FunctionId, VariableId},
PropName,
};

use super::{Callee, Expr, Module, PropNameOrExpr, Stmt, Type, UnaryOp,
UpdateOp};
```

```
use crate::symbol_table::SymbolTable;

pub struct Formatter<'a> {
spaces: usize,
buf: String,
table: &'a SymbolTable,
}

impl<'a> Formatter<'a> {
pub const fn new(table: &'a SymbolTable) -> Self {
Self {
spaces: 0,
buf: String::new(),
table,
}
}
pub fn emit_string(&mut self) -> String {
core::mem::replace(&mut self.buf, String::new())
}
fn emit_spaces(&mut self) {
for _ in 0..self.spaces {
self.buf.push(' ')
}
}
fn new_scope(&mut self) {
self.spaces += 4;
}
fn close_scope(&mut self) {
self.spaces -= 4;
}
fn write_str(&mut self, s: &str) {
self.buf.push_str(s);
}
fn write_int<I: itoa::Integer>(&mut self, i: I) {
let mut buf = itoa::Buffer::new();
self.write_str(buf.format(i))
}
pub fn format_module(&mut self, m: &Module) {
let func = self
.table
.functions
.get(&m.main_function)
.expect("invalid function");
for stmt in &func.stmts {
```

```
self.format_stmt(stmt);
}
}
pub fn format_stmt(&mut self, stmt: &Stmt) {
match stmt {
Stmt::Block { label } => {
self.emit_spaces();
self.write_str(&label);
self.write_str(":{\n");

self.new_scope()
}
Stmt::EndBlock => {
self.close_scope();
self.emit_spaces();
self.write_str("}\n");
}
Stmt::DeclareClass(id) => {
let class = self.table.classes.get(id).expect("invalid class");

self.write_str("class class");
self.write_int(id.0);

if let Some(sup) = class.extends {
self.write_str(" extends class");
self.write_int(sup.0);
};

if !class.implements.is_empty() {
self.write_str(" implements ");
for (i, iface) in class.implements.iter().enumerate() {
self.write_str("iface");
self.write_int(iface.0);

if i + 1 != class.implements.len() {
self.write_str(", ")
}
}
}

self.write_str("{\n");
self.new_scope();

if let Some((id, _)) = &class.constructor {
```

```
self.emit_spaces();
self.write_str("constructor");
self.format_function_body(*id);
}

for (name, (_, ty)) in &class.static_properties {
self.emit_spaces();
self.write_str("static ");
self.format_propname(name);
self.write_str(":");
self.format_ty(ty);
self.write_str(";\n")
}

for (name, prop) in &class.properties {
self.emit_spaces();
if prop.readonly {
self.write_str("readonly ");
}
self.format_propname(name);
self.write_str(":");
self.format_ty(&prop.ty);

if let Some(init) = &prop.initialiser {
self.write_str("=");
self.format_expr(init);
self.write_str(";\n");
}
}

for (name, (id, _)) in &class.static_methods {
self.emit_spaces();
self.write_str("static function ");
self.format_propname(name);
self.format_function_body(*id);
}

for (name, (id, _)) in &class.methods {
self.emit_spaces();
self.write_str("function ");
self.format_propname(name);
self.format_function_body(*id);
}
```

```
self.close_scope();
self.emit_spaces();
self.write_str("}\n");
}
Stmt::DeclareFunction(id) => {
self.emit_spaces();
self.write_str("function fun");
self.write_int(id.0);
self.format_function_body(*id);
}
Stmt::DeclareGenericClass(_id) => {}
Stmt::DeclareGenericFunction(_id) => {}
Stmt::DeclareGenericInterface(_id) => {}
Stmt::DeclareInterface(id) => {
let iface = self.table.interfaces.get(id).expect("invalid interface");

self.emit_spaces();
self.write_str("interface iface");
self.write_int(id.0);

if iface.extends.len() > 0 {
self.write_str(" extends");

let mut iter = iface.extends.iter();
self.write_str(" class");
self.write_int(iter.next().unwrap().0);

for c in iter {
self.write_str(", class");
self.write_int(c.0);
}
}

if iface.implements.len() > 0 {
self.write_str(" implements");

let mut iter = iface.implements.iter();
self.write_str(" iface");
self.write_int(iter.next().unwrap().0);

for c in iter {
self.write_str(", iface");
self.write_int(c.0);
}
```

```
}

self.write_str("{\n");
self.new_scope();

for (propname, prop) in &iface.properties {
self.emit_spaces();
self.format_propname(propname);
if prop.optional {
self.write_str("?")
}
self.write_str(": ");
self.format_ty(&prop.ty);
self.write_str(";\n");
}

for (name, method) in &iface.methods {
self.emit_spaces();
self.format_propname(name);
self.write_str("(");

for param in &method.params {
self.format_ty(param);
self.write_str(", ");
}
self.write_str("):");
self.format_ty(&method.return_ty);
self.write_str(";\n")
}

self.close_scope();
self.write_str("}\n");
}
Stmt::DeclareVar(id, ty) => {
self.emit_spaces();
self.write_str("var var");
self.write_int(id.0);
self.write_str(":");
self.format_ty(ty);
self.write_str("\n");
}
Stmt::DropVar(_) => {}
Stmt::If { test } => {
self.emit_spaces();
```

```
self.write_str("if (");
self.format_expr(test);
self.write_str("){\n");

self.new_scope();
}
Stmt::EndIf => {
self.close_scope();
self.emit_spaces();
self.write_str("}\n")
}
Stmt::Else => {
self.emit_spaces();
self.write_str("else {\n");
self.new_scope();
}
Stmt::EndElse => {
self.close_scope();
self.emit_spaces();
self.write_str("}\n")
}
Stmt::Switch(test) => {
self.emit_spaces();
self.write_str("switch (");
self.format_expr(test);
self.write_str("){\n");

self.new_scope();
}
Stmt::EndSwitch => {
self.close_scope();
self.emit_spaces();
self.write_str("}\n");
}
Stmt::SwitchCase(test) => {
self.emit_spaces();
self.write_str("case ");
self.format_expr(test);
self.write_str(":\n");
self.new_scope();
}
Stmt::EndSwitchCase => {
self.emit_spaces();
self.write_str("break;\n");
```

```
self.close_scope();
}
Stmt::DefaultCase => {
self.emit_spaces();
self.write_str("default:\n");
self.new_scope();
}
Stmt::EndDefaultCase => {
self.emit_spaces();
self.write_str("break;");
self.close_scope();
}
Stmt::Loop { label } => {
self.emit_spaces();

if let Some(label) = label {
self.write_str(&label);
self.write_str(":");
}

self.write_str("for (;;){\n");
self.new_scope();
}
Stmt::EndLoop => {
self.close_scope();
self.emit_spaces();
self.write_str("}");
}
Stmt::Try => {
self.emit_spaces();
self.write_str("try {\n");
self.new_scope();
}
Stmt::EndTry => {
self.close_scope();
self.emit_spaces();
self.write_str("}\n");
}
Stmt::Catch(id, ty) => {
self.emit_spaces();
self.write_str("catch (");
self.write_var(*id);
self.write_str(":");
self.format_ty(ty);
```

```
self.write_str("){\n");
self.new_scope();
}
Stmt::EndCatch => {
self.close_scope();
self.emit_spaces();
self.write_str("}\n");
}
Stmt::Finally => {
self.emit_spaces();
self.write_str("finally {\n");
self.new_scope();
}
Stmt::EndFinally => {
self.close_scope();
self.emit_spaces();
self.write_str("}\n");
}
Stmt::Break(label) => {
self.emit_spaces();

if let Some(label) = label {
self.write_str("break ");
self.write_str(&label);
self.write_str("\n")
} else {
self.write_str("break\n");
}
}
Stmt::Continue(label) => {
self.emit_spaces();

if let Some(label) = label {
self.write_str("continue ");
self.write_str(&label);
self.write_str("\n")
} else {
self.write_str("continue\n");
}
}
Stmt::Return(r) => {
self.emit_spaces();
self.write_str("return ");
self.format_expr(r);
```

```
self.write_str("\n");
}
Stmt::Throw(t) => {
self.emit_spaces();
self.write_str("throw ");
self.format_expr(t);
self.write_str("\n")
}
Stmt::Expr(e) => {
self.emit_spaces();
self.format_expr(e);
self.write_str("\n")
}
}
}

fn write_var(&mut self, id: VariableId) {
self.write_str("var");
let mut buf = native_js_common::itoa::Buffer::new();
self.write_str(buf.format(id.0));
}

pub fn format_ty(&mut self, ty: &Type) {
match ty {
Type::Alias(_) => unreachable!(),
Type::Any => self.write_str("any"),
Type::AnyObject => self.write_str("object"),
Type::LiteralObject(obj) => {
self.write_str("{");
for (p, ty) in obj.iter() {
self.format_propname(p);
self.write_str(":");
self.format_ty(ty);
self.write_str(", ");
}
self.write_str("}");
}
Type::Array(a) => {
self.format_ty(a);
self.write_str("[]")
}
Type::Bigint => self.write_str("bigint"),
Type::LiteralBigint(b) => self.write_int(*b),
Type::Bool => self.write_str("boolean"),
```

```
Type::LiteralBool(b) => self.write_str(if *b { "true" } else { "false" }),
Type::Enum(e) => {
self.write_str("enum");
self.write_int(e.0);
}
Type::Function(f) => {
self.write_str("(this:");
self.format_ty(&f.this_ty);

for p in &f.params {
self.write_str(",");
self.format_ty(p);
}
self.write_str(")=>");
self.format_ty(&f.return_ty);
}
Type::Interface(id) => {
self.write_str("iface");
self.write_int(id.0);
}
Type::Object(id) => {
self.write_str("class");
self.write_int(id.0);
}
Type::Generic(id) => {
self.write_str("generic");
self.write_int(id.0);
}
Type::Int | Type::Number => self.write_str("number"),
Type::LiteralInt(i) => {
self.write_int(*i);
}
Type::LiteralNumber(n) => {
self.write_str(&n.to_string());
}
Type::Map(k, v) => {
self.write_str("Map<");
self.format_ty(k);
self.write_str(",");
self.format_ty(v);
self.write_str(">");
}
Type::Iterator(e) => {
self.write_str("Iterator<");
```

```
self.format_ty(e);
self.write_str(">")
}
Type::Null => self.write_str("null"),
Type::Promise(p) => {
self.write_str("Promise<");
self.format_ty(p);
self.write_str(">");
}
Type::Regex => self.write_str("Regex"),
Type::String => self.write_str("string"),
Type::LiteralString(s) => self.write_str(&s),
Type::Symbol => self.write_str("symbol"),
Type::Undefined => self.write_str("undefined"),
Type::Tuple(tu) => {
self.write_str("[");
for (i, t) in tu.iter().enumerate() {
self.format_ty(t);

if i != tu.len() - 1 {
self.write_str(",")
}
}
self.write_str("]")
}
Type::Union(u) => {
for (i, t) in u.iter().enumerate() {
self.format_ty(t);
if i != u.len() - 1 {
self.write_str(" | ")
}
}
}
}
}
pub fn format_expr(&mut self, expr: &Expr) {
match expr {
Expr::Array { values } => {
self.write_str("[");

for (i, v) in values.iter().enumerate() {
self.format_expr(v);
if i != values.len() - 1 {
self.write_str(",")
```

```
}
}
self.write_str("]")
}
Expr::AssertNonNull(e) => {
self.format_expr(e);
self.write_str("!");
}
Expr::Await(a) => {
self.write_str("await ");
self.format_expr(a);
}
Expr::Bigint(i) => {
let mut buf = itoa::Buffer::new();
self.write_str(buf.format(*i));
self.write_str("n")
}
Expr::Bin { op, left, right } => {
self.write_str("(");
self.format_expr(&left);
self.write_str(")");

self.write_str(op.as_str());

self.write_str("(");
self.format_expr(&right);
self.write_str(")");
}
Expr::Bool(b) => {
if *b {
self.write_str("true")
} else {
self.write_str("false")
}
}
Expr::Call {
callee,
args,
optional,
} => {
match callee.as_ref() {
Callee::Expr(e) => self.format_expr(e),
Callee::Function(id) => {
self.write_str("fun");
```

```
self.write_int(id.0)
}
Callee::Member { object, prop } => {
self.format_expr(object);
self.format_propname_or_expr(prop);
}
Callee::Super(_) => self.write_str("super"),
};

if *optional {
self.write_str("?.(")
} else {
self.write_str("(");
}

for (i, arg) in args.iter().enumerate() {
self.format_expr(arg);

if i != args.len() - 1 {
self.write_str(",")
}
}

self.write_str(")")
}
Expr::Cast(e, ty) => {
self.format_expr(e);
self.write_str(" as ");
self.format_ty(ty);
}
Expr::Closure(id) => {
self.write_str("function fun");
self.write_int(id.0);
self.format_function_body(*id);
}
Expr::Function(id) => {
self.write_str("fun");
self.write_int(id.0);
}
Expr::Int(i) => self.write_int(*i),
Expr::Number(f) => self.write_str(&f.to_string()),
Expr::Member {
object,
key,
```

```
optional,
} => {
self.format_expr(&object);

if *optional {
if let PropNameOrExpr::PropName(PropName::Ident(id)) = key {
self.write_str("?.");
self.write_str(&id);
} else {
self.write_str("?.");
self.format_propname_or_expr(key);
}
} else {
self.format_propname_or_expr(key);
}
}
Expr::MemberAssign {
op,
object,
key,
value,
} => {
self.format_expr(&object);
self.format_propname_or_expr(key);

self.write_str(op.as_str());
self.format_expr(&value);
}
Expr::MemberUpdate { op, object, key } => {
match op {
UpdateOp::PrefixAdd => self.write_str("++"),
UpdateOp::PrefixSub => self.write_str("--"),
_ => {}
}
self.format_expr(&object);
self.format_propname_or_expr(key);

match op {
UpdateOp::SuffixAdd => self.write_str("++"),
UpdateOp::SuffixSub => self.write_str("--"),
_ => {}
}
}
Expr::New { class, args } => {
```

```
self.write_str("new class");
self.write_int(class.0);

self.write_str("(");

for (i, arg) in args.iter().enumerate() {
self.format_expr(arg);
if i != args.len() - 1 {
self.write_str(",")
}
}
self.write_str(")");
}
Expr::Null => self.write_str("null"),
Expr::Object { props } => {
self.write_str("{");
for (p, v) in props {
self.format_propname(p);
self.write_str(":");
self.format_expr(v);
self.write_str(",")
}
self.write_str("}")
}
Expr::Push(e) => {
self.write_str("__pushstack__(");
self.format_expr(e);
self.write_str(")")
}
Expr::Pop => {
self.write_str("__popstack__()");
}
Expr::ReadStack => {
self.write_str("__readstack__()");
}
// todo: regex
Expr::Regex() => {}
Expr::Seq(a, b) => {
self.write_str("(");
self.format_expr(a);
self.write_str(",");
self.format_expr(b);
self.write_str(")")
}
```

```
Expr::String(s) => {
self.write_str("\"");
self.write_str(s);
self.write_str("\"");
}
Expr::Symbol(s) => {
self.write_str(&s.to_string());
}
Expr::Ternary { test, left, right } => {
self.format_expr(&test);
self.write_str("?");
self.format_expr(&left);
self.write_str(":");
self.format_expr(&right);
}
Expr::This => self.write_str("this"),
Expr::Tuple { values } => {
self.write_str("[");

for (i, v) in values.iter().enumerate() {
self.format_expr(v);
if i != values.len() - 1 {
self.write_str(",")
}
}
self.write_str("]")
}
Expr::Unary { op, value } => {
let op = match op {
UnaryOp::BitNot => "~",
UnaryOp::LogicalNot => "!",
UnaryOp::Minus => "-",
UnaryOp::Plus => "+",
UnaryOp::Typeof => "typeof ",
UnaryOp::Void => "void ",
};
self.write_str(op);
self.write_str("(");
self.format_expr(&value);
self.write_str(")");
}
Expr::Undefined => self.write_str("undefined"),
Expr::VarAssign {
op,
```

```
variable,
value,
} => {
self.write_str("var");
self.write_int(variable.0);

self.write_str(op.as_str());

self.format_expr(&value);
}
Expr::VarLoad { span: _, variable } => {
self.write_str("var");
self.write_int(variable.0);
}
Expr::VarUpdate { op, variable } => {
match op {
UpdateOp::PrefixAdd => self.write_str("++"),
UpdateOp::PrefixSub => self.write_str("--"),
_ => {}
}
self.write_str("var");
self.write_int(variable.0);

match op {
UpdateOp::SuffixAdd => self.write_str("++"),
UpdateOp::SuffixSub => self.write_str("--"),
_ => {}
}
}
Expr::Yield(y) => {
self.write_str("yield ");
self.format_expr(y);
}
}
}

fn format_function_body(&mut self, id: FunctionId) {
let func = self.table.functions.get(&id).expect("invalid function");
self.write_str("(this:");
self.format_ty(&func.this_ty);

for p in func.params.iter() {
self.write_str(", var");
self.write_int(p.id.0);
```

```
self.write_str(":");
self.format_ty(&p.ty);
}
self.write_str("):");
self.format_ty(&func.return_ty);
self.write_str("{\n");
self.new_scope();

for s in &func.stmts {
self.format_stmt(s);
}

self.close_scope();
self.emit_spaces();
self.write_str("}\n");
}

fn format_propname_or_expr(&mut self, prop: &PropNameOrExpr) {
match prop {
PropNameOrExpr::PropName(p) => self.format_propname(p),
PropNameOrExpr::Expr(e, _ty) => {
self.write_str("[");
self.format_expr(e);
self.write_str("]")
}
}
}

fn format_propname(&mut self, prop: &PropName) {
match prop {
PropName::Ident(id) => {
self.write_str(".");
self.write_str(id);
}
PropName::Int(i) => {
self.write_str("[");
self.write_int(*i);
self.write_str("]");
}
PropName::Private(p) => {
self.write_str(".#");
self.write_str(p);
}
PropName::String(s) => {
```

```
self.write_str("[\"");
self.write_str(s);
self.write_str("\"]");
}
PropName::Symbol(s) => {
self.write_str("[");
self.write_str(&s.to_string());
self.write_str("]")
}
}
}
}
```

5.2.9 native-ts-hir/transform/mod.rs

```
mod class;
mod context;
mod expr;
mod function;
mod module;
mod stmt;
mod types;

use std::{
collections::{HashMap, HashSet},
sync::atomic::{AtomicUsize, Ordering},
};

use context::*;

use native_js_common::error::Error;
use native_ts_parser::swc_core::common::Span;
use native_ts_parser::swc_core::ecma::ast as swc;

use crate::{
ast::{self, Expr, Function, FunctionParam, ModuleExport, Stmt, Type},
common::{AliasId, ClassId, EnumId, FunctionId, InterfaceId, VariableId},
symbol_table::SymbolTable,
PropName,
};

type Result<T> = std::result::Result<T, Error<Span>>;

/// span, ty, fulfills
struct TypeCheck {
```

```
span: Span,
ty: Type,
fulfills: Type,
}

pub struct Transformer {
/// pended type checks tha cannot be done during translation
type_checks: Vec<TypeCheck>,
/// contains scope and definitions
context: Context,

break_labels: HashSet<String>,
continue_labels: HashSet<String>,

/// the current this type
this_ty: Type,
return_ty: Type,
/// the current super class
super_class: Option<ClassId>,
/// indicates if current context is constructor
is_in_constructor: bool,
}

impl Transformer {
pub fn new() -> Self {
Self {
type_checks: Vec::new(),
context: Context::new(),
break_labels: Default::default(),
continue_labels: Default::default(),
this_ty: Type::Any,
return_ty: Type::Undefined,
super_class: None,
is_in_constructor: false,
}
}
pub fn anonymous_name(&self) -> String {
static COUNT: AtomicUsize = AtomicUsize::new(0);

let mut buf = native_js_common::itoa::Buffer::new();

return "anonymous".to_string() + buf.format(COUNT.fetch_add(1,
Ordering::SeqCst));
}
```

```
pub fn transform_module(&mut self, module: &swc::Module) ->
Result<crate::ast::Module> {
let mut export_default = ModuleExport::Undefined;
let mut module_exports = HashMap::new();

for i in &module.body {
if let swc::ModuleItem::ModuleDecl(swc::ModuleDecl::ExportDefaultDecl(d)) = i
{
let re = match &d.decl {
swc::DefaultDecl::Class(c) => {
self.hoist_class(c.ident.as_ref().map(|id| id.sym.as_ref()), &c.class)
}
swc::DefaultDecl::Fn(f) => {
self.hoist_function(f.ident.as_ref().map(|id| id.sym.as_ref()), &f.function)?;
Ok(())
}
swc::DefaultDecl::TsInterfaceDecl(i) => {
self.hoist_interface(Some(&i.id.sym), &i)
}
};

if let Err(e) = re {
return Err(e);
}
}
}

// hoist
self.hoist(module.body.iter().filter_map(|i| {
if let Some(m) = i.as_module_decl() {
match m {
swc::ModuleDecl::ExportDecl(d) => {
return Some(&d.decl);
}
_ => None,
}
} else {
return i.as_stmt().and_then(|s| s.as_decl());
}
}))?;

for item in &module.body {
match item {
```

```
swc::ModuleItem::ModuleDecl(d) => match d {
swc::ModuleDecl::ExportDefaultDecl(decl) => match &decl.decl {
swc::DefaultDecl::Class(c) => {
let id = c
.ident
.as_ref()
.map(|id| self.context.get_class_id(&id.sym))
.unwrap_or(ClassId::new());
self.translate_class(
id,
c.ident
.as_ref()
.map(|i| i.sym.as_ref())
.unwrap_or("default")
.to_string(),
&c.class,
)?;
self.context.func().stmts.push(Stmt::DeclareClass(id));

export_default = ModuleExport::Class(id);
}
swc::DefaultDecl::Fn(f) => {
let id = f
.ident
.as_ref()
.map(|id| self.context.get_func_id(&id.sym))
.unwrap_or(FunctionId::new());
self.translate_function(id, None, &f.function)?;
self.context.func().stmts.push(Stmt::DeclareFunction(id));

export_default = ModuleExport::Function(id);
}
swc::DefaultDecl::TsInterfaceDecl(i) => {
let id = self.context.get_interface_id(&i.id.sym);
self.context.func().stmts.push(Stmt::DeclareInterface(id));

export_default = ModuleExport::Interface(id);
}
},
swc::ModuleDecl::ExportDefaultExpr(expr) => {
let varid = VariableId::new();
let (expr, ty) = self.translate_expr(&expr.expr, None)?;
self.context
.func()
```

```
.stmts
.push(Stmt::DeclareVar(varid, ty.clone()));
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: varid,
value: Box::new(expr),
})));

export_default = ModuleExport::Var(varid, ty);
}
swc::ModuleDecl::ExportDecl(decl) => {
self.translate_decl(&decl.decl)?;
}
swc::ModuleDecl::ExportNamed(n) => {
if let Some(src) = &n.src {
let module_id = self.find_module(&src.value);

for s in &n.specifiers {
match s {
swc::ExportSpecifier::Namespace(n) => {
let name = self.translate_module_export_name(&n.name);
module_exports
.insert(name, ModuleExport::NameSpace(module_id));
}
swc::ExportSpecifier::Default(d) => {
let name = PropName::Ident(d.exported.sym.to_string());
module_exports
.insert(name, self.module_default_export(module_id));
}
swc::ExportSpecifier::Named(n) => {
let origin_name =
self.translate_module_export_name(&n.orig);
let module_export =
self.module_export(module_id, &origin_name);

if module_export.is_none() {
return Err(Error::syntax_error(
n.span,
format!(
"module '{}' has no export '{}'",
src.value, origin_name
),
));
}
```

```
let module_export = module_export.unwrap();

let exported_name = if let Some(exported) = &n.exported {
self.translate_module_export_name(exported)
} else {
origin_name
};

if n.is_type_only {
match &module_export {
ModuleExport::Var(_, _)
| ModuleExport::NameSpace(_) => {
return Err(Error::syntax_error(
n.span,
"type only export can only export type",
))
}
_ => {}
}
}

module_exports.insert(exported_name, module_export);
}
}
}
} else {
for s in &n.specifiers {
match s {
swc::ExportSpecifier::Named(n) => {
let origin_name = match &n.orig {
swc::ModuleExportName::Ident(id) => id.sym.to_string(),
swc::ModuleExportName::Str(_) => unimplemented!(),
};

let binding = if let Some(bind) =
self.context.find(&origin_name)
{
bind
} else {
return Err(Error::syntax_error(
n.span,
format!("undefined identifier '{}'", origin_name),
));
};
```

```
let export = match binding {
Binding::Class(c) => ModuleExport::Class(*c),
Binding::GenericClass(_) => todo!("export generic"),
Binding::Enum(e) => ModuleExport::Enum(*e),
Binding::Function(f) => ModuleExport::Function(*f),
Binding::GenericFunction(_) => todo!("export generic"),
Binding::Generic(_) => unreachable!(),
Binding::Interface(i) => ModuleExport::Interface(*i),
Binding::GenericInterface(_) => todo!("export generic"),
Binding::TypeAlias(id) => ModuleExport::Alias(*id),
Binding::GenericTypeAlias(_) => todo!("export generic"),
Binding::Using { .. } => {
// TODO: export using
return Err(Error::syntax_error(
n.span,
"export 'using' declare is not allowed",
));
}
Binding::Var { id, ty, .. } => {
ModuleExport::Var(*id, ty.clone())
}
Binding::NameSpace(n) => ModuleExport::NameSpace(*n),
};

let exported_name = if let Some(exported) = &n.exported {
self.translate_module_export_name(exported)
} else {
PropName::Ident(origin_name)
};

module_exports.insert(exported_name, export);
}
_ => unreachable!(),
}
}
}
}
_ => {}
},
swc::ModuleItem::Stmt(s) => {
// translate statement
self.translate_stmt(s, None)?;
}
```

```
}
}

// finish up type checks
for check in &self.type_checks {
self.type_check(check.span, &check.ty, &check.fulfills)?;
}

let main = self.context.end_function();

return Ok(ast::Module {
table: SymbolTable {
external_functions: Default::default(),
functions: core::mem::replace(&mut self.context.functions, Default::default()),
classes: core::mem::replace(&mut self.context.classes, Default::default()),
interfaces: core::mem::replace(&mut self.context.interfaces, Default::default()),
enums: core::mem::replace(&mut self.context.enums, Default::default()),
},
main_function: main,
default_export: export_default,
exports: module_exports,
});
}

pub fn hoist_stmts<'a, I: Iterator<Item = &'a swc::Stmt> + Clone>(
&mut self,
stmts: I,
) -> Result<()> {
self.hoist(stmts.filter_map(|s| s.as_decl()))
}

pub fn hoist<'a, I: Iterator<Item = &'a swc::Decl> + Clone>(&mut self, stmts:
I) -> Result<()> {
// hoist the type names
for decl in stmts.clone() {
match decl {
swc::Decl::Class(class) => {
self.hoist_class(Some(&class.ident.sym), &class.class)?;
}
swc::Decl::TsEnum(e) => {
self.hoist_enum(&e)?;
}
swc::Decl::TsInterface(iface) => {
self.hoist_interface(Some(&iface.id.sym), &iface)?;
```

```
}
swc::Decl::TsModule(_m) => {
todo!("module declare")
}
swc::Decl::TsTypeAlias(alias) => {
// translate later
self.hoist_alias(&alias.id.sym, alias)?;
}
// hoist later
swc::Decl::Fn(_) => {}
swc::Decl::Var(_) => {}
// do not hoist using
swc::Decl::Using(_) => {}
}
}

// translate type alias
for decl in stmts.clone() {
if let swc::Decl::TsTypeAlias(a) = decl {
match self.context.find(&a.id.sym) {
Some(Binding::TypeAlias(id)) => {
debug_assert!(
a.type_params.is_none()
|| a.type_params.as_ref().is_some_and(|p| p.params.is_empty())
);

let id = *id;

// translate the type
let ty = self.translate_type_alias(&a)?;

self.context.alias.insert(id, ty);
}
Some(Binding::GenericTypeAlias(_id)) => {
todo!("generic alias")
}
_ => unreachable!(),
}
}
}

// translate all the interfaces
for decl in stmts.clone() {
if let swc::Decl::TsInterface(iface) = decl {
```

```
match self.context.find(&iface.id.sym) {
Some(Binding::Interface(id)) => {
// copy the id
let id = *id;
// translate the interface
let ty = self.translate_interface(&iface)?;

let slot = self.context.interfaces.insert(id, ty);

// there should be no interface declared
debug_assert!(slot.is_none());
}
Some(Binding::GenericInterface(_id)) => {
todo!("generic interfaces")
}
_ => unreachable!(),
};
}
}

// translate classes
for decl in stmts.clone() {
if let swc::Decl::Class(class) = decl {
match self.context.find(&class.ident.sym) {
Some(Binding::Class(id)) => {
let id = *id;
let c = self.translate_class_ty(id, &class.class)?;
self.context.classes.insert(id, c);
}
Some(Binding::GenericClass(_id)) => {
todo!("generic classes")
}
_ => unreachable!(),
};
}
}

// finish translating type and normalise them
self.normalise_types();

// hoist variables
for decl in stmts.clone() {
if let swc::Decl::Var(v) = decl {
self.hoist_vardecl(v)?;
```

```
}
}

// hoist functions
for decl in stmts{
if let swc::Decl::Fn(f) = decl{
self.hoist_function(Some(&f.ident.sym), &f.function)?;
}
}

return Ok(());
}

pub fn hoist_class(&mut self, name: Option<&str>, class: &swc::Class) ->
Result<()> {
let id = ClassId::new();
if class.type_params.is_some() {
if !self
.context
.declare(name.unwrap_or(""), Binding::GenericClass(id))
{
return Err(Error::syntax_error(class.span, "duplicated identifier"));
}
} else {
if !self.context.declare(name.unwrap_or(""), Binding::Class(id)) {
return Err(Error::syntax_error(class.span, "duplicated identifier"));
}
}
return Ok(());
}

pub fn hoist_function(&mut self, name: Option<&str>, func: &swc::Function) ->
Result<FunctionId> {
let id = FunctionId::new();
if func.type_params.is_some() {
if let Some(name) = name{
if !self
.context
.declare(name, Binding::GenericFunction(id))
{
return Err(Error::syntax_error(func.span, "duplicated identifier"));
}
}
} else {
```

```
if let Some(name) = name{
if !self
.context
.declare(name, Binding::Function(id))
{
return Err(Error::syntax_error(func.span, "duplicated identifier"));
}
}

let f = self.translate_function_ty(func)?;

// insert the function type
self.context.functions.insert(
id,
Function {
this_ty: f.this_ty,
params: f
.params
.into_iter()
.map(|ty| FunctionParam {
id: VariableId::new(),
ty: ty,
})
.collect(),
return_ty: f.return_ty,
variables: Default::default(),
captures: Vec::new(),
stmts: Vec::new(),
},
);
}
return Ok(id);
}

pub fn hoist_interface(
&mut self,
name: Option<&str>,
iface: &swc::TsInterfaceDecl,
) -> Result<()> {
let id = InterfaceId::new();
if iface.type_params.is_some() {
if !self
.context
.declare(name.unwrap_or(""), Binding::GenericInterface(id))
```

```
{
return Err(Error::syntax_error(iface.span, "duplicated identifier"));
}
} else {
if !self
.context
.declare(name.unwrap_or(""), Binding::Interface(id))
{
return Err(Error::syntax_error(iface.span, "duplicated identifier"));
}
}
return Ok(());
}

pub fn hoist_enum(&mut self, e: &swc::TsEnumDecl) -> Result<()> {
let id = EnumId::new();
if !self.context.declare(&e.id.sym, Binding::Enum(id)) {
return Err(Error::syntax_error(e.id.span, "duplicated identifier"));
}

// translate enum already
let ty = self.translate_enum(e)?;

let slot = self.context.enums.insert(id, ty);

debug_assert!(slot.is_none());

return Ok(());
}

pub fn hoist_alias(&mut self, name: &str, alias: &swc::TsTypeAliasDecl) ->
Result<()> {
if alias.type_params.is_some() {
if !self
.context
.declare(name, Binding::GenericTypeAlias(AliasId::new()))
{
return Err(Error::syntax_error(alias.span, "duplicated identifier"));
}
} else {
if !self
.context
.declare(name, Binding::TypeAlias(AliasId::new()))
{
```

```
return Err(Error::syntax_error(alias.span, "duplicated identifier"));
}
}
return Ok(());
}

pub fn hoist_vardecl(&mut self, _decl: &swc::VarDecl) -> Result<()> {
return Ok(());
}

pub fn normalise_types(&mut self) {
// types does not backreference therefore this function is safe
unsafe {
for alias in (self as *mut Self)
.as_mut()
.unwrap_unchecked()
.context
.alias
.values_mut()
{
self.normalise_type(alias);
}

for iface in (self as *mut Self)
.as_mut()
.unwrap_unchecked()
.context
.interfaces
.values_mut()
{
for (_name, prop) in &mut iface.properties {
self.normalise_type(&mut prop.ty)
}
}

for class in (self as *mut Self)
.as_mut()
.unwrap_unchecked()
.context
.classes
.values_mut()
{
for (_name, prop) in &mut class.properties {
self.normalise_type(&mut prop.ty);
```

```
}
}
}
}

pub fn normalise_type(&mut self, ty: &mut Type) {
match ty {
Type::Alias(id) => {
let t = unsafe { (self as *mut Self).as_mut().unwrap_unchecked() }
.context
.alias
.get_mut(id)
.expect("invalid alias type");
self.normalise_type(t);

*ty = t.clone();
}
Type::LiteralObject(obj) => {
for (_p, ty) in obj.iter_mut() {
self.normalise_type(ty);
}
}
Type::Array(elem) => {
self.normalise_type(elem);
}
Type::Enum(_) => {}
Type::Function(func) => {
self.normalise_type(&mut func.this_ty);
self.normalise_type(&mut func.return_ty);
for param in &mut func.params {
self.normalise_type(param);
}
}
Type::Map(key, value) => {
self.normalise_type(key);
self.normalise_type(value);
}
Type::Promise(re) => {
self.normalise_type(re);
}
Type::Tuple(tys) => {
for ty in tys.iter_mut() {
self.normalise_type(ty);
}
```

```
}
Type::Iterator(ty) => {
self.normalise_type(ty);
}
Type::Union(u) => {
for i in u.iter_mut() {
self.normalise_type(i);
}

let mut has_union = false;
for i in u.iter() {
if let Type::Union(_) = i {
has_union = true;
}
}

if !has_union {
return;
}

let mut tys = Vec::new();
for i in u.iter() {
if let Type::Union(v) = i {
for ty in v.iter() {
if !tys.contains(ty) {
tys.push(ty.clone())
}
}
} else {
if !tys.contains(i) {
tys.push(i.clone());
}
}
}

tys.sort();

*ty = Type::Union(tys.into_boxed_slice());
}
Type::Any
| Type::AnyObject
| Type::Bigint
| Type::LiteralBigint(_)
| Type::Bool
```

```
| Type::LiteralBool(_)
| Type::Int
| Type::Null
| Type::Number
| Type::LiteralNumber(_)
| Type::LiteralInt(_)
| Type::Object(_)
| Type::Regex
| Type::String
| Type::LiteralString(_)
| Type::Symbol
| Type::Undefined
| Type::Interface(_)
| Type::Generic(_) => {}
}
}
}
```

5.2.10 native-ts-hir/transform/types.rs

```
use std::collections::HashMap;


use native_js_common::error::Error;

use native_ts_parser::swc_core::common::{Span, Spanned};

use native_ts_parser::swc_core::ecma::ast as swc;


use crate::ast::{

    EnumType, EnumVariantDesc, FuncType, InterfaceMethod, InterfacePropertyDesc,
InterfaceType,

    PropNameOrExpr, Type,

};

use crate::common::{AliasId, ClassId, FunctionId, InterfaceId};

use crate::{PropName, Symbol};


type Result<T> = std::result::Result<T, Error<Span>>;


use super::{context::Binding, Transformer};
```

```
impl Transformer {
    /// translate a function type without translating its contents
    pub fn translate_function_ty(&mut self, func: &swc::Function) -> Result<FuncType> {
        // generic function
        if func.type_params.is_some() {
            todo!("generic function")
        }

        // the default 'this' type
        let mut this_ty = Type::Any;
        // the default return type
        let mut return_ty = Type::Undefined;
        // stores the params
        let mut params = Vec::new();
        // is variable argument
        let is_var_arg = false;

        // loop through params
        for (i, p) in func.params.iter().enumerate() {
            // only support ident as param
            if let Some(ident) = p.pat.as_ident() {
                // translate param type
                if let Some(ann) = &ident.type_ann {
                    // translate the type
                    let ty = self.translate_type(&ann.type_ann)?;

                    // if ident is 'this' and is first, this type is set
                    if i == 0 && ident.sym.as_ref() == "this" {
                        this_ty = ty;
                    } else {
                        // push param type
                        params.push(ty);
```

```
            };
        } else {
            // function param must have type annotation
            return Err(Error::syntax_error(ident.span, "missing type annotation"));
        }
    } else if let Some(rest) = p.pat.as_rest() {
        // variable agument

        // variable argument can only be declared at the last param
        if i != func.params.len() - 1 {
            return Err(Error::syntax_error(
                rest.dot3_token,
                "variable arguments is only allowed at the last param",
            ));
        }

        // TODO: variable argument
        return Err(Error::syntax_error(
            rest.dot3_token,
            "variable arguments not supported",
        ));
    } else {
        // we currently only supports ident params
        return Err(Error::syntax_error(
            p.span,
            "destructive params not allowed",
        ));
    }
}

// translate the return type if any
if let Some(ann) = &func.return_type {
```

```
        return_ty = self.translate_type(&ann.type_ann)?;
    }

    // set function to async if declared
    if func.is_async {
        return_ty = Type::Promise(Box::new(return_ty));
    }

    // set function to generator if declared
    if func.is_generator {
        return_ty = Type::Iterator(Box::new(return_ty));
    }

    // return the function type
    return Ok(FuncType {
        this_ty,
        params,
        var_arg: is_var_arg,
        return_ty,
    });
}

/// translate an interface type for an interface declare
pub fn translate_interface(&mut self, iface: &swc::TsInterfaceDecl) -> Result<InterfaceType> {
    // interface type body
    let mut iface_ty = InterfaceType {
        name: iface.id.sym.to_string(),
        extends: Vec::new(),
        implements: Vec::new(),
        properties: HashMap::new(),
        methods: HashMap::new(),
    };
```

```
// translate constrains
for ty in &iface.extends {
    // translate type arguments
    let type_args = if let Some(type_args) = &ty.type_args {
        self.translate_type_args(&type_args)?
    } else {
        // no type arguments
        Vec::new()
    };

    // translate extended type
    let t = self.translate_expr_type(&ty.expr, &type_args)?;

    match t {
        // a class
        Type::Object(class_id) => iface_ty.extends.push(class_id),
        // an interface
        Type::Interface(iface_id) => iface_ty.implements.push(iface_id),
        // other types are not allowed
        _ => {
            return Err(Error::syntax_error(
                ty.span,
                format!("expected class or interface, found type '{:?}'", t),
            ))
        }
    }
}

// generic interface
if let Some(_) = &iface.type_params {
    panic!("generic interface")
```

```
}

// translate interface body
for elem in &iface.body.body {
    match elem {
        swc::TsTypeElement::TsCallSignatureDecl(c) => {
            // TODO: call signature declare
            return Err(Error::syntax_error(c.span, "call signature not allowed"));
        }
        swc::TsTypeElement::TsConstructSignatureDecl(c) => {
            // TODO: construct signature declare
            return Err(Error::syntax_error(
                c.span,
                "constructor signature not allowed",
            ));
        }
        swc::TsTypeElement::TsIndexSignature(i) => {
            // TODO: index signature
            return Err(Error::syntax_error(i.span, "index signature not allowed"));
        }
        swc::TsTypeElement::TsGetterSignature(g) => {
            // TODO: getter
            return Err(Error::syntax_error(g.span, "getter not supported"));
        }
        swc::TsTypeElement::TsSetterSignature(s) => {
            // TODO: setter
            return Err(Error::syntax_error(s.span, "setter not supported"));
        }
        swc::TsTypeElement::TsPropertySignature(p) => {
            // initialiser not allowed in interface
            if let Some(init) = &p.init {
                return Err(Error::syntax_error(init.span(), "initialiser not allowed"));
```

```
}
//
if let Some(type_params) = &p.type_params {
    return Err(Error::syntax_error(
        type_params.span,
        "generics not allowed",
    ));
}
//
if !p.params.is_empty() {
    return Err(Error::syntax_error(p.span, "params not allowed"));
}
// translate property name
let key = if p.computed {
    // computed property name
    self.translate_computed_prop_name(&p.key)?
} else {
    // identifier property name
    if let Some(id) = p.key.as_ident() {
        PropNameOrExpr::PropName(PropName::Ident(id.sym.to_string()))
    } else {
        self.translate_computed_prop_name(&p.key)?
    }
};

// match property name
let key = match key {
    PropNameOrExpr::Expr(..) => {
        // property must be known at compile time
        return Err(Error::syntax_error(
            p.key.span(),
            "property of interface must be literal",
```

```
        ));
    }
    // a property name
    PropNameOrExpr::PropName(p) => p,
};

// check if property name is already declared
if iface_ty.properties.contains_key(&key) {
    return Err(Error::syntax_error(p.span, "duplicated attributes"));
}

// translate type annotation
if let Some(ann) = &p.type_ann {
    let mut ty = self.translate_type(&ann.type_ann)?;

    // optional type
    if p.optional {
        ty = ty.union(Type::Undefined);
    }

    // insert property
    iface_ty.properties.insert(
        key,
        InterfacePropertyDesc {
            ty: ty,
            readonly: p.readonly,
            optional: p.optional,
        },
    );
} else {
    // property must be annotated
    return Err(Error::syntax_error(p.span, "missing type annotation"));
```

```
    }
}
// an interface method
swc::TsTypeElement::TsMethodSignature(m) => {
    // generic method
    if let Some(type_params) = &m.type_params {
        return Err(Error::syntax_error(
            type_params.span,
            "generics not allowed",
        ));
    }

    // the method body
    let mut method_ty = InterfaceMethod {
        readonly: m.readonly,
        optional: m.optional,
        params: Vec::new(),
        return_ty: Type::Undefined,
    };

    // method name
    let key = if m.computed {
        self.translate_computed_prop_name(&m.key)?
    } else {
        // ident name
        if let Some(id) = m.key.as_ident() {
            PropNameOrExpr::PropName(PropName::Ident(id.sym.to_string()))
        } else {
            self.translate_computed_prop_name(&m.key)?
        }
    };
```

```
// match key
let key = match key {
    PropNameOrExpr::Expr(..) => {
        // dynamic names are not allowed
        return Err(Error::syntax_error(
            m.key.span(),
            "property of interface must be literal",
        ));
    }
    PropNameOrExpr::PropName(p) => p,
};

// check if method already declared
if iface_ty.methods.contains_key(&key) {
    return Err(Error::syntax_error(m.span, "duplicated methods"));
}

// translate parameters
for param in &m.params {
    match param {
        swc::TsFnParam::Ident(ident) => {
            if let Some(ann) = &ident.type_ann {
                let mut ty = self.translate_type(&ann.type_ann)?;

                if ident.optional {
                    ty = ty.union(Type::Undefined);
                }
                method_ty.params.push(ty);
            }
        }
        _ => {
            return Err(Error::syntax_error(
```

```
                        param.span(),
                        "destructive params not allowed",
                    ))
                }
            }
        }

        if let Some(ann) = &m.type_ann {
            let ty = self.translate_type(&ann.type_ann)?;
            method_ty.return_ty = ty;
        }

        iface_ty.methods.insert(key, method_ty);
    }
  }
}

  return Ok(iface_ty);
}

pub fn translate_enum(&mut self, e: &swc::TsEnumDecl) -> Result<EnumType> {
  let mut variants = Vec::new();

  for m in &e.members {
    let name = match &m.id {
      swc::TsEnumMemberId::Ident(id) => PropName::Ident(id.sym.to_string()),
      swc::TsEnumMemberId::Str(s) => PropName::String(s.value.to_string()),
    };
    variants.push(EnumVariantDesc { name: name });
  }

  Ok(EnumType {
```

```
        name: e.id.sym.to_string(),
        variants: variants,
    })
}

pub fn translate_type_alias(&mut self, alias: &swc::TsTypeAliasDecl) -> Result<Type> {
    if alias.type_params.is_none() {
        return self.translate_type(&alias.type_ann);
    }

    todo!()
}

pub fn translate_expr_type(&mut self, expr: &swc::Expr, type_args: &[Type]) -> Result<Type> {
    let binding = match self.find_expr_binding(&expr) {
        Some(b) => b,
        None => return Err(Error::syntax_error(expr.span(), "undefined identifier")),
    };

    let mut generics_allowed = false;

    let ty = match binding {
        Binding::Class(c) => Type::Object(c),
        Binding::Enum(e) => Type::Enum(e),
        Binding::Interface(i) => Type::Interface(i),
        Binding::TypeAlias(id) => {
            if let Some(ty) = self.context.alias.get(&id) {
                ty.clone()
            } else {
                Type::Alias(id)
            }
        }
```

```
Binding::GenericFunction(_) | Binding::Function(_) => {
    return Err(Error::syntax_error(
        expr.span(),
        "expected type, found function",
    ))
}
Binding::Generic(_) => todo!("generics"),
Binding::GenericClass(id) => {
    generics_allowed = true;

    if type_args.len() == 0 {
        return Err(Error::syntax_error(expr.span(), "missing type arguments"));
    }
    let id = self.solve_generic_class(id, type_args)?;
    Type::Object(id)
}

Binding::GenericInterface(id) => {
    generics_allowed = true;

    if type_args.is_empty() {
        return Err(Error::syntax_error(expr.span(), "missing type arguments"));
    }

    let id = self.solve_generic_interface(id, type_args)?;
    Type::Interface(id)
}
Binding::GenericTypeAlias(id) => {
    generics_allowed = true;

    if type_args.is_empty() {
        return Err(Error::syntax_error(expr.span(), "missing type arguments"));
```

```
        }

        let id = self.solve_generic_alias(id, type_args)?;

        if let Some(ty) = self.context.alias.get(&id) {
            ty.clone()
        } else {
            Type::Alias(id)
        }
    }
    Binding::NameSpace(_) => {
        return Err(Error::syntax_error(
            expr.span(),
            "expected type, found namespace",
        ))
    }
    Binding::Using { .. } | Binding::Var { .. } => {
        return Err(Error::syntax_error(
            expr.span(),
            "expected type, found variable",
        ))
    }
};

if !generics_allowed {
    if !type_args.is_empty() {
        return Err(Error::syntax_error(
            expr.span(),
            "expected 0 type arguments",
        ));
    }
}
```

```
    return Ok(ty);
}

fn find_expr_binding(&mut self, expr: &swc::Expr) -> Option<Binding> {
    match expr {
        swc::Expr::Paren(p) => return self.find_expr_binding(&p.expr),
        swc::Expr::Member(m) => {
            let binding = self.find_expr_binding(&m.obj)?;

            match binding {
                Binding::NameSpace(mid) => {
                    let prop = match &m.prop {
                        swc::MemberProp::Computed(_) => return None,
                        swc::MemberProp::Ident(id) => PropName::Ident(id.to_string()),
                        swc::MemberProp::PrivateName(_) => return None,
                    };
                    return self.find_binding_from_module(mid, &prop);
                }
                _ => return None,
            }
        }
        swc::Expr::Ident(id) => return self.context.find(&id.sym).cloned(),
        _ => None,
    }
}

pub fn translate_type(&mut self, ty: &swc::TsType) -> Result<Type> {
    match ty {
        swc::TsType::TsArrayType(array) => {
            let member = self.translate_type(&array.elem_type)?;
            return Ok(Type::Array(Box::new(member)));
```

```
}
swc::TsType::TsConditionalType(c) => {
    let check_ty = self.translate_type(&c.check_type)?;
    let extends_ty = self.translate_type(&c.extends_type)?;
    let false_ty = self.translate_type(&c.false_type)?;
    let true_ty = self.translate_type(&c.true_type)?;

    let t = self.type_check(c.span, &check_ty, &extends_ty).is_ok();

    if t {
        return Ok(true_ty);
    } else {
        return Ok(false_ty);
    }
}
swc::TsType::TsFnOrConstructorType(func) => {
    let func = self.translate_func_type(func)?;

    return Ok(Type::Function(Box::new(func)));
}
swc::TsType::TsImportType(t) => {
    // TODO: import types
    return Err(Error::syntax_error(t.span, "import types not supported"));
}
swc::TsType::TsIndexedAccessType(i) => {
    if i.readonly {
        // todo!()
    }

    let index_ty = self.translate_type(&i.index_type)?;
    let value_ty = self.translate_type(&i.obj_type)?;
```

```
    return Ok(Type::Map(Box::new(index_ty), Box::new(value_ty)));
}
swc::TsType::TsInferType(i) => {
    // TODO: infer types
    return Err(Error::syntax_error(i.span, "infer types not supported"));
}
swc::TsType::TsKeywordType(key) => {
    let ty = match key.kind {
        swc::TsKeywordTypeKind::TsAnyKeyword => Type::Any,
        swc::TsKeywordTypeKind::TsBigIntKeyword => Type::Bigint,
        swc::TsKeywordTypeKind::TsBooleanKeyword => Type::Bool,
        swc::TsKeywordTypeKind::TsIntrinsicKeyword => {
            return Err(Error::syntax_error(
                key.span,
                "instrinsic types not supported",
            ))
        }
        swc::TsKeywordTypeKind::TsNeverKeyword => {
            //return Err(Error::syntax_error(key.span, "never type not supported"))
            Type::Undefined
        }
        swc::TsKeywordTypeKind::TsNullKeyword => Type::Null,
        swc::TsKeywordTypeKind::TsNumberKeyword => Type::Number,
        swc::TsKeywordTypeKind::TsObjectKeyword => Type::AnyObject,
        swc::TsKeywordTypeKind::TsStringKeyword => Type::String,
        swc::TsKeywordTypeKind::TsSymbolKeyword => Type::Symbol,
        swc::TsKeywordTypeKind::TsUndefinedKeyword => Type::Undefined,
        swc::TsKeywordTypeKind::TsUnknownKeyword => Type::Any,
        swc::TsKeywordTypeKind::TsVoidKeyword => {
            Type::Union(Box::new([Type::Undefined, Type::Null]))
        }
    };
```

```
    return Ok(ty);
}
swc::TsType::TsLitType(l) => {
    return Err(Error::syntax_error(l.span, "literal types not supported"))
}
swc::TsType::TsMappedType(m) => {
    return Err(Error::syntax_error(m.span, "mapped types not supported"))
}
swc::TsType::TsOptionalType(t) => {
    let ty = self.translate_type(&t.type_ann)?;

    return Ok(ty.union(Type::Undefined));
}
swc::TsType::TsParenthesizedType(p) => return self.translate_type(&p.type_ann),
swc::TsType::TsRestType(t) => {
    return Err(Error::syntax_error(t.span, "rest type not supported"))
}
swc::TsType::TsThisType(_) => return Ok(self.this_ty.clone()),
swc::TsType::TsTupleType(t) => {
    let mut tys = Vec::new();
    for i in &t.elem_types {
        let ty = self.translate_type(&i.ty)?;
        tys.push(ty);
    }

    return Ok(Type::Tuple(tys.into_boxed_slice()));
}
swc::TsType::TsTypeLit(l) => {
    return Err(Error::syntax_error(l.span, "type literal not supported"))
}
swc::TsType::TsTypeOperator(o) => {
```

```
    let ty = self.translate_type(&o.type_ann)?;

    match o.op {
        swc::TsTypeOperatorOp::KeyOf => match ty {
            Type::Map(k, _) => return Ok(*k),
            _ => {
                return Err(Error::syntax_error(
                    o.span,
                    "expected index accessing type",
                ))
            }
        },
        swc::TsTypeOperatorOp::ReadOnly => return Ok(ty),
        swc::TsTypeOperatorOp::Unique => return Ok(ty),
    }
}
swc::TsType::TsTypePredicate(p) => match &p.param_name {
    swc::TsThisTypeOrIdent::Ident(_ident) => {
        if let Some(ty) = &p.type_ann {
            let _ty = self.translate_type(&ty.type_ann)?;
            return Ok(Type::Bool);
        } else {
            return Err(Error::syntax_error(p.span, "missing type annotation"));
        }
    }
    swc::TsThisTypeOrIdent::TsThisType(_t) => {
        if let Some(ann) = &p.type_ann {
            let _ty = self.translate_type(&ann.type_ann)?;

            return Ok(Type::Bool);
        } else {
            return Err(Error::syntax_error(p.span, "missing type annotation"));
```

```
        }
    }
},
swc::TsType::TsTypeQuery(q) => {
    // typeof operator
    let mut allow_generics = false;

    let ty = match &q.expr_name {
        swc::TsTypeQueryExpr::TsEntityName(name) => match self.find_binding(name) {
            Some(Binding::GenericClass(_)) | Some(Binding::Class(_)) => {
                return Err(Error::syntax_error(
                    q.span,
                    "cannot infer type, class is not a value",
                ))
            }
            Some(Binding::Enum(_)) => {
                return Err(Error::syntax_error(
                    q.span,
                    "cannot infer type, enum is not a value",
                ))
            }
            Some(Binding::GenericFunction(id)) => {
                allow_generics = true;

                if let Some(type_args) = &q.type_args {
                    let type_args = self.translate_type_args(&type_args)?;
                    let id = self.solve_generic_function(id, &type_args)?;

                    let func = self.context.functions.get(&id).unwrap();

                    Type::Function(Box::new(FuncType {
                        this_ty: func.this_ty.clone(),
```

```
            params: func.params.iter().map(|p| p.ty.clone()).collect(),
            var_arg: false,
            return_ty: func.return_ty.clone(),
        }))
    } else {
        return Err(Error::syntax_error(
            q.span,
            "cannot infer type, generic function has no concrete type",
        ));
    }
}
Some(Binding::Function(f)) => {
    let func = self.context.functions.get(&f).unwrap();

    Type::Function(Box::new(FuncType {
        this_ty: func.this_ty.clone(),
        params: func.params.iter().map(|p| p.ty.clone()).collect(),
        var_arg: false,
        return_ty: func.return_ty.clone(),
    }))
}
Some(Binding::GenericTypeAlias { .. }) | Some(Binding::Generic(_)) => {
    return Err(Error::syntax_error(
        q.span,
        "cannot infer type, generic is not a value",
    ))
}
Some(Binding::GenericInterface(_)) | Some(Binding::Interface(_)) => {
    return Err(Error::syntax_error(
        q.span,
        "cannot infer type, interface is not a value",
    ))
```

```
        }
        Some(Binding::NameSpace(_)) => {
            return Err(Error::syntax_error(
                q.span,
                "cannot infer type, namespace is not a value",
            ))
        }
        Some(Binding::TypeAlias(_)) => {
            return Err(Error::syntax_error(
                q.span,
                "cannot infer type, type alias is not a value",
            ))
        }
        Some(Binding::Using { ty, .. }) => ty.clone(),
        Some(Binding::Var { ty, .. }) => ty.clone(),
        None => {
            return Err(Error::syntax_error(name.span(), "undefined identifier"))
        }
    },
    swc::TsTypeQueryExpr::Import(_) => {
        todo!("import type")
    }
};

// should not have type arguments
if !allow_generics {
    if let Some(args) = &q.type_args {
        if args.params.len() != 0 {
            return Err(Error::syntax_error(
                args.span,
                "expected zero type arguments",
            ));
```

```
            }
        }
    }

    return Ok(ty);
}
swc::TsType::TsTypeRef(r) => {
    let binding = match self.find_binding(&r.type_name) {
        Some(b) => b,
        None => {
            return Err(Error::syntax_error(
                r.type_name.span(),
                "undefined identifier",
            ))
        }
    };

    let mut generics_allowed = false;

    let ty = match binding {
        Binding::Class(c) => Type::Object(c),
        Binding::Enum(e) => Type::Enum(e),
        Binding::Interface(i) => Type::Interface(i),
        Binding::TypeAlias(id) => {
            if let Some(ty) = self.context.alias.get(&id) {
                ty.clone()
            } else {
                Type::Alias(id)
            }
        }
        Binding::GenericFunction(_) | Binding::Function(_) => {
            return Err(Error::syntax_error(
```

```
        r.type_name.span(),
        "expected type, found function",
    ))
}
Binding::Generic(_) => todo!("generics"),
Binding::GenericClass(id) => {
    generics_allowed = true;

    if let Some(type_args) = &r.type_params {
        let type_args = self.translate_type_args(&type_args)?;
        let id = self.solve_generic_class(id, &type_args)?;
        Type::Object(id)
    } else {
        return Err(Error::syntax_error(r.span, "missing type arguments"));
    }
}

Binding::GenericInterface(id) => {
    generics_allowed = true;

    if let Some(type_args) = &r.type_params {
        let type_args = self.translate_type_args(&type_args)?;
        let id = self.solve_generic_interface(id, &type_args)?;
        Type::Interface(id)
    } else {
        return Err(Error::syntax_error(r.span, "missing type arguments"));
    }
}
Binding::GenericTypeAlias(id) => {
    generics_allowed = true;

    if let Some(type_args) = &r.type_params {
```

```
            let type_args = self.translate_type_args(&type_args)?;
            let id = self.solve_generic_alias(id, &type_args)?;

            if let Some(ty) = self.context.alias.get(&id) {
                ty.clone()
            } else {
                Type::Alias(id)
            }
        } else {
            return Err(Error::syntax_error(r.span, "missing type arguments"));
        }
    }
    Binding::NameSpace(_) => {
        return Err(Error::syntax_error(
            r.type_name.span(),
            "expected type, found name space",
        ))
    }
    Binding::Using { .. } | Binding::Var { .. } => {
        return Err(Error::syntax_error(
            r.type_name.span(),
            "expected type, found variable",
        ))
    }
};

if !generics_allowed {
    if let Some(args) = &r.type_params {
        if args.params.len() != 0 {
            return Err(Error::syntax_error(
                args.span,
                "expected 0 type arguments",
```

```
                    ));
                }
            }
        }

        return Ok(ty);
    }
    swc::TsType::TsUnionOrIntersectionType(u) => match u {
        swc::TsUnionOrIntersectionType::TsIntersectionType(i) => {
            return self.translate_intersection_type(i)
        }
        swc::TsUnionOrIntersectionType::TsUnionType(u) => {
            let mut tys = Vec::new();

            for ty in &u.types {
                let ty = self.translate_type(&ty)?;

                if !tys.contains(&ty) {
                    tys.push(ty);
                }
            }

            tys.sort();

            return Ok(Type::Union(tys.into_boxed_slice()));
        }
    },
  }
}

  pub fn translate_func_type(&mut self, func: &swc::TsFnOrConstructorType) ->
Result<FuncType> {
```

```
match func {
    swc::TsFnOrConstructorType::TsConstructorType(cons) => {
        return Err(Error::syntax_error(
            cons.span,
            "constructor type not allowed",
        ))
    }
    swc::TsFnOrConstructorType::TsFnType(func) => {
        if func.type_params.is_some() {
            return Err(Error::syntax_error(
                func.span,
                "fucntion type cannot be generic",
            ));
        }

        // this type default to any
        let mut this_ty = Type::Any;
        let mut params = Vec::new();

        // translate params
        for (i, p) in func.params.iter().enumerate() {
            if let swc::TsFnParam::Ident(id) = p {
                if id.type_ann.is_none() {
                    return Err(Error::syntax_error(id.span, "missing type annotation"));
                }
                let mut ty =
                    self.translate_type(&id.type_ann.as_ref().unwrap().type_ann)?;

                // optional type
                if id.optional {
                    ty = ty.union(Type::Undefined);
                }
```

```
                // explicit this type
                if id.sym.as_ref() == "this" && i == 0 {
                    this_ty = ty;
                    continue;
                }

                params.push(ty);
            } else {
                // function type should not be destructive
                return Err(Error::syntax_error(
                    p.span(),
                    "destructive params not allowed",
                ));
            }
        }

        // return type
        let return_ty = self.translate_type(&func.type_ann.type_ann)?;

        return Ok(FuncType {
            this_ty,
            params: params,
            var_arg: false,
            return_ty,
        });
    }
};
}

pub fn translate_intersection_type(
    &mut self,
```

```
        intersec: &swc::TsIntersectionType,
    ) -> Result<Type> {
        let mut iface = InterfaceType {
            name: self.anonymous_name(),
            extends: Vec::new(),
            implements: Vec::new(),
            properties: Default::default(),
            methods: Default::default(),
        };

        for ty in &intersec.types {
            let ty = self.translate_type(&ty)?;

            match ty{
                Type::Interface(id) => {
                    if !iface.implements.contains(&id){
                        iface.implements.push(id);
                    }
                }
                Type::Object(id) => {
                    // only push if not already exist
                    if !iface.extends.contains(&id){
                        iface.extends.push(id);
                    }
                }
                _ => {
                    return Err(Error::syntax_error(intersec.span, "An intersection cannot extend a primitive type; only interfaces and classes can intersect"))
                }
            };
        }
```

```
    let id = InterfaceId::new();

    self.context.interfaces.insert(id, iface);

    return Ok(Type::Interface(id));
}

fn find_binding(&mut self, entity_name: &swc::TsEntityName) -> Option<Binding> {
    match entity_name {
        swc::TsEntityName::Ident(id) => return self.context.find(&id.sym).map(|b| b.clone()),
        swc::TsEntityName::TsQualifiedName(q) => {
            let left = self.find_binding(&q.left)?;

            match left {
                Binding::NameSpace(namespace) => {
                    return self.find_binding_from_module(
                        namespace,
                        &PropName::Ident(q.right.sym.to_string()),
                    );
                }
                _ => return None,
            }
        }
    }
}

pub fn translate_type_args(
    &mut self,
    args: &swc::TsTypeParamInstantiation,
) -> Result<Vec<Type>> {
    let mut v = Vec::with_capacity(args.params.len());
```

```
    for ty in &args.params {
        v.push(self.translate_type(&ty)?);
    }

    return Ok(v);
}

pub fn solve_generic_function(
    &mut self,
    _id: FunctionId,
    _type_args: &[Type],
) -> Result<FunctionId> {
    todo!("generic function")
}

pub fn solve_generic_class(&mut self, _id: ClassId, _type_args: &[Type]) -> Result<ClassId> {
    todo!("generic class")
}

pub fn solve_generic_interface(
    &mut self,
    _id: InterfaceId,
    _type_args: &[Type],
) -> Result<InterfaceId> {
    todo!("generic class")
}

pub fn solve_generic_alias(&mut self, _id: AliasId, _type_args: &[Type]) -> Result<AliasId> {
    todo!("generic class")
}

// returns the iterator type, iterator result type and the value type
```

```
    pub fn type_is_iterable(&self, span: Span, iterable_ty: &Type) -> Result<(Type, Type, Type)> {
        let iterator_result_ty: Type;
        let value_ty: Type;

        let iterator_func_ty = match self.type_has_property(
            &iterable_ty,
            &PropName::Symbol(crate::Symbol::Iterator),
            true,
        ) {
            Some(ty) => ty,
            None => {
                return Err(Error::syntax_error(
                    span,
                    format!("type " is not iterable, missing property [Symbol.iterator]"),
                ))
            }
        };

        let iterator_ty = match iterator_func_ty {
            Type::Function(func) => {
                // param must be empty
                if !func.params.is_empty() {
                    return Err(Error::syntax_error(span, format!("type " is not iterable, property [Symbol.iterator] is expected to have 0 arguments")));
                }
                // check this type matches iterable
                match self.type_check(span, &iterable_ty, &func.this_ty){
                    Ok(_) => {},
                    Err(_) => {
                        return Err(Error::syntax_error(span, format!("type " is not iterable, property [Symbol.iterator] has mismatched 'this' argument: type " is not assignable to "")))
                    }
```

```
            };
            // the return type is the iterator type
            func.return_ty
        }
        _ => {
            return Err(Error::syntax_error(
                span,
                format!("type " is not iterable, property [Symbol.iterator] is not callable"),
            ))
        }
    };

    // check iterator have next() method
    match self.type_has_property(&iterator_ty, &PropName::Ident("next".to_string()), true) {
        Some(next_func_ty) => {
            // check next is a function
            iterator_result_ty = match next_func_ty{
                Type::Function(func) => {
                    // param must be empty
                    if !func.params.is_empty(){
                        return Err(Error::syntax_error(span, format!("type " is not iterable, property [Symbol.iterator]().next is expected to have 0 arguments")))
                    }
                    // 'this' type must be equal to iterator
                    match self.type_check(span, &iterator_ty, &func.this_ty){
                        Ok(_) => {},
                        Err(_) => {
                            return Err(Error::syntax_error(span, format!("type " is not iterable, property [Symbol.iterator]().next has mismatched 'this' argument: type " is not assignable to "")))
                        }
                    };
                    // the return type is iterator result
```

```
                    func.return_ty
                }
                // next is not a function
                _ => return Err(Error::syntax_error(span, format!("type " is not iterable, property [Symbol.iterator]().next is not callable")))
            };

            if self
                .type_has_property(
                    &iterator_result_ty,
                    &PropName::Ident("done".to_string()),
                    false,
                )
                .is_none()
            {
                return Err(Error::syntax_error(span, format!("type " is not iterable, missing property [Symbol.iterator]().next().done")));
            }
            if let Some(value) = self.type_has_property(
                &iterator_result_ty,
                &PropName::Ident("value".to_string()),
                false,
            ) {
                value_ty = value;
            } else {
                return Err(Error::syntax_error(span, format!("type " is not iterable, missing property [Symbol.iterator]().next().value")));
            }
        }
        // no property next
        None => {
            return Err(Error::syntax_error(
                span,
```

```
                format!("type " is not iterable, missing property [Symbol.iterator]().next"),
            ))
        }
    };

    return Ok((iterator_ty, iterator_result_ty, value_ty));
}

pub fn type_check(&self, span: Span, ty: &Type, fulfills: &Type) -> Result<()> {
    // fast return
    if ty == fulfills {
        return Ok(());
    }

    if let Type::Union(u) = ty {
        if u.iter().all(|t| self.type_check(span, t, fulfills).is_ok()) {
            return Ok(());
        }
    }

    match fulfills {
        // every type can be converted to any and bool
        Type::Bool | Type::Any => return Ok(()),
        // alias is only used when hoisting
        Type::Alias(_) => unreachable!("unresolved alias"),
        Type::AnyObject => {
            if ty.is_object() {
                return Ok(());
            }
        }
        Type::Array(array_elem) => {
            // a tuple may be converted to array
```

```
    if let Type::Tuple(t) = ty {
        // check if all element of tuple is convertable to element of array
        if t.iter()
            .all(|e| self.type_check(span, e, &array_elem).is_ok())
        {
            return Ok(());
        }
    }
}

// these types must be strictly obayed
Type::LiteralObject(_)
| Type::Enum(_)
| Type::Function(_)
| Type::Generic(_)
| Type::Map(_, _)
| Type::Null
| Type::Promise(_)
| Type::Regex
| Type::LiteralBool(_)
| Type::LiteralNumber(_)
| Type::LiteralInt(_)
| Type::LiteralBigint(_)
| Type::LiteralString(_)
| Type::Symbol
| Type::Tuple(_)
| Type::Undefined => {}
// bigint
Type::Bigint => {
    if let Type::LiteralBigint(_) = ty {
        return Ok(());
    }
```

```
}
// string
Type::String => {
    if let Type::LiteralString(_) = ty {
        return Ok(());
    }
}
// number and int are compatable
Type::Number | Type::Int => {
    if ty == &Type::Number || ty == &Type::Int {
        return Ok(());
    }
    if let Type::LiteralInt(_) = ty {
        return Ok(());
    }
    if let Type::LiteralNumber(_) = ty {
        return Ok(());
    }
}
// a union requirement just have to fulill a single alternative
Type::Union(u) => {
    // if ty is a union, all elements must respect fulfills
    if let Type::Union(u) = ty {
        for t in u.iter() {
            self.type_check(span, t, fulfills)?;
        }
    } else {
        // if ty fulfills any one of union, it is true
        if u.iter().any(|t| self.type_check(span, ty, t).is_ok()) {
            return Ok(());
        }
    }
```

```
}
Type::Interface(iface) => {
    let iface = self
        .context
        .interfaces
        .get(iface)
        .expect("invalid interface");

    for im in &iface.implements {
        self.type_check(span, ty, &Type::Interface(*im))?;
    }

    for ex in &iface.extends {
        self.type_check(span, ty, &Type::Object(*ex))?;
    }

    for (name, attr) in &iface.properties {
        if let Some(attr_ty) = self.type_has_property(ty, name, false) {
            self.type_check(span, &attr_ty, &attr.ty)?;
        } else {
            if attr.optional {
                continue;
            }
            return Err(Error::syntax_error(
                span,
                format!(
                    "Property '{}' is missing in type " but required in type "",
                    name
                ),
            ));
        }
    }
```

```
for (name, method) in &iface.methods {
    if let Some(attr) = self.type_has_property(ty, name, true) {
        if let Type::Function(func) = &attr {
            // this of function must be equal to ty
            if &func.this_ty != ty {
                return Err(Error::syntax_error(
                    span,
                    format!(
                        "Method '{}' is missing in type " but required in type "",
                        name
                    ),
                ));
            }
            if &func.params != &method.params {
                return Err(Error::syntax_error(
                    span,
                    format!("Method '{}' have incompatable arguments", name),
                ));
            }
            if &func.return_ty != &method.return_ty {
                return Err(Error::syntax_error(
                    span,
                    format!("Method '{}' have incompatable return types", name),
                ));
            }
        }
    } else {
        if method.optional {
            continue;
        }
        return Err(Error::syntax_error(
```

```
                span,
                format!(
                    "Method '{}' is missing in type " but required in type "",
                    name
                ),
            ));
        }
    }
    return Ok(());
}
Type::Object(class_id) => {
    if let Type::Object(obj_ty_id) = ty {
        // fast return
        if obj_ty_id == class_id {
            return Ok(());
        }

        let obj_class = self.context.classes.get(obj_ty_id).expect("invalid class");

        // check the super type
        if let Some(super_class_id) = obj_class.extends {
            // super type must fulfil requirement
            return self.type_check(span, &Type::Object(super_class_id), fulfills);
        }
    }
}
Type::Iterator(iter_elem) => {
    if let Some(next_ty) =
        self.type_has_property(ty, &PropName::Ident("next".to_owned()), true)
    {
        if let Type::Function(func) = &next_ty {
            if func.params.len() == 0 && iter_elem.as_ref() == &func.return_ty {
```

```
                    return Ok(());
                }
            }
        }
    }
    return Err(Error::syntax_error(
        span,
        format!("type " is not assignable to type ""),
    ));
}

pub fn type_has_property(&self, ty: &Type, prop: &PropName, method: bool) -> Option<Type>
{
    if let PropName::Ident(ident) = prop {
        if ident == "toString" {
            return Some(Type::Function(Box::new(FuncType {
                this_ty: Type::Any,
                params: Vec::new(),
                var_arg: false,
                return_ty: Type::String,
            })));
        }
    }
    match ty {
        Type::Alias(_) => unreachable!(),
        Type::Any => None,
        Type::AnyObject => None,
        Type::Iterator(_) => {
            match prop {
                PropName::Ident(ident) => match ident.as_str() {
                    "next" => todo!(),
```

```
            _ => {}
        },
        _ => {}
    };
    return None;
}
Type::LiteralObject(obj) => {
    for (p, ty) in obj.iter() {
        if p == prop {
            return Some(ty.clone());
        }
    }
    return None;
}
Type::Object(class) => {
    let class = self.context.classes.get(class).expect("invalid class");
    if method {
        if let Some((_id, func_ty)) = class.methods.get(prop) {
            return Some(Type::Function(Box::new(func_ty.clone())));
        }
    }
    if let Some(attr) = class.properties.get(prop) {
        return Some(attr.ty.clone());
    }
    if let Some((_id, func_ty)) = class.methods.get(prop) {
        return Some(Type::Function(Box::new(func_ty.clone())));
    }
    return None;
}
Type::Interface(id) => {
    let iface = self.context.interfaces.get(id).expect("invalid interface");
    if method {
```

```
            if let Some(m) = iface.methods.get(prop) {
                return Some(Type::Function(Box::new(FuncType {
                    this_ty: Type::Interface(*id),
                    params: m.params.clone(),
                    var_arg: false,
                    return_ty: m.return_ty.clone(),
                })));
            }
        }
        if let Some(attr) = iface.properties.get(prop) {
            return Some(attr.ty.clone());
        }
        if let Some(m) = iface.methods.get(prop) {
            return Some(Type::Function(Box::new(FuncType {
                this_ty: Type::Interface(*id),
                params: m.params.clone(),
                var_arg: false,
                return_ty: m.return_ty.clone(),
            })));
        }
        return None;
    }
    Type::Tuple(elems) => match prop {
        PropName::Int(index) => {
            if *index >= elems.len() as i32 {
                return None;
            }
            if *index < 0 {
                return None;
            }
            return Some(elems[*index as usize].clone());
        }
```

```
    PropName::Ident(ident) => match ident.as_str() {
        "length" => Some(Type::Int),
        _ => None,
    },
    _ => None,
},
Type::Array(elem) => {
    match prop {
        PropName::Int(_) => Some(elem.as_ref().clone()),
        PropName::Private(_) | PropName::String(_) => None,
        PropName::Ident(ident) => {
            match ident.as_str() {
                "length" => Some(Type::Int),
                "at" => Some(Type::Function(Box::new(FuncType {
                    this_ty: Type::Array(elem.clone()),
                    params: vec![Type::Int.union(Type::Undefined)],
                    var_arg: false,
                    return_ty: Type::Undefined.union(elem.as_ref().clone()),
                }))),
                "concat" => Some(Type::Function(Box::new(FuncType {
                    this_ty: Type::Array(elem.clone()),
                    params: vec![Type::Array(elem.clone())],
                    var_arg: true,
                    return_ty: Type::Array(elem.clone()),
                }))),
                "copyWithin" => Some(Type::Function(Box::new(FuncType {
                    this_ty: Type::Array(elem.clone()),
                    params: vec![
                        Type::Int,
                        Type::Int,
                        Type::Int.union(Type::Undefined),
                    ],
```

```
    var_arg: false,
    return_ty: Type::Array(elem.clone()),
})),
"entries" => Some(Type::Function(Box::new(FuncType {
    this_ty: Type::Array(elem.clone()),
    params: vec![],
    var_arg: false,
    return_ty: Type::Iterator(Box::new(Type::Tuple(Box::new([
        Type::Int,
        elem.as_ref().clone(),
    ])))),
}))),
"every" => Some(Type::Function(Box::new(FuncType {
    this_ty: Type::Array(elem.clone()),
    params: vec![
        Type::Function(Box::new(FuncType {
            this_ty: Type::Any,
            params: vec![
                // the element
                elem.as_ref().clone(),
                // index
                Type::Int,
                // this array
                Type::Array(elem.clone()),
            ],
            var_arg: false,
            return_ty: Type::Bool,
        })),
        Type::Any,
    ],
    var_arg: false,
    return_ty: Type::Bool,
```

```
    }))),
    "fill" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![
            elem.as_ref().clone(),
            Type::Int.union(Type::Undefined),
            Type::Int.union(Type::Undefined),
        ],
        var_arg: false,
        return_ty: Type::Array(elem.clone()),
    }))),
    "filter" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![
            // callback
            Type::Function(Box::new(FuncType {
                this_ty: Type::Any,
                params: vec![
                    // current element
                    elem.as_ref().clone(),
                    // index
                    Type::Int,
                    // this array
                    Type::Array(elem.clone()),
                ],
                var_arg: false,
                return_ty: Type::Bool,
            })),
            // this arg
            Type::Any,
        ],
        var_arg: false,
```

```
        return_ty: Type::Array(elem.clone()),
    }))),
    "find" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![
            // callback
            Type::Function(Box::new(FuncType {
                this_ty: Type::Any,
                params: vec![
                    // current element
                    elem.as_ref().clone(),
                    // index
                    Type::Int,
                    // this array
                    Type::Array(elem.clone()),
                ],
                var_arg: false,
                return_ty: Type::Bool,
            })),
            // this arg
            Type::Any,
        ],
        var_arg: false,
        return_ty: elem.as_ref().clone().union(Type::Undefined),
    }))),
    "findIndex" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![
            // callback
            Type::Function(Box::new(FuncType {
                this_ty: Type::Any,
                params: vec![
```

```
                // current element
                elem.as_ref().clone(),
                // index
                Type::Int,
                // this array
                Type::Array(elem.clone()),
            ],
            var_arg: false,
            return_ty: Type::Bool,
        })),
        // this arg
        Type::Any,
    ],
    var_arg: false,
    return_ty: Type::Int,
}))),
"findLast" => Some(Type::Function(Box::new(FuncType {
    this_ty: Type::Array(elem.clone()),
    params: vec![
        // callback
        Type::Function(Box::new(FuncType {
            this_ty: Type::Any,
            params: vec![
                // current element
                elem.as_ref().clone(),
                // index
                Type::Int,
                // this array
                Type::Array(elem.clone()),
            ],
            var_arg: false,
            return_ty: Type::Bool,
```

```
            })),
            // this arg
            Type::Any,
        ],
        var_arg: false,
        return_ty: elem.as_ref().clone().union(Type::Undefined),
    }))),
    "findLastIndex" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![
            // callback
            Type::Function(Box::new(FuncType {
                this_ty: Type::Any,
                params: vec![
                    // current element
                    elem.as_ref().clone(),
                    // index
                    Type::Int,
                    // this array
                    Type::Array(elem.clone()),
                ],
                var_arg: false,
                return_ty: Type::Bool,
            })),
            // this arg
            Type::Any,
        ],
        var_arg: false,
        return_ty: Type::Int,
    }))),
    "flat" => todo!(),
    "flatMap" => Some(Type::Function(Box::new(FuncType {
```

```
        this_ty: Type::Array(elem.clone()),
        params: vec![
            // callback
            Type::Function(Box::new(FuncType {
                this_ty: Type::Any,
                params: vec![
                    // current element
                    elem.as_ref().clone(),
                    // index
                    Type::Int,
                    // this array
                    Type::Array(elem.clone()),
                ],
                var_arg: false,
                return_ty: Type::Array(elem.clone()),
            })),
        ],
        var_arg: false,
        return_ty: Type::Array(elem.clone()),
    }))),
    "forEach" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![
            // callback
            Type::Function(Box::new(FuncType {
                this_ty: Type::Any,
                params: vec![
                    // current element
                    elem.as_ref().clone(),
                    // index
                    Type::Int,
                    // this array
```

```
                Type::Array(elem.clone()),
            ],
            var_arg: false,
            return_ty: Type::Undefined,
        })),
        // this arg
        Type::Any,
    ],
    var_arg: false,
    return_ty: Type::Undefined,
}))),
"includes" => Some(Type::Function(Box::new(FuncType {
    this_ty: Type::Array(elem.clone()),
    params: vec![
        elem.as_ref().clone(),
        Type::Int.union(Type::Undefined),
    ],
    var_arg: false,
    return_ty: Type::Bool,
}))),
"indexOf" => Some(Type::Function(Box::new(FuncType {
    this_ty: Type::Array(elem.clone()),
    params: vec![
        elem.as_ref().clone(),
        Type::Int.union(Type::Undefined),
    ],
    var_arg: false,
    return_ty: Type::Int,
}))),
"lastIndexOf" => Some(Type::Function(Box::new(FuncType {
    this_ty: Type::Array(elem.clone()),
    params: vec![
```

```
            elem.as_ref().clone(),
            Type::Int.union(Type::Undefined),
        ],
        var_arg: false,
        return_ty: Type::Int,
    }))),
    "join" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![Type::String.union(Type::Undefined)],
        var_arg: false,
        return_ty: Type::String,
    }))),
    "keys" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![],
        var_arg: false,
        return_ty: Type::Iterator(Box::new(Type::Int)),
    }))),
    "map" => todo!(),
    "pop" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![],
        var_arg: false,
        return_ty: elem.as_ref().clone().union(Type::Undefined),
    }))),
    "push" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![elem.as_ref().clone()],
        var_arg: true,
        return_ty: Type::Int,
    }))),
    "reduce" => Some(Type::Function(Box::new(FuncType {
```

```
    this_ty: Type::Array(elem.clone()),
    params: vec![
        Type::Function(Box::new(FuncType {
            this_ty: Type::Any,
            params: vec![
                elem.as_ref().clone(),
                elem.as_ref().clone(),
                Type::Int,
            ],
            var_arg: false,
            return_ty: elem.as_ref().clone(),
        })),
        elem.as_ref().clone().union(Type::Undefined),
    ],
    var_arg: false,
    return_ty: elem.as_ref().clone(),
}))),
"reduceRight" => Some(Type::Function(Box::new(FuncType {
    this_ty: Type::Array(elem.clone()),
    params: vec![
        Type::Function(Box::new(FuncType {
            this_ty: Type::Any,
            params: vec![
                elem.as_ref().clone(),
                elem.as_ref().clone(),
                Type::Int,
            ],
            var_arg: false,
            return_ty: elem.as_ref().clone(),
        })),
        elem.as_ref().clone().union(Type::Undefined),
    ],
```

```
        var_arg: false,
        return_ty: elem.as_ref().clone(),
    }))),
    "reverse" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![],
        var_arg: false,
        return_ty: Type::Undefined,
    }))),
    "shift" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![],
        var_arg: false,
        return_ty: elem.as_ref().clone().union(Type::Undefined),
    }))),
    "slice" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![
            Type::Int.union(Type::Undefined),
            Type::Int.union(Type::Undefined),
        ],
        var_arg: false,
        return_ty: Type::Array(elem.clone()),
    }))),
    "some" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![
            // callback
            Type::Function(Box::new(FuncType {
                this_ty: Type::Any,
                params: vec![
                    // current element
```

```
                elem.as_ref().clone(),
                // index
                Type::Int,
                // this array
                Type::Array(elem.clone()),
            ],
            var_arg: false,
            return_ty: Type::Bool,
        })),
        // this arg
        Type::Any,
    ],
    var_arg: false,
    return_ty: Type::Bool,
}))),
"sort" => Some(Type::Function(Box::new(FuncType {
    this_ty: Type::Array(elem.clone()),
    params: vec![Type::Function(Box::new(FuncType {
        this_ty: Type::Any,
        params: vec![elem.as_ref().clone(), elem.as_ref().clone()],
        var_arg: false,
        return_ty: Type::Int,
    }))],
    var_arg: false,
    return_ty: Type::Array(elem.clone()),
}))),
"splice" => Some(Type::Function(Box::new(FuncType {
    this_ty: Type::Array(elem.clone()),
    params: vec![
        // start
        Type::Int,
        // delete count
```

```
            Type::Int.union(Type::Undefined),
            // elements
            elem.as_ref().clone(),
        ],
        var_arg: true,
        return_ty: Type::Array(elem.clone()),
    }))),
    "toReverse" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![],
        var_arg: false,
        return_ty: Type::Array(elem.clone()),
    }))),
    "toSorted" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![Type::Function(Box::new(FuncType {
            this_ty: Type::Any,
            params: vec![elem.as_ref().clone(), elem.as_ref().clone()],
            var_arg: false,
            return_ty: Type::Int,
        }))],
        var_arg: false,
        return_ty: Type::Array(elem.clone()),
    }))),
    "toSplice" => Some(Type::Function(Box::new(FuncType {
        this_ty: Type::Array(elem.clone()),
        params: vec![
            // start
            Type::Int,
            // delete count
            Type::Int.union(Type::Undefined),
            // elements
```

```
                elem.as_ref().clone(),
            ],
            var_arg: true,
            return_ty: Type::Array(elem.clone()),
        }))),
        "unshift" => Some(Type::Function(Box::new(FuncType {
            this_ty: Type::Array(elem.clone()),
            params: vec![elem.as_ref().clone()],
            var_arg: true,
            return_ty: Type::Int,
        }))),
        "values" => Some(Type::Function(Box::new(FuncType {
            this_ty: Type::Array(elem.clone()),
            params: vec![],
            var_arg: false,
            return_ty: Type::Iterator(elem.clone()),
        }))),
        "with" => Some(Type::Function(Box::new(FuncType {
            this_ty: Type::Array(elem.clone()),
            params: vec![Type::Int, elem.as_ref().clone()],
            var_arg: false,
            return_ty: Type::Array(elem.clone()),
        }))),
        _ => None,
      }
    }
    PropName::Symbol(sym) => match sym {
      Symbol::Iterator => Some(Type::Iterator(elem.clone())),
      Symbol::Unscopables => todo!(),
      _ => None,
    },
  }
```

```
        }
        _ => todo!(),
      }
    }
}
```

5.2.11 native-ts-hir/transform/stmt.rs

```
use native_js_common::error::Error;
use native_ts_parser::swc_core::common::{Span, Spanned};
use native_ts_parser::swc_core::ecma::ast as swc;

use crate::{
ast::{Callee, Expr, PropNameOrExpr, Stmt, Type},
common::VariableId,
PropName,
};

use super::{context::Binding, Transformer};

type Result<T> = std::result::Result<T, Error<Span>>;

impl Transformer {
/// translates a statement
pub fn translate_stmt(&mut self, stmt: &swc::Stmt, label: Option<&str>) ->
Result<()> {
match stmt {
swc::Stmt::Block(b) => self.translate_block_stmt(b, label)?,
swc::Stmt::Break(b) => {
// check for label
if let Some(label) = &b.label {
if !self.break_labels.contains(label.sym.as_ref()) {
return Err(Error::syntax_error(label.span, "undefined label"));
}
}
// push break stmt
self.context
.func()
.stmts
.push(Stmt::Break(label.map(|l| l.to_string())));
}
swc::Stmt::Continue(c) => {
```

```
// check for label
if let Some(label) = &c.label {
if !self.continue_labels.contains(label.sym.as_ref()) {
return Err(Error::syntax_error(label.span, "undefined label"));
}
}
// push continue stmt
self.context
.func()
.stmts
.push(Stmt::Continue(label.map(|l| l.to_string())))
}
swc::Stmt::Debugger(d) => {
return Err(Error::syntax_error(
d.span,
"debugger statement not allowed",
))
}
swc::Stmt::Decl(d) => self.translate_decl(d)?,
swc::Stmt::DoWhile(d) => self.translate_do_while(d, label)?,
swc::Stmt::Empty(_) => {
// does nothing
}
swc::Stmt::Expr(e) => {
// translate the expression
let (expr, _ty) = self.translate_expr(&e.expr, None)?;
self.context.func().stmts.push(Stmt::Expr(Box::new(expr)));
}
swc::Stmt::For(f) => self.translate_for_stmt(f, label)?,
swc::Stmt::ForIn(f) => self.translate_for_in_stmt(f, label)?,
swc::Stmt::ForOf(f) => self.translate_for_of_stmt(f, label)?,
swc::Stmt::If(i) => self.translate_if_stmt(i)?,
swc::Stmt::Labeled(l) => self.translate_stmt(&l.body, Some(&l.label.sym))?,
swc::Stmt::Return(r) => {
// translate argument with expected return type
let arg = if let Some(e) = &r.arg {
self.translate_expr(e, Some(&self.return_ty.clone()))?.0
} else {
// check if undefined is returnable
self.type_check(r.span, &Type::Undefined, &self.return_ty.clone())?;
Expr::Undefined
};

// return
```

```
self.context.func().stmts.push(Stmt::Return(Box::new(arg)));
}
swc::Stmt::Switch(s) => self.translate_switch_stmt(s)?,
swc::Stmt::Throw(t) => {
// does not have to be type checked
let (expr, _ty) = self.translate_expr(&t.arg, None)?;
// push throw stmt
self.context.func().stmts.push(Stmt::Throw(Box::new(expr)));
}
swc::Stmt::Try(t) => self.translate_try_catch_stmt(t)?,
swc::Stmt::While(w) => self.translate_while_stmt(w, label)?,
swc::Stmt::With(w) => {
// with statement is deprecated
return Err(Error::syntax_error(w.span, "with statement is deprecated"));
}
}

return Ok(());
}

pub fn translate_block_stmt(
&mut self,
block: &swc::BlockStmt,
label: Option<&str>,
) -> Result<()> {
let mut old_break = false;
// insert the label
if let Some(label) = label {
old_break = !self.break_labels.insert(label.to_string());
// block
self.context.func().stmts.push(Stmt::Block {
label: label.to_string(),
});
}

// open a new scope
self.context.new_scope();

self.hoist_stmts(block.stmts.iter())?;

for stmt in &block.stmts {
self.translate_stmt(stmt, None)?;
}
```

```
// close the scope
self.context.end_scope();

// remove the label
if let Some(label) = label {
if !old_break {
self.break_labels.remove(label);
}
// end block
self.context.func().stmts.push(Stmt::EndBlock);
}

return Ok(());
}

pub fn translate_decl(&mut self, decl: &swc::Decl) -> Result<()> {
match decl {
swc::Decl::Class(c) => {
let id = self.context.get_class_id(&c.ident.sym);
self.translate_class(id, c.ident.sym.to_string(), &c.class)?;
self.context.func().stmts.push(Stmt::DeclareClass(id));
}
swc::Decl::Fn(f) => {
let id = self.context.get_func_id(&f.ident.sym);
self.translate_function(id, None, &f.function)?;
self.context.func().stmts.push(Stmt::DeclareFunction(id));
}
swc::Decl::TsEnum(_) => {
// does nothing
}
swc::Decl::TsInterface(_) => {
// does nothing
}
swc::Decl::TsModule(_) => {
// does nothing
}
swc::Decl::TsTypeAlias(_) => {
// does noting
}
swc::Decl::Using(u) => {
self.translate_using_decl(u)?;
}
swc::Decl::Var(decl) => {
self.translate_var_decl(decl)?;
```

```
}
}
return Ok(());
}

/// variable declaration
pub fn translate_var_decl(&mut self, decl: &swc::VarDecl) ->
Result<Vec<VariableId>> {
let mut ids = Vec::new();

for d in &decl.decls {
let init = if let Some(init) = &d.init{
Some(self.translate_expr(&init, None)?)
} else{
None
};
ids.extend_from_slice(
&self.translate_pat_var_decl(decl.kind, &d.name, init, None)?)
}
return Ok(ids);
}

fn translate_pat_var_decl(&mut self, kind: swc::VarDeclKind, pat: &swc::Pat,
init: Option<(Expr, Type)>, parent_ann: Option<(Type, Span)>) ->
Result<Vec<VariableId>>{
match pat {
swc::Pat::Ident(id) => {
Ok(vec![self.translate_ident_var_dec(
kind,
id,
init,
parent_ann
)?])
}
swc::Pat::Array(a) => {
self.translate_array_pat_decl(
kind,
a,
init,
parent_ann
)
}
swc::Pat::Object(obj) => {
self.tranlslate_object_pat_decl(
```

```
kind,
obj,
init,
parent_ann
)
}
swc::Pat::Assign(a) => {
return Err(Error::syntax_error(
a.span,
"invalid left-hand side assignment",
))
}
swc::Pat::Expr(e) => {
return Err(Error::syntax_error(
e.span(),
"invalid left-hand side assignment",
))
}
swc::Pat::Invalid(i) => {
return Err(Error::syntax_error(
i.span,
"invalid left-hand side assignment",
))
}
// todo: rest assignment
swc::Pat::Rest(r) => {
return Err(Error::syntax_error(
r.dot3_token,
"rest assignment not supportd",
))
}
}
}

fn translate_ident_var_dec(
&mut self,
kind: swc::VarDeclKind,
ident: &swc::BindingIdent,
init: Option<(Expr, Type)>,
parent_ann: Option<(Type, Span)>
) -> Result<VariableId> {
let varid = VariableId::new();

let mut ty = None;
```

```
let mut init_expr = None;

if let Some(ann) = &ident.type_ann {
ty = Some(self.translate_type(&ann.type_ann)?);
} else{
if let Some((ann, _)) = parent_ann{
ty = Some(ann);
}
}

if let Some((mut init, init_ty)) = init {
match init_ty {
Type::LiteralBigint(_) => {
ty = Some(Type::Bigint);
}
Type::LiteralBool(_) => ty = Some(Type::Bool),
Type::LiteralInt(_) | Type::Int => {
ty = Some(Type::Number);
init = Expr::Cast(Box::new(init), Type::Number);
}
Type::LiteralNumber(_) => ty = Some(Type::Number),
Type::LiteralString(_) => ty = Some(Type::String),
_ => ty = Some(init_ty),
}

init_expr = Some(init);
}

if ty.is_none() {
return Err(Error::syntax_error(ident.span, "missing type annotation"));
}

// declare variable
if !self.context.declare(
&ident.sym,
super::context::Binding::Var {
writable: kind != swc::VarDeclKind::Const,
redeclarable: kind == swc::VarDeclKind::Var,
id: varid,
ty: ty.as_ref().unwrap().clone(),
},
) {
// the identifier is already declared
return Err(Error::syntax_error(ident.span, "duplicated identifier"));
```

```
}

self.context
.func()
.stmts
.push(Stmt::DeclareVar(varid, ty.unwrap()));

if let Some(init) = init_expr {
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: varid,
value: Box::new(init),
})));
}

return Ok(varid);
}

fn tranlslate_object_pat_decl(
&mut self,
kind: swc::VarDeclKind,
obj: &swc::ObjectPat,
init: Option<(Expr, Type)>,
// annotation given by parent pattern
parent_ann: Option<(Type, Span)>,
) -> Result<Vec<VariableId>> {
if obj.optional {
return Err(Error::syntax_error(obj.span, "object pattern cannot be optional"))
}

todo!()
}

fn translate_array_pat_decl(
&mut self,
kind: swc::VarDeclKind,
pat: &swc::ArrayPat,
init: Option<(Expr, Type)>,
// annotation given by parent pattern
parent_ann: Option<(Type, Span)>,
) -> Result<Vec<VariableId>> {
// variable ids declared
let mut ids = Vec::new();
// todo: optional assignment
```

```
if pat.optional{
return Err(Error::syntax_error(pat.span, "array pattern cannot be optional"))
}
// translate type annotation
let (ty_ann, ty_ann_span) = if let Some(ann) = pat.type_ann.as_ref() {
// type annotation is already given
if parent_ann.is_some() {
// a duplicated annotation happened
return Err(Error::syntax_error(ann.span, "conflicted type annotation"));
}
// translate type annotation
let ty = self.translate_type(&ann.type_ann)?;
// only accepts array or tuple
match &ty {
Type::Array(_) => {
// do nothing
}
Type::Tuple(t) => {
// check tuple length
if t.len() != pat.elems.len() {
return Err(Error::syntax_error(
ann.span,
"number of elements in tuple does not match that of pattern",
));
}
}
// not tuple or array
_ => {
return Err(Error::syntax_error(
ann.span,
"expected array or tuple type",
))
}
}
// return type
(Some(ty), Some(ann.span))
} else {
if let Some((ann, ann_span)) = parent_ann {
// only accepts array or tuple
match &ann {
// do nothing for array type
Type::Array(_) => {}
// tuple type must have exact length
Type::Tuple(t) => {
```

```
// check length of tuple type
if t.len() != pat.elems.len() {
return Err(Error::syntax_error(
ann_span,
"number of elements in tuple does not match that of pattern",
));
}
}
// must be array or tuple
_ => {
return Err(Error::syntax_error(
ann_span,
"expected array or tuple type",
))
}
}
// return annotation
(Some(ann), Some(ann_span))
} else {
// no annotation
(None, None)
}
};

// push the initialiser to stack and get its type
let init_ty = if let Some((e, t)) = init{
// push the initialiser to stack
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::Push(Box::new(e)))));
// return type
Some(t)
} else{
// no initialiser
None
};

for i in 0..pat.elems.len(){
// get the type of element at index
let ann_prop_ty = match &ty_ann{
Some(Type::Array(a)) => Some((a.as_ref().clone(), ty_ann_span.unwrap())),
Some(Type::Tuple(t)) => Some((t[i].clone(), ty_ann_span.unwrap())),
_ => None
};

// if some, a variable is declared, other wise, skip index
```

```
if let Some(p) = &pat.elems[i]{
// an initialiser is present
if let Some(init_ty) = &init_ty{
// check if initialiser has index
if let Some(prop_ty) = self.type_has_property(init_ty, &PropName::Int(i as _),
false){
// construct member expression
let member_expr = Expr::Member {
object: Box::new(Expr::ReadStack),
key: PropNameOrExpr::PropName(PropName::Int(i as _)),
optional: false
};
// translate pat variable declare
let vs = self.translate_pat_var_decl(kind, p, Some((member_expr, prop_ty)),
ann_prop_ty)?;
// push ids
ids.extend_from_slice(&vs);

} else{
// the initialiser does not have index
return Err(Error::syntax_error(p.span(), format!("type '' has no property '{}'",
i)))
}
} else{
// no initialiser
let vs = self.translate_pat_var_decl(kind, p, None, ann_prop_ty)?;
// push ids
ids.extend_from_slice(&vs);
}
};
}
// pop the initialiser from stack
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::Pop)));

return Ok(ids)
}

pub fn translate_using_decl(&mut self, decl: &swc::UsingDecl) ->
Result<Vec<VariableId>> {
let mut ids = Vec::new();

for d in &decl.decls {
if let Some(ident) = d.name.as_ident() {
let varid = VariableId::new();
```

```
ids.push(varid);

let mut ty = None;
let mut init_expr = None;

if let Some(ann) = &ident.type_ann {
ty = Some(self.translate_type(&ann.type_ann)?);
}
if let Some(init) = &d.init {
let (init, init_ty) = self.translate_expr(&init, ty.as_ref())?;
init_expr = Some(init);

if ty.is_none() {
ty = Some(init_ty);
}
}

if ty.is_none() {
return Err(Error::syntax_error(ident.span, "missing type annotation"));
}

// declare variable
if !self.context.declare(
&ident.sym,
super::context::Binding::Using {
is_await: decl.is_await,
id: varid,
ty: ty.as_ref().unwrap().clone(),
},
) {
return Err(Error::syntax_error(ident.span, "duplicated identifier"));
}

self.context
.func()
.stmts
.push(Stmt::DeclareVar(varid, ty.as_ref().unwrap().clone()));

if let Some(init) = init_expr {
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: varid,
value: Box::new(init),
})));
```

```
}
} else {
return Err(Error::syntax_error(
d.span,
"destructive variable not supported",
));
}
}
return Ok(ids);
}

pub fn translate_do_while(&mut self, d: &swc::DoWhileStmt, label:
Option<&str>) -> Result<()> {
let mut old_break = false;
let mut old_continue = false;
if let Some(label) = label {
old_break = !self.break_labels.insert(label.to_string());
old_continue = !self.continue_labels.insert(label.to_string());
}

self.context.new_scope();
self.context.func().stmts.push(Stmt::Loop {
label: label.map(|l| l.to_string()),
});

// translate blody
self.translate_stmt(&d.body, None)?;

// break if test
let (test, _ty) = self.translate_expr(&d.test, Some(&Type::Bool))?;

let func = self.context.func();
func.stmts.push(Stmt::If { test: Box::new(test) });
func.stmts.push(Stmt::Break(label.map(|l| l.to_string())));
func.stmts.push(Stmt::EndIf);

// end scope must go before end loop
self.context.end_scope();
// end loop
self.context.func().stmts.push(Stmt::EndLoop);

if let Some(label) = label {
if !old_break {
self.break_labels.remove(label);
```

```
}
if !old_continue {
self.continue_labels.remove(label);
}
}
return Ok(());
}

pub fn translate_while_stmt(&mut self, w: &swc::WhileStmt, label:
Option<&str>) -> Result<()> {
let mut old_break = false;
let mut old_continue = false;
if let Some(label) = label {
old_break = !self.break_labels.insert(label.to_string());
old_continue = !self.continue_labels.insert(label.to_string());
}
// push loop
self.context.func().stmts.push(Stmt::Loop {
label: label.map(|l| l.to_string()),
});
// open new scope
self.context.new_scope();

// break if test
let (test, _ty) = self.translate_expr(&w.test, Some(&Type::Bool))?;

let func = self.context.func();
func.stmts.push(Stmt::If { test: Box::new(test) });
func.stmts.push(Stmt::Break(label.map(|l| l.to_string())));
func.stmts.push(Stmt::EndIf);

// end scope must go before end loop
self.context.end_scope();

// translate blody
self.translate_stmt(&w.body, None)?;

self.context.func().stmts.push(Stmt::EndLoop);

if let Some(label) = label {
if !old_break {
self.break_labels.remove(label);
}
if !old_continue {
```

```
self.continue_labels.remove(label);
}
}

return Ok(());
}

pub fn translate_for_stmt(&mut self, f: &swc::ForStmt, label: Option<&str>) ->
Result<()> {
// is a label with same name exist
let mut old_break = false;
let mut old_continue = false;
// register label
if let Some(label) = label {
old_break = !self.break_labels.insert(label.to_string());
old_continue = !self.continue_labels.insert(label.to_string());
}

// open new context
self.context.new_scope();
// initialise
if let Some(init) = &f.init {
match init {
swc::VarDeclOrExpr::Expr(e) => {
// translate expression
let (e, _ty) = self.translate_expr(e, None)?;
// push expression
self.context.func().stmts.push(Stmt::Expr(Box::new(e)));
}
swc::VarDeclOrExpr::VarDecl(decl) => {
// only hoist non var
if decl.kind != swc::VarDeclKind::Var {
self.hoist_vardecl(decl)?;
}
// translate variable declare
self.translate_var_decl(decl)?;
}
}
}
// enter loop
self.context.func().stmts.push(Stmt::Loop {
label: label.map(|s| s.to_string()),
});
```

```
// break if false
if let Some(test) = &f.test {
let (test, _ty) = self.translate_expr(&test, Some(&Type::Bool))?;
let func = self.context.func();
// break if not test
func.stmts.push(Stmt::If {
test: Box::new(Expr::Unary {
op: crate::ast::UnaryOp::LogicalNot,
value: Box::new(test),
}),
});
func.stmts.push(Stmt::Break(None));
func.stmts.push(Stmt::EndIf);
}

self.translate_stmt(&f.body, None)?;

if let Some(update) = &f.update {
let (expr, _ty) = self.translate_expr(&update, None)?;
self.context.func().stmts.push(Stmt::Expr(Box::new(expr)));
}

self.context.end_scope();
self.context.func().stmts.push(Stmt::EndLoop);

// remove label
if let Some(label) = label {
if !old_break {
self.break_labels.remove(label);
}
if !old_continue {
self.continue_labels.remove(label);
}
}
return Ok(());
}

pub fn translate_for_in_stmt(&mut self, f: &swc::ForInStmt, label:
Option<&str>) -> Result<()> {
let mut old_break = false;
let mut old_continue = false;
if let Some(label) = label {
old_break = !self.break_labels.insert(label.to_string());
old_continue = !self.continue_labels.insert(label.to_string());
```

```
}

self.context.new_scope();

let (iterable_expr, iterable_ty) = self.translate_expr(&f.right, None)?;

// stores the iterator
let iterator_var = VariableId::new();
let iterator_result_var = VariableId::new();

// counter stores index
let counter = VariableId::new();

// register counter variable
self.context.func().variables.insert(
counter,
crate::ast::VariableDesc {
ty: Type::Int,
is_heap: false,
is_captured: false,
},
);
self.context
.func()
.stmts
.push(Stmt::DeclareVar(counter, Type::Int));
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: counter,
value: Box::new(Expr::Int(0)),
})));

match &iterable_ty {
Type::Tuple(_) | Type::Array(_) => {
// register the variable
self.context.func().variables.insert(
iterator_var,
crate::ast::VariableDesc {
ty: Type::Int,
is_heap: false,
is_captured: false,
},
);
// declare variable
```

```
self.context
.func()
.stmts
.push(Stmt::DeclareVar(iterator_var, Type::Int));
// assign array.length to iterator
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: iterator_var,
value: Box::new(Expr::Member {
object: Box::new(iterable_expr),
key: PropNameOrExpr::PropName(PropName::Ident("length".to_string())),
optional: false,
}),
})));
}
_ => {
let (iterator_ty, iterator_result_ty, _iterator_value_ty) =
self.type_is_iterable(f.right.span(), &iterable_ty)?;

// register the variable
self.context.func().variables.insert(
iterator_var,
crate::ast::VariableDesc {
ty: iterator_ty.clone(),
is_heap: false,
is_captured: false,
},
);
self.context.func().variables.insert(
iterator_result_var,
crate::ast::VariableDesc {
ty: iterator_result_ty.clone(),
is_heap: false,
is_captured: false,
},
);

self.context
.func()
.stmts
.push(Stmt::DeclareVar(iterator_var, iterator_ty));
self.context
.func()
.stmts
```

```
.push(Stmt::DeclareVar(iterator_result_var, iterator_result_ty));

// iterator = iterable[Symbol.iterator]();
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: iterator_var,
value: Box::new(Expr::Call {
callee: Box::new(Callee::Member {
object: iterable_expr,
prop: PropNameOrExpr::PropName(PropName::Symbol(
crate::Symbol::Iterator,
)),
}),
args: Vec::new(),
optional: false,
})));
}
};

self.context.func().stmts.push(Stmt::Loop {
label: label.map(|l| l.to_string()),
});

// the breaking condition
match &iterable_ty {
Type::Tuple(_) | Type::Array(_) => {
// break if counter == length
self.context.func().stmts.push(Stmt::If {
test: Box::new(Expr::Bin {
op: crate::ast::BinOp::EqEqEq,
left: Box::new(Expr::VarLoad {
span: Span::default(),
variable: iterator_var,
}),
right: Box::new(Expr::VarLoad {
span: Span::default(),
variable: counter,
}),
}),
});
self.context.func().stmts.push(Stmt::Break(None));
self.context.func().stmts.push(Stmt::EndIf);
}
```

```
_ => {
// iterator_result = iterator.next();
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: iterator_result_var,
value: Box::new(Expr::Call {
callee: Box::new(Callee::Member {
object: Expr::VarLoad {
span: Span::default(),
variable: iterator_var,
},
prop: PropNameOrExpr::PropName(PropName::Ident("next".to_string())),
}),
args: Vec::new(),
optional: false,
}),
})));

// if (iterator_result.done) {
// break
// }
self.context.func().stmts.push(Stmt::If {
test: Box::new(Expr::Member {
object: Box::new(Expr::VarLoad {
span: Span::default(),
variable: iterator_result_var,
}),
key: PropNameOrExpr::PropName(PropName::Ident("done".to_string())),
optional: false,
}),
});

self.context.func().stmts.push(Stmt::Break(None));
self.context.func().stmts.push(Stmt::EndIf);
}
};

self.translate_for_head(
&f.left,
// counter++
Expr::Cast(
Box::new(Expr::VarUpdate {
op: crate::ast::UpdateOp::SuffixAdd,
variable: counter,
```

```
}),
Type::Number,
),
Type::Number,
)?;

// translate the body
self.translate_stmt(&f.body, None)?;

// end scope
self.context.end_scope();
// end loop
self.context.func().stmts.push(Stmt::EndLoop);

if let Some(label) = label {
if !old_break {
self.break_labels.remove(label);
}
if !old_continue {
self.continue_labels.remove(label);
}
}
return Ok(());
}

pub fn translate_for_of_stmt(&mut self, f: &swc::ForOfStmt, label:
Option<&str>) -> Result<()> {
let mut old_break = false;
let mut old_continue = false;
if let Some(label) = label {
old_break = !self.break_labels.insert(label.to_string());
old_continue = !self.continue_labels.insert(label.to_string());
}

self.context.new_scope();

let (iterable_expr, iterable_ty) = self.translate_expr(&f.right, None)?;

let mut iterator_value_ty = Type::Undefined;

// stores the iterator
let iterator_var = VariableId::new();
let iterator_result_var = VariableId::new();
```

```
// counter stores index
let counter = VariableId::new();

// register counter variable
self.context.func().variables.insert(
counter,
crate::ast::VariableDesc {
ty: Type::Int,
is_heap: false,
is_captured: false,
},
);
self.context
.func()
.stmts
.push(Stmt::DeclareVar(counter, Type::Int));
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: counter,
value: Box::new(Expr::Int(0)),
})));

match &iterable_ty {
Type::Tuple(_) | Type::Array(_) => {
// register the variable
self.context.func().variables.insert(
iterator_var,
crate::ast::VariableDesc {
ty: iterable_ty.clone(),
is_heap: false,
is_captured: false,
},
);
self.context.func().variables.insert(
iterator_result_var,
crate::ast::VariableDesc {
ty: Type::Int,
is_heap: false,
is_captured: false,
},
);
// declare variable
self.context
.func()
```

```
.stmts
.push(Stmt::DeclareVar(iterator_var, iterable_ty.clone()));
self.context
.func()
.stmts
.push(Stmt::DeclareVar(iterator_result_var, Type::Int));
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: iterator_var,
value: Box::new(iterable_expr),
})));
// assign array.length to iterator result
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: iterator_result_var,
value: Box::new(Expr::Member {
object: Box::new(Expr::VarLoad {
span: Span::default(),
variable: iterator_var,
}),
key: PropNameOrExpr::PropName(PropName::Ident("length".to_string())),
optional: false,
}),
})));
}
_ => {
let (iterator_ty, iterator_result_ty, value_ty) =
self.type_is_iterable(f.right.span(), &iterable_ty)?;

iterator_value_ty = value_ty;

// register the variable
self.context.func().variables.insert(
iterator_var,
crate::ast::VariableDesc {
ty: iterator_ty.clone(),
is_heap: false,
is_captured: false,
},
);
self.context.func().variables.insert(
iterator_result_var,
crate::ast::VariableDesc {
ty: iterator_result_ty.clone(),
```

```
is_heap: false,
is_captured: false,
},
);

self.context
.func()
.stmts
.push(Stmt::DeclareVar(iterator_var, iterator_ty));
self.context
.func()
.stmts
.push(Stmt::DeclareVar(iterator_result_var, iterator_result_ty));

// iterator = iterable[Symbol.iterator]();
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: iterator_var,
value: Box::new(Expr::Call {
callee: Box::new(Callee::Member {
object: iterable_expr,
prop: PropNameOrExpr::PropName(PropName::Symbol(
crate::Symbol::Iterator,
)),
}),
args: Vec::new(),
optional: false,
})),
})));
}
};

self.context.func().stmts.push(Stmt::Loop {
label: label.map(|l| l.to_string()),
});

// the breaking condition
match &iterable_ty {
Type::Tuple(_) | Type::Array(_) => {
// break if counter == length
self.context.func().stmts.push(Stmt::If {
test: Box::new(Expr::Bin {
op: crate::ast::BinOp::EqEqEq,
left: Box::new(Expr::VarLoad {
```

```
span: Span::default(),
variable: iterator_result_var,
}),
right: Box::new(Expr::VarLoad {
span: Span::default(),
variable: counter,
}),
}),
});
self.context.func().stmts.push(Stmt::Break(None));
self.context.func().stmts.push(Stmt::EndIf);
}
_ => {
// iterator_result = iterator.next();
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: iterator_result_var,
value: Box::new(Expr::Call {
callee: Box::new(Callee::Member {
object: Expr::VarLoad {
span: Span::default(),
variable: iterator_var,
},
prop: PropNameOrExpr::PropName(PropName::Ident("next".to_string())),
}),
args: Vec::new(),
optional: false,
}),
})));

// if (iterator_result.done) {
// break
// }
self.context.func().stmts.push(Stmt::If {
test: Box::new(Expr::Member {
object: Box::new(Expr::VarLoad {
span: Span::default(),
variable: iterator_result_var,
}),
key: PropNameOrExpr::PropName(PropName::Ident("done".to_string())),
optional: false,
}),
});
```

```
self.context.func().stmts.push(Stmt::Break(None));
self.context.func().stmts.push(Stmt::EndIf);
}
};

match &iterable_ty {
Type::Array(elem) => {
self.translate_for_head(
&f.left,
Expr::Member {
object: Box::new(Expr::VarLoad {
span: Span::default(),
variable: iterator_var,
}),
key: PropNameOrExpr::Expr(
Box::new(Expr::VarUpdate {
op: crate::ast::UpdateOp::SuffixAdd,
variable: counter,
}),
Type::Int,
),
optional: false,
},
elem.as_ref().clone(),
)?;
}
Type::Tuple(elems) => {
self.translate_for_head(
&f.left,
Expr::Member {
object: Box::new(Expr::VarLoad {
span: Span::default(),
variable: iterator_var,
}),
key: PropNameOrExpr::Expr(
Box::new(Expr::VarUpdate {
op: crate::ast::UpdateOp::SuffixAdd,
variable: counter,
}),
Type::Int,
),
optional: false,
},
Type::Union(elems.clone()),
```

```
)?;
}
_ => {
self.translate_for_head(
&f.left,
// counter++
Expr::Member {
object: Box::new(Expr::VarLoad {
span: Span::default(),
variable: iterator_result_var,
}),
key: PropNameOrExpr::PropName(PropName::Ident("value".to_string())),
optional: false,
},
iterator_value_ty,
)?;
}
}

// translate the body
self.translate_stmt(&f.body, None)?;

// end scope
self.context.end_scope();
// end loop
self.context.func().stmts.push(Stmt::EndLoop);

if let Some(label) = label {
if !old_break {
self.break_labels.remove(label);
}
if !old_continue {
self.continue_labels.remove(label);
}
}
return Ok(());
}

fn translate_for_head(
&mut self,
for_head: &swc::ForHead,
mut expr: Expr,
ty: Type,
) -> Result<()> {
```

```
match for_head {
swc::ForHead::Pat(p) => {
if let Some(ident) = p.as_ident() {
if let Some(ann) = &ident.type_ann {
return Err(Error::syntax_error(ann.span, "type annotation not allowed"));
}

match self.context.find(&ident.sym).cloned() {
Some(Binding::Var {
writable,
id,
ty: var_ty,
..
}) => {
self.type_check(ident.span, &ty, &var_ty)?;

if ty != var_ty {
expr = Expr::Cast(Box::new(expr), var_ty);
}

if !writable {
return Err(Error::syntax_error(
ident.span,
"cannot assign to constant variable",
));
}
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: id,
value: Box::new(expr),
})));
return Ok(());
}
Some(Binding::Using { .. }) => {
return Err(Error::syntax_error(
ident.span,
"cannot assign to constant variable",
))
}
None => {
return Err(Error::syntax_error(
ident.span,
format!("undefined identifier '{}'", ident.sym),
))
```

```
}
_ => {
return Err(Error::syntax_error(
ident.span,
format!(
"expected variable, identifier '{}' is not a variable",
ident.sym
),
))
}
}
} else {
return Err(Error::syntax_error(
p.span(),
"for head can only have one variable binding",
));
}
}
swc::ForHead::UsingDecl(u) => {
if u.decls.len() != 1 {
return Err(Error::syntax_error(
u.span,
"for head can only have one variable binding",
));
}
if let Some(ident) = u.decls[0].name.as_ident() {
let var_id = VariableId::new();
let var_ty;

if let Some(ann) = &ident.type_ann {
let t = self.translate_type(&ann.type_ann)?;
self.type_check(ann.span, &ty, &t)?;

if ty != t {
expr = Expr::Cast(Box::new(expr), t.clone());
}

var_ty = t;
} else {
var_ty = ty;
};

if !self.context.declare(
&ident.sym,
```

```
Binding::Using {
id: var_id,
ty: var_ty.clone(),
is_await: u.is_await,
},
) {
return Err(Error::syntax_error(
ident.id.span,
format!("duplicated identifier '{}'", ident.sym),
));
}

self.context
.func()
.stmts
.push(Stmt::DeclareVar(var_id, var_ty));
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: var_id,
value: Box::new(expr),
})));
} else {
return Err(Error::syntax_error(
u.span,
"destructive pattern not allowed",
));
}
}
swc::ForHead::VarDecl(v) => {
if v.decls.len() != 1 {
return Err(Error::syntax_error(
v.span,
"for head can only have one variable binding",
));
}
if let Some(ident) = v.decls[0].name.as_ident() {
let var_id = VariableId::new();
let var_ty;

if let Some(ann) = &ident.type_ann {
let t = self.translate_type(&ann.type_ann)?;
self.type_check(ann.span, &ty, &t)?;

if ty != t {
```

```
expr = Expr::Cast(Box::new(expr), t.clone());
}

var_ty = t;
} else {
var_ty = ty;
};

if !self.context.declare(
&ident.sym,
Binding::Var {
writable: v.kind != swc::VarDeclKind::Const,
redeclarable: v.kind == swc::VarDeclKind::Var,
id: var_id,
ty: var_ty.clone(),
},
) {
return Err(Error::syntax_error(
ident.id.span,
format!("duplicated identifier '{}'", ident.sym),
));
}

self.context
.func()
.stmts
.push(Stmt::DeclareVar(var_id, var_ty));
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: var_id,
value: Box::new(expr),
})));
} else {
return Err(Error::syntax_error(
v.span,
"destructive pattern not allowed",
));
}
}
}
return Ok(());
}

pub fn translate_if_stmt(&mut self, i: &swc::IfStmt) -> Result<()> {
```

```
self.context.new_scope();

let (test, _ty) = self.translate_expr(&i.test, Some(&Type::Bool))?;
self.context.func().stmts.push(Stmt::If { test: Box::new(test) });

self.hoist_stmts([i.cons.as_ref()].into_iter())?;
self.translate_stmt(&i.cons, None)?;

self.context.end_scope();
self.context.func().stmts.push(Stmt::EndIf);

if let Some(alt) = &i.alt {
self.context.new_scope();
self.context.func().stmts.push(Stmt::Else);

self.hoist_stmts([alt.as_ref()].into_iter())?;
self.translate_stmt(&alt, None)?;

self.context.end_scope();
self.context.func().stmts.push(Stmt::EndElse);
}
return Ok(());
}

pub fn translate_try_catch_stmt(&mut self, t: &swc::TryStmt) -> Result<()> {
self.context.func().stmts.push(Stmt::Try);

self.translate_block_stmt(&t.block, None)?;

self.context.func().stmts.push(Stmt::EndTry);

if let Some(handler) = &t.handler {
let varid = VariableId::new();
let mut catch_ty = Type::Any;

self.context.new_scope();

if let Some(pat) = &handler.param {
// only accept ident
if let Some(ident) = pat.as_ident() {
if let Some(ann) = &ident.type_ann {
catch_ty = self.translate_type(&ann.type_ann)?;
}
self.context.declare(
```

```
&ident.sym,
super::context::Binding::Var {
writable: false,
redeclarable: false,
id: varid,
ty: catch_ty.clone(),
},
);
} else {
return Err(Error::syntax_error(pat.span(), "expected ident"));
}
} else {
self.context.func().variables.insert(
varid,
crate::ast::VariableDesc {
ty: Type::Any,
is_heap: false,
is_captured: false,
},
);
}

// enter catch
self.context.func().stmts.push(Stmt::Catch(varid, Box::new(catch_ty)));

// translate the body
self.translate_block_stmt(&handler.body, None)?;

// must go before end catch
self.context.end_scope();
// end catch
self.context.func().stmts.push(Stmt::EndCatch);
}

if let Some(finally) = &t.finalizer {
self.context.func().stmts.push(Stmt::Finally);

// translate the block
self.translate_block_stmt(&finally, None)?;

self.context.func().stmts.push(Stmt::EndFinally);
}

return Ok(());
```

```
}

pub fn translate_switch_stmt(&mut self, s: &swc::SwitchStmt) -> Result<()> {
let (test, test_ty) = self.translate_expr(&s.discriminant, None)?;

// push switch
self.context.func().stmts.push(Stmt::Switch(Box::new(test)));

for case in &s.cases {
if let Some(expr) = &case.test {
let (expr, _ty) = self.translate_expr(&expr, Some(&test_ty))?;

self.context.new_scope();
self.context.func().stmts.push(Stmt::SwitchCase(Box::new(expr)));

self.hoist_stmts(case.cons.iter())?;

for stmt in &case.cons {
self.translate_stmt(stmt, None)?;
}

self.context.end_scope();
self.context.func().stmts.push(Stmt::EndSwitchCase);
}
}

let mut default = false;
for case in &s.cases {
if case.test.is_none() {
if default {
return Err(Error::syntax_error(
case.span,
"A 'default' clause cannot appear more than once in a 'switch' statement",
));
}
default = true;
self.context.func().stmts.push(Stmt::DefaultCase);
self.context.new_scope();

self.hoist_stmts(case.cons.iter())?;

for stmt in &case.cons {
self.translate_stmt(stmt, None)?;
}
```

```
self.context.end_scope();
self.context.func().stmts.push(Stmt::EndDefaultCase);
}
}

// end switch
self.context.func().stmts.push(Stmt::EndSwitch);

return Ok(());
}
}
```

5.2.12 native-ts-hir/transform/function.rs

```
use native_js_common::error::Error;
use native_ts_parser::swc_core::common::{Span, Spanned};
use native_ts_parser::swc_core::ecma::ast as swc;

use crate::{
ast::{Expr, FuncType, Stmt, Type},
common::FunctionId,
transform::context::Binding,
};

use super::Transformer;

type Result<T> = std::result::Result<T, Error<Span>>;

impl Transformer {
pub fn translate_arrow(
&mut self,
func: &swc::ArrowExpr,
expected: Option<&FuncType>,
) -> Result<(Expr, Type)> {
let id = FunctionId::new();

let mut func_ty = FuncType {
this_ty: self.this_ty.clone(),
params: Vec::new(),
var_arg: false,
return_ty: Type::Undefined,
};

// function params
```

```
for p in func.params.iter() {
if let Some(ident) = p.as_ident() {
if ident.type_ann.is_none() {
return Err(Error::syntax_error(ident.span, "missing type annotation"));
}
let mut ty = self.translate_type(&ident.type_ann.as_ref().unwrap().type_ann)?;

if ident.optional {
ty = ty.union(Type::Undefined);
}
func_ty.params.push(ty);
} else {
return Err(Error::syntax_error(
p.span(),
"destructive param is not allowed",
));
}
}

let mut return_ty = match &func.return_type {
Some(ann) => Some(self.translate_type(&ann.type_ann)?),
None => None,
};

self.context.new_function(id);

match func.body.as_ref() {
swc::BlockStmtOrExpr::Expr(e) => {
let (expr, ty) = self.translate_expr(e, return_ty.as_ref())?;

let mut need_cast = false;

if return_ty.is_none() {
if let Some(expected) = expected {
self.type_check(e.span(), &ty, &expected.return_ty)?;
return_ty = Some(expected.return_ty.clone());

need_cast = expected.return_ty != ty;
} else {
return_ty = Some(ty);
}
}

if need_cast {
```

```
self.context.func().stmts.push(Stmt::Return(Box::new(Expr::Cast(
Box::new(expr),
expected.unwrap().return_ty.clone(),
))))
} else {
// simply return
self.context.func().stmts.push(Stmt::Return(Box::new(expr)));
}
}
swc::BlockStmtOrExpr::BlockStmt(b) => {
if return_ty.is_none() {
return_ty = Some(Type::Undefined);

if let Some(expected) = expected {
return_ty = Some(expected.return_ty.clone());
}
}
self.return_ty = return_ty.as_ref().cloned().unwrap();

self.translate_block_stmt(b, None)?;
}
}

func_ty.return_ty = return_ty.unwrap();

let func_id = self.context.end_function();
debug_assert!(func_id == id);

return Ok((Expr::Closure(id), Type::Function(Box::new(func_ty))));
}

pub fn translate_function(
&mut self,
id: FunctionId,
class_this_ty: Option<Type>,
func: &swc::Function,
) -> Result<()> {
self.context.new_function(id);

let mut this_ty = Type::Any;

if let Some(_type_params) = &func.type_params {
todo!("generic function")
}
```

```
for (i, p) in func.params.iter().enumerate() {
if let Some(ident) = p.pat.as_ident() {
let id = self.context.func().params[i].id;
let ty = self.context.func().params[i].ty.clone();

// declare binding
self.context.declare(
&ident.sym,
Binding::Var {
writable: true,
redeclarable: true,
id: id,
ty: ty,
},
);
}
}

if let Some(ty) = class_this_ty {
this_ty = ty;
}

let mut return_ty = if let Some(ann) = &func.return_type {
self.translate_type(&ann.type_ann)?
} else {
Type::Undefined
};

if func.is_async {
return_ty = Type::Promise(Box::new(return_ty));
}
if func.is_generator {
return_ty = Type::Iterator(Box::new(return_ty));
}

let old_this_ty = core::mem::replace(&mut self.this_ty, this_ty);
let old_return_ty = core::mem::replace(&mut self.return_ty, return_ty);

// translate body
if let Some(block) = &func.body {
self.translate_block_stmt(block, None)?
} else {
return Err(Error::syntax_error(func.span, "missing function body"));
```

```
}

let this_ty = core::mem::replace(&mut self.this_ty, old_this_ty);
let return_ty = core::mem::replace(&mut self.return_ty, old_return_ty);

let f = self.context.func();
f.this_ty = this_ty;
f.return_ty = return_ty;

let func_id = self.context.end_function();
debug_assert!(func_id == id);

return Ok(());
}
}
```

5.2.13 native-ts-hir/transform/expr.rs

```
use native_js_common::error::Error;
use num_traits::ToPrimitive;

use native_ts_parser::swc_core::common::{Span, Spanned};
use native_ts_parser::swc_core::ecma::ast as swc;

use crate::{
    ast::{Callee, Expr, FuncType, PropNameOrExpr, Type},
    PropName,
};

use super::{context::Binding, Transformer};

type Result<T> = std::result::Result<T, Error<Span>>;

impl Transformer {
    /// entry for translateing expressions
    pub(crate) fn translate_expr(
        &mut self,
```

```
    expr: &swc::Expr,
    expected_ty: Option<&Type>,
) -> Result<(Expr, Type)> {
    let (mut e, ty) = match expr {
        swc::Expr::Array(a) => self.translate_array_expr(a, expected_ty)?,
        swc::Expr::Arrow(a) => {
            let expected = expected_ty.and_then(|ty| {
                if let Type::Function(f) = ty {
                    Some(f.as_ref())
                } else {
                    None
                }
            });
            self.translate_arrow(a, expected)?
        }
        swc::Expr::Assign(a) => self.translate_assign(a)?,
        // await expression
        swc::Expr::Await(a) => {
            // translate the promise
            let (e, mut ty) = self.translate_expr(&a.arg, None)?;

            // get the result type of promise
            if let Type::Promise(p) = ty {
                ty = *p;
            };

            // get the result type of promise in union
            if let Type::Union(u) = ty {
                // create new union
                let mut v = Vec::with_capacity(u.len());

                for t in u.iter() {
```

```
            // type is promise
            if let Type::Promise(p) = t {
                // push promise result type
                v.push(p.as_ref().clone());
            } else {
                // push the type
                v.push(t.clone())
            }
        }
        // write union
        ty = Type::Union(v.into_boxed_slice())
    }

    (Expr::Await(Box::new(e)), ty)
}
// binary expression
swc::Expr::Bin(b) => self.translate_bin(b)?,
// function call expression
swc::Expr::Call(c) => self.translate_call(c)?,
// class exression
swc::Expr::Class(c) => {
    // class can only be declared as a type
    return Err(Error::syntax_error(
        c.span(),
        "class expression not allowed",
    ))
}
// conditional expression
swc::Expr::Cond(cond) => self.translate_cond(cond)?,
// function expression
swc::Expr::Fn(f) => {
    let id = self.hoist_function(None, &f.function)?;
```

```
    self.translate_function(id, None, &f.function)?;

    let ty = self
        .context
        .functions
        .get(&id)
        .expect("invalid function")
        .ty();

    (Expr::Closure(id), Type::Function(Box::new(ty)))
}
swc::Expr::Ident(id) => self.translate_var_load(id)?,
swc::Expr::This(_) => (Expr::This, self.this_ty.clone()),
swc::Expr::Object(o) => {
    return Err(Error::syntax_error(o.span, "object literal not allowed"))
}
swc::Expr::Unary(u) => self.translate_unary_expr(u)?,
swc::Expr::Update(u) => self.translate_update_expr(u)?,
swc::Expr::Member(m) => self.translate_member(m)?,
swc::Expr::SuperProp(s) => self.translate_super_prop_expr(s)?,
swc::Expr::New(n) => self.translate_new_expr(n)?,
swc::Expr::Seq(s) => self.translate_seq_expr(s, expected_ty)?,
swc::Expr::Lit(l) => self.translate_lit(l, expected_ty)?,
// todo: string template
swc::Expr::Tpl(_) => todo!(),
swc::Expr::TaggedTpl(_) => todo!(),
swc::Expr::Yield(y) => {
    let arg = if let Some(expr) = &y.arg {
        let r = self.return_ty.clone();
        let (e, _) = self.translate_expr(expr, Some(&r))?;
        e
    } else {
```

```
            self.type_check(y.span, &Type::Undefined, &self.return_ty)?;
            Expr::Undefined
        };
        // for now use undefined
        (Expr::Yield(Box::new(arg)), Type::Undefined)
    }
    swc::Expr::Paren(p) => self.translate_expr(&p.expr, expected_ty)?,

    swc::Expr::MetaProp(m) => {
        return Err(Error::syntax_error(m.span, "meta properties not supported"))
    }
    swc::Expr::PrivateName(p) => {
        return Err(Error::syntax_error(p.span, "invalid expression"))
    }
    swc::Expr::Invalid(i) => return Err(Error::syntax_error(i.span, "invalid expression")),
    swc::Expr::OptChain(o) => self.translate_optchain(o)?,

    swc::Expr::TsAs(a) => {
        let ty = self.translate_type(&a.type_ann)?;
        // translate expr will handle the cast
        let (expr, ty) = self.translate_expr(&a.expr, Some(&ty))?;
        (expr, ty)
    }
    swc::Expr::TsConstAssertion(c) => {
        // does nothing for now
        self.translate_expr(&c.expr, expected_ty)?
    }
    swc::Expr::TsInstantiation(_i) => {
        todo!("generics")
    }
    swc::Expr::TsNonNull(n) => {
        let (expr, mut ty) = self.translate_expr(&n.expr, None)?;
```

```
    if ty == Type::Undefined || ty == Type::Null {
        return Err(Error::syntax_error(
            n.span,
            "non null assertion cannot be applied to null types",
        ));
    }

    if let Type::Union(u) = &ty {
        let mut v = Vec::new();
        for ty in u.iter() {
            if ty != &Type::Undefined && ty != &Type::Null {
                v.push(ty.clone());
            }
        }
        ty = Type::Union(v.into_boxed_slice());
    }

    (Expr::AssertNonNull(Box::new(expr)), ty)
}
swc::Expr::TsSatisfies(s) => {
    let ty = self.translate_type(&s.type_ann)?;
    let (expr, expr_ty) = self.translate_expr(&s.expr, expected_ty)?;
    // check satisfies
    self.type_check(s.span, &expr_ty, &ty)?;

    (expr, expr_ty)
}
swc::Expr::TsTypeAssertion(t) => {
    let ty = self.translate_type(&t.type_ann)?;
    // translate expr will handle the cast
    let (expr, ty) = self.translate_expr(&t.expr, Some(&ty))?;
```

```
            (expr, ty)
        }

        swc::Expr::JSXElement(_)
        | swc::Expr::JSXEmpty(_)
        | swc::Expr::JSXFragment(_)
        | swc::Expr::JSXMember(_)
        | swc::Expr::JSXNamespacedName(_) => unimplemented!(),
    };

    if let Some(expected) = expected_ty {
        self.type_check(expr.span(), &ty, expected)?;

        if !ty.eq(expected) {
            e = Expr::Cast(Box::new(e), expected.clone())
        }

        return Ok((e, expected.clone()));
    }

    return Ok((e, ty));
}

/// array construction expression
pub fn translate_array_expr(
    &mut self,
    a: &swc::ArrayLit,
    expected_ty: Option<&Type>,
) -> Result<(Expr, Type)> {
    match expected_ty {
        Some(Type::Array(expected_elem_ty)) => {
            let mut elements = Vec::new();
```

```
    for elem in &a.elems {
        let (span, mut expr, ty) = if let Some(elem) = elem {
            if let Some(spread) = elem.spread {
                return Err(Error::syntax_error(
                    spread,
                    "spread expression not allowed",
                ));
            }

            let (e, t) = self.translate_expr(&elem.expr, Some(expected_elem_ty))?;
            (elem.expr.span(), e, t)
        } else {
            (a.span, Expr::Undefined, Type::Undefined)
        };

        self.type_check(span, &Type::Undefined, &expected_elem_ty)?;

        if !expected_elem_ty.as_ref().eq(&ty) {
            expr = Expr::Cast(Box::new(expr), expected_elem_ty.as_ref().clone());
        }

        elements.push(expr);
    }

    return Ok((
        Expr::Array { values: elements },
        Type::Array(expected_elem_ty.clone()),
    ));
}
Some(Type::Tuple(element_tys)) => {
    if element_tys.len() != a.elems.len() {
```

```
    return Err(Error::syntax_error(
        a.span,
        format!(
            "expected {} elements, {} were given",
            element_tys.len(),
            a.elems.len()
        ),
    ));
}

let mut values = Vec::with_capacity(element_tys.len());

for (i, elem) in a.elems.iter().enumerate() {
    let expected = element_tys.get(i).unwrap();

    let (_span, expr, _ty) = if let Some(elem) = elem {
        // spread expression not allowed
        if let Some(spread) = elem.spread {
            return Err(Error::syntax_error(
                spread,
                "spread expression not allowed",
            ));
        }
        // translate value
        let (e, t) = self.translate_expr(&elem.expr, Some(expected))?;

        (elem.span(), e, t)
    } else {
        // check if undefined fulfills expected type
        self.type_check(a.span, &Type::Undefined, expected)?;

        (a.span, Expr::Undefined, Type::Undefined)
```

```
            };

            // push expression to values
            values.push(expr);
        }

        // return the tuple
        return Ok((
            Expr::Tuple { values: values },
            Type::Tuple(element_tys.clone()),
        ));
    }
    _ => {
        let mut ty: Option<Type> = None;
        let mut values = Vec::new();

        for elem in &a.elems {
            // translate the element if not empty
            let (mut expr, mut t) = if let Some(elem) = elem {
                let (expr, t) = self.translate_expr(&elem.expr, None)?;
                (expr, t)
            } else {
                // if it is empty, it is undefined
                (Expr::Undefined, Type::Undefined)
            };

            // cannot be int, must be casted to number
            if t == Type::Int {
                t = Type::Number;
                expr = Expr::Cast(Box::new(expr), Type::Number);
            }
```

```
                if let Some(chained) = ty {
                    ty = Some(chained.union(t));
                } else {
                    ty = Some(t);
                };
                values.push(expr);
            }

            let array_ty = ty
                .map(|t| Type::Array(Box::new(t)))
                .or_else(|| Some(Type::Array(Box::new(Type::Any))))
                .unwrap();

            return Ok((Expr::Array { values: values }, array_ty));
        }
    }
}

pub fn translate_assign(&mut self, a: &swc::AssignExpr) -> Result<(Expr, Type)> {
    let (value, value_ty) = self.translate_expr(&a.right, None)?;

    return self.translate_assign_target(a.span, &a.left, a.op, value, value_ty);
}

pub fn translate_assign_target(
    &mut self,
    span: Span,
    target: &swc::AssignTarget,
    op: swc::AssignOp,
    value: Expr,
    value_ty: Type,
) -> Result<(Expr, Type)> {
```

```
match target {
    // trnalate simple assignments
    swc::AssignTarget::Simple(simple) => match simple {
        swc::SimpleAssignTarget::Ident(id) => {
            // variable assignment
            self.translate_var_assign(span, &id.id, op, value, value_ty)
        }
        swc::SimpleAssignTarget::Member(m) => {
            // member assignment
            self.translate_member_assign(span, m, op, value, value_ty)
        }
        swc::SimpleAssignTarget::Invalid(i) => {
            // invalid target
            Err(Error::syntax_error(i.span, "invalid assignment target"))
        }
        // todo: simple assignments
        _ => todo!("assignment"),
    },
    // translate pattern assignments
    swc::AssignTarget::Pat(p) => match p {
        swc::AssignTargetPat::Object(pat) => {
            // only assignment is allowed
            if op != swc::AssignOp::Assign {
                return Err(Error::syntax_error(
                    span,
                    format!("operator '{}' is not allowed on type 'object'", op.as_str()),
                ));
            }
            // translate object pattern assignment
            self.translate_object_pat_assign(pat, value, value_ty)
        }
        swc::AssignTargetPat::Array(pat) => {
```

```
            // only assignment is allowed
            if op != swc::AssignOp::Assign {
                return Err(Error::syntax_error(
                    span,
                    format!("operator '{}' is not allowed on type 'object'", op.as_str()),
                ));
            }
            // translate array pattern assignment
            self.translate_array_pat_assign(pat, value, value_ty)
        }
        swc::AssignTargetPat::Invalid(i) => {
            // invalid pattern
            Err(Error::syntax_error(i.span, "invalid pattern"))
        }
    },
  }
}

/// check the operand type of an assignment operation
fn check_assign_op_type(&mut self, span: Span, op: swc::AssignOp, ty: &Type) -> Result<()> {
  match op {
    swc::AssignOp::Assign => {}
    swc::AssignOp::AddAssign => match ty {
      Type::Bigint
      | Type::LiteralBigint(_)
      | Type::Number
      | Type::LiteralNumber(_)
      | Type::Int
      | Type::LiteralInt(_)
      | Type::String
      | Type::LiteralString(_) => {}
      _ => {
```

```
        return Err(Error::syntax_error(
            span,
            "operator '+=' only accepts number, bigint and string",
        ))
    }
},
swc::AssignOp::SubAssign
| swc::AssignOp::MulAssign
| swc::AssignOp::DivAssign
| swc::AssignOp::ExpAssign
| swc::AssignOp::ModAssign
| swc::AssignOp::LShiftAssign
| swc::AssignOp::RShiftAssign
| swc::AssignOp::ZeroFillRShiftAssign
| swc::AssignOp::BitAndAssign
| swc::AssignOp::BitOrAssign
| swc::AssignOp::BitXorAssign => match ty {
    Type::Bigint
    | Type::LiteralBigint(_)
    | Type::Number
    | Type::LiteralNumber(_)
    | Type::Int
    | Type::LiteralInt(_) => {}
    _ => {
        return Err(Error::syntax_error(
            span,
            format!("operator '{}' only accepts number and bigint", op.as_str()),
        ))
    }
},
swc::AssignOp::NullishAssign => {}
swc::AssignOp::OrAssign => {}
```

```
        swc::AssignOp::AndAssign => {
            if ty != &Type::Bool {
                return Err(Error::syntax_error(
                    span,
                    "operator '&&=' only accepts boolean",
                ));
            }
        }
    };

    return Ok(());
}

pub fn translate_var_assign(
    &mut self,
    span: Span,
    var: &swc::Ident,
    op: swc::AssignOp,
    mut value: Expr,
    value_ty: Type,
) -> Result<(Expr, Type)> {
    // get the variable id and variable type
    let (varid, var_ty) = if let Some(binding) = self.context.find(&var.sym) {
        match binding {
            Binding::Var {
                writable,
                redeclarable: _,
                id,
                ty,
            } => {
                // constants are not writable
                if !*writable {
```

```
                return Err(Error::syntax_error(
                    var.span,
                    format!("variable '{}' is immutable", &var.sym),
                ));
            }
            // return id and type
            (*id, ty.clone())
        }
        // using declare cannot be mutated
        Binding::Using { .. } => {
            return Err(Error::syntax_error(
                var.span,
                format!("variable '{}' is immutable", &var.sym),
            ))
        }
        // all other bindings are not considered variable
        _ => {
            return Err(Error::syntax_error(
                var.span,
                format!("identifier '{}' is not a variable", &var.sym),
            ))
        }
    }
} else {
    return Err(Error::syntax_error(
        var.span,
        format!("undeclared identifier '{}'", var.sym),
    ));
};

// type check
self.type_check(var.span, &value_ty, &var_ty)?;
```

```
    if value_ty != var_ty {
        value = Expr::Cast(Box::new(value), var_ty.clone());
    };

    // check if operand type is valid for assignment
    self.check_assign_op_type(span, op, &var_ty)?;

    return Ok((
        Expr::VarAssign {
            op: op.into(),
            variable: varid,
            value: Box::new(value),
        },
        var_ty,
    ));
}

pub fn translate_member_assign(
    &mut self,
    span: Span,
    member: &swc::MemberExpr,
    op: swc::AssignOp,
    mut value: Expr,
    value_ty: Type,
) -> Result<(Expr, Type)> {
    let (member_expr, member_ty) = self.translate_member(member)?;

    self.type_check(span, &value_ty, &member_ty)?;

    if value_ty != member_ty {
        value = Expr::Cast(Box::new(value), member_ty.clone());
```

```
    }

    if let Expr::Member {
        object,
        key,
        optional,
    } = member_expr
    {
        if optional {
            return Err(Error::syntax_error(
                member.span(),
                "invalid left-hand side assignment",
            ));
        }

        self.check_assign_op_type(span, op, &member_ty)?;

        return Ok((
            Expr::MemberAssign {
                op: op.into(),
                object: object,
                key: key,
                value: Box::new(value),
            },
            member_ty,
        ));
    } else {
        unreachable!()
    }
}

pub fn translate_object_pat_assign(
```

```
    &mut self,
    pat: &swc::ObjectPat,
    value: Expr,
    value_ty: Type,
) -> Result<(Expr, Type)> {
    // top level pattern cannot be optional
    if pat.optional {
        return Err(Error::syntax_error(
            pat.span,
            "object pattern cannot be optional",
        ));
    }

    // type annotation not allowed
    if let Some(ann) = &pat.type_ann {
        return Err(Error::syntax_error(ann.span, "type annotation not allowed"));
    }

    // result object expression
    let mut obj_expr = Vec::new();
    // result object type
    let mut obj_ty = Vec::new();
    // counter
    let mut i = 0;

    // translate property assignment
    for prop in &pat.props {
        // is first property
        let is_first = i == 0;
        // is last property
        let is_last = i == pat.props.len() - 1;
        // increment counter
```

```
i += 1;

match prop {
    swc::ObjectPatProp::Assign(a) => {
        // construct prop name
        let prop = PropName::Ident(a.key.id.sym.to_string());
        // default value
        if let Some(_) = &a.value {
            // todo: assignment pattern
            return Err(Error::syntax_error(
                a.span,
                "default assignment is not allowed",
            ));
        }
        // type annotation not allowed
        if let Some(ann) = &a.key.type_ann {
            return Err(Error::syntax_error(ann.span, "type annotation not allowed"))
        }

        // value type of property
        let value_ty =
            if let Some(value_ty) = self.type_has_property(&value_ty, &prop, false) {
                value_ty
            } else {
                return Err(Error::syntax_error(
                    a.span,
                    format!("value has no property '{}'", a.key.id.sym),
                ));
            };

        let v = if is_first {
            // todo: avoid clone
```

```
            Expr::Push(Box::new(value.clone()))
        } else if is_last {
            Expr::Pop
        } else {
            Expr::ReadStack
        };

        let v = Expr::Member {
            object: Box::new(v),
            key: PropNameOrExpr::PropName(prop.clone()),
            optional: false,
        };

        let (assign_expr, assign_ty) =
            self.translate_var_assign(a.span, &a.key.id, swc::AssignOp::Assign, v, value_ty)?;

        obj_expr.push((prop.clone(), assign_expr));
        obj_ty.push((prop, assign_ty));
    }
    swc::ObjectPatProp::KeyValue(k) => {
        let key = self.translate_prop_name(&k.key)?;
        let key = match key {
            PropNameOrExpr::PropName(p) => p,
            PropNameOrExpr::Expr(_, _) => unimplemented!(),
        };

        let v = if is_first {
            // todo: avoid clone
            Expr::Push(Box::new(value.clone()))
        } else if is_last {
            Expr::Pop
        } else {
```

```
            Expr::ReadStack
        };

        let v = Expr::Member {
            object: Box::new(v),
            key: PropNameOrExpr::PropName(key.clone()),
            optional: false,
        };

        if let Some(value_ty) = self.type_has_property(&value_ty, &key, false) {
            // translate pattern assignment
            let (expr, assign_ty) = self.translate_pat_assign(
                &k.value,
                swc::AssignOp::Assign,
                v,
                value_ty,
            )?;

            obj_expr.push((key.clone(), expr));
            obj_ty.push((key, assign_ty))

        } else {
            return Err(Error::syntax_error(
                k.key.span(),
                format!("type " has no property '{}'", key),
            ));
        }
    }
    swc::ObjectPatProp::Rest(r) => {
        // todo: rest assignment
        return Err(Error::syntax_error(
            r.dot3_token,
```

```
                    "rest assignment not supported",
                ));
            }
        }
    }

    obj_ty.sort_by(|a, b| a.0.cmp(&b.0));

    return Ok((
        Expr::Object { props: obj_expr },
        Type::LiteralObject(obj_ty.into()),
    ));
}

pub fn translate_array_pat_assign(
    &mut self,
    pat: &swc::ArrayPat,
    value: Expr,
    value_ty: Type,
) -> Result<(Expr, Type)> {
    if pat.optional {
        if value_ty == Type::Undefined {
            return Ok((Expr::Undefined, Type::Undefined));
        }
    }
    if let Some(ann) = &pat.type_ann{
        return Err(Error::syntax_error(ann.span, "type annotation not allowed"))
    }

    let mut exprs = Vec::new();
    let mut tys = Vec::new();
```

```
let mut obj = Expr::Push(Box::new(value));

for i in 0..pat.elems.len() {
    let is_first = i == 0;
    let is_last = i == pat.elems.len() - 1;

    // translate the required assignment expression for the element
    let (expr, ty) = if let Some(elem) = &pat.elems[i] {
        // check if the target value has index property
        if let Some(value_ty) =
            self.type_has_property(&value_ty, &PropName::Int(i as _), false)
        {
            // get the object from stack
            let obj = if is_first {
                let o = obj;
                obj = Expr::ReadStack;
                o
            } else if is_last {
                // pop from stack
                Expr::Pop
            } else {
                // read from stack
                Expr::ReadStack
            };

            // translate pattern assignment of element
            let (expr, value_ty) = self.translate_pat_assign(
                elem,
                swc::AssignOp::Assign,
                Expr::Member {
                    // the object
                    object: Box::new(obj),
```

```
            // the index
            key: PropNameOrExpr::PropName(PropName::Int(i as _)),
            // not optional
            optional: false,
        },
        value_ty,
    )?;

    // return the assignment expression and type
    (expr, value_ty)
} else {
    // no index property is found, check if assignment is optional
    let is_optional = match elem {
        swc::Pat::Array(a) => a.optional,
        // todo: assign pattern
        swc::Pat::Assign(a) => {
            return Err(Error::syntax_error(
                a.span,
                "assignment pattern not supported",
            ))
        }
        // expr pattern is only valid in for in loops
        swc::Pat::Expr(_) => unreachable!(),
        swc::Pat::Ident(id) => id.optional,
        swc::Pat::Invalid(i) => {
            return Err(Error::syntax_error(
                i.span,
                "invalid left-hand side assignment",
            ))
        }
        swc::Pat::Object(o) => o.optional,
        // todo: rest assignment
```

```
                swc::Pat::Rest(r) => {
                    return Err(Error::syntax_error(
                        r.dot3_token,
                        "rest assignment not supported",
                    ))
                }
            };

            // return undefined if optional
            if is_optional {
                (Expr::Undefined, Type::Undefined)
            } else {
                // has no property and not optional, return error
                return Err(Error::syntax_error(
                    elem.span(),
                    format!("type " has no property '{}'", i),
                ));
            }
        }
    } else {
        // there is no assignment in this index, return undefined
        (Expr::Undefined, Type::Undefined)
    };

    // push to array type construction
    tys.push(ty);
    // push expression to array
    exprs.push(expr);
  }

  return Ok((Expr::Array { values: exprs }, Type::Tuple(tys.into_boxed_slice())));
}
```

```
pub fn translate_pat_assign(
    &mut self,
    pat: &swc::Pat,
    op: swc::AssignOp,
    value: Expr,
    value_ty: Type,
) -> Result<(Expr, Type)> {
    match pat {
        // destructive array assignment
        swc::Pat::Array(a) => self.translate_array_pat_assign(a, value, value_ty),
        // destructive object assignment
        swc::Pat::Object(o) => self.translate_object_pat_assign(o, value, value_ty),
        // todo: assignment pattern
        swc::Pat::Assign(a) => {
            return Err(Error::syntax_error(
                a.span,
                "assignment pattern not supported",
            ))
        }
        // expr pattern is only valid in for-in loops
        swc::Pat::Expr(_) => unreachable!(),
        // simple variable assignment
        swc::Pat::Ident(id) => {
            self.translate_var_assign(pat.span(), &id.id, op, value, value_ty)
        }
        swc::Pat::Invalid(i) => {
            return Err(Error::syntax_error(
                i.span,
                "invalid left-hand side assignment",
            ))
        }
```

```
        // todo: rest assignment
        swc::Pat::Rest(r) => {
            return Err(Error::syntax_error(
                r.dot3_token,
                "rest assignment is not supported",
            ))
        }
    }
}

pub fn translate_bin(&mut self, b: &swc::BinExpr) -> Result<(Expr, Type)> {
    if b.op == swc::BinaryOp::In {
        let prop = self.translate_computed_prop_name(&b.left)?;
        let (right, right_ty) = self.translate_expr(&b.right, None)?;

        match prop {
            PropNameOrExpr::PropName(prop) => {
                if let Some(_) = self.type_has_property(&right_ty, &prop, false) {
                    return Ok((
                        Expr::Seq(Box::new(right), Box::new(Expr::Bool(true))),
                        Type::Bool,
                    ));
                } else {
                    return Ok((
                        Expr::Seq(Box::new(right), Box::new(Expr::Bool(false))),
                        Type::Bool,
                    ));
                }
            }
            PropNameOrExpr::Expr(..) => {
                // TODO
                return Err(Error::syntax_error(
```

```
                b.span,
                "computed property name 'in' operation not supported",
            ));
        }
    }
}
let (mut left, mut left_ty) = self.translate_expr(&b.left, None)?;
let (mut right, mut right_ty) = self.translate_expr(&b.right, None)?;

// treat literal int as int
if let Type::LiteralInt(_) = left_ty {
    left_ty = Type::Int;
}
if let Type::LiteralInt(_) = right_ty {
    right_ty = Type::Int;
}
// treat literal number as number
if let Type::LiteralNumber(_) = left_ty {
    left_ty = Type::Number;
}
if let Type::LiteralNumber(_) = right_ty {
    right_ty = Type::Number;
}
// treat literal bigint as bigint
if let Type::LiteralBigint(_) = left_ty {
    left_ty = Type::Bigint;
}
if let Type::LiteralBigint(_) = right_ty {
    right_ty = Type::Bigint;
}
// treat literal bool as bool
if let Type::LiteralBool(_) = left_ty {
```

```
    left_ty = Type::Bool;
}
if let Type::LiteralBool(_) = right_ty {
    right_ty = Type::Bool;
}
// treat literal string as string
if let Type::LiteralString(_) = left_ty {
    left_ty = Type::String;
}
if let Type::LiteralString(_) = right_ty {
    right_ty = Type::String;
}

// the result type
let ty;

// check op and the result type
match b.op {
    swc::BinaryOp::Add
    | swc::BinaryOp::Sub
    | swc::BinaryOp::Div
    | swc::BinaryOp::Mul
    | swc::BinaryOp::Mod
    | swc::BinaryOp::Exp => {
        // both are number
        if left_ty == Type::Number && right_ty == Type::Number {
            ty = Type::Number;
        } else if left_ty == Type::Int && right_ty == Type::Int {
            // both are int
            ty = Type::Int;
        } else if left_ty == Type::Int && right_ty == Type::Number {
            // cast left hand side as number
```

```
                left_ty = Type::Number;
                left = Expr::Cast(Box::new(left), Type::Number);
                // result is number
                ty = Type::Number;
            } else if left_ty == Type::Number && right_ty == Type::Int {
                // cast right hand side as number
                right_ty = Type::Number;
                right = Expr::Cast(Box::new(right), Type::Number);
                // result is number
                ty = Type::Number;
            // both are bigint
            } else if left_ty == Type::Bigint && right_ty == Type::Bigint {
                ty = Type::Bigint;
            // both are string
            } else if b.op == swc::BinaryOp::Add
                && left_ty == Type::String
                && right_ty == Type::String
            {
                ty = Type::String
            } else {
                // unsupported types
                return Err(Error::syntax_error(b.span, format!("The operand of an arithmetic operation must be of type 'number' or 'bigint'")));
            }

            // right should be equal to left
            debug_assert!(right_ty == left_ty);
            debug_assert!(left_ty == Type::Int || left_ty == Type::Number || left_ty == Type::Bigint);
        }
        swc::BinaryOp::BitAnd
        | swc::BinaryOp::BitOr
        | swc::BinaryOp::BitXor
```

```
        | swc::BinaryOp::LShift
        | swc::BinaryOp::RShift
        | swc::BinaryOp::ZeroFillRShift => {
            if left_ty == Type::Number {
                // cast to int
                left_ty = Type::Int;
                left = Expr::Cast(Box::new(left), Type::Int);
            }
            if right_ty == Type::Number {
                // cast to int
                right_ty = Type::Int;
                right = Expr::Cast(Box::new(right), Type::Int)
            }
            // both sides must be int or bigint
            if !(left_ty == Type::Int && right_ty == Type::Int)
                || !(left_ty == Type::Bigint && right_ty == Type::Bigint)
            {
                return Err(Error::syntax_error(b.span, format!("The operand of an arithmetic operation must be of type 'number' or 'bigint'")));
            }

            // left should be equal to right
            debug_assert!(left_ty == right_ty);
            debug_assert!(left_ty == Type::Int || left_ty == Type::Bigint);

            // either bigint or int
            ty = left_ty;
        }
        swc::BinaryOp::EqEq
        | swc::BinaryOp::EqEqEq
        | swc::BinaryOp::NotEq
        | swc::BinaryOp::NotEqEq => {
```

```
            // result must be boolean
            ty = Type::Bool;
        }
        swc::BinaryOp::Gt | swc::BinaryOp::GtEq | swc::BinaryOp::Lt | swc::BinaryOp::LtEq => {
            // both are number
            if left_ty == Type::Number && right_ty == Type::Number {
                ty = Type::Bool;
            // both are int
            } else if left_ty == Type::Int && right_ty == Type::Int {
                ty = Type::Bool;
            } else if left_ty == Type::Int && right_ty == Type::Number {
                // cast left to number
                left_ty = Type::Number;
                left = Expr::Cast(Box::new(left), Type::Number);
                ty = Type::Bool;
            } else if left_ty == Type::Number && right_ty == Type::Int {
                // cast right to number
                right_ty = Type::Number;
                right = Expr::Cast(Box::new(right), Type::Number);
                ty = Type::Bool;
            // both are bigint
            } else if left_ty == Type::Bigint && right_ty == Type::Bigint {
                ty = Type::Bool;
            } else {
                // not bigint, number or int
                return Err(Error::syntax_error(b.span, format!("The operand of an arithmetic operation must be of type 'number' or 'bigint'")));
            }

            debug_assert!(right_ty == left_ty);
        }
        swc::BinaryOp::NullishCoalescing | swc::BinaryOp::LogicalOr => {
```

```
            ty = left_ty.union(right_ty);
        }
        swc::BinaryOp::LogicalAnd => {
            if left_ty != Type::Bool {
                left_ty = Type::Bool;
                left = Expr::Cast(Box::new(left), Type::Bool);
            }
            if right_ty != Type::Bool {
                right_ty = Type::Bool;
                right = Expr::Cast(Box::new(right), Type::Bool);
            }

            debug_assert_eq!(right_ty, left_ty);
            debug_assert_eq!(right_ty, Type::Bool);

            ty = Type::Bool
        }
        swc::BinaryOp::In | swc::BinaryOp::InstanceOf => unreachable!(),
    };

    return Ok((
        Expr::Bin {
            op: b.op.into(),
            left: Box::new(left),
            right: Box::new(right),
        },
        ty,
    ));
}

/// translates the call expression.
/// currently, generics are not supported yet
```

```
pub fn translate_call(&mut self, call: &swc::CallExpr) -> Result<(Expr, Type)> {
    let (callee, callee_ty) = match &call.callee {
        swc::Callee::Super(s) => {
            if !self.is_in_constructor {
                return Err(Error::syntax_error(
                    s.span,
                    "super call is only allowed in constructors",
                ));
            }

            let sup = self.super_class.expect("invalid super class");

            let constructor = &self
                .context
                .classes
                .get(&sup)
                .expect("invalid class")
                .constructor;

            let func_ty = if let Some((_, ty)) = constructor {
                ty.clone()
            } else {
                FuncType {
                    this_ty: Type::Object(sup),
                    params: Vec::new(),
                    var_arg: false,
                    return_ty: Type::Undefined,
                }
            };

            (Callee::Super(sup), func_ty)
        }
```

```
swc::Callee::Import(i) => {
    return Err(Error::syntax_error(i.span, "dynamic import not allowed"))
}
swc::Callee::Expr(e) => {
    // translate the expression
    let (expr, ty) = self.translate_expr(e, None)?;

    // check it is a function
    let func_ty = if let Type::Function(func) = ty {
        *func
    } else {
        return Err(Error::syntax_error(call.span, "callee is not a function"));
    };

    // convert to callee
    match expr {
        Expr::Member {
            object,
            key,
            optional: _,
        } => {
            // check if object matches func_ty.this type
            // TODO
            (
                Callee::Member {
                    object: *object,
                    prop: key,
                },
                func_ty,
            )
        }
        Expr::Function(f) => {
```

```
                // check this type matches
                let this_ty = self.this_ty.clone();
                self.type_check(call.span, &this_ty, &func_ty.this_ty)?;

                (Callee::Function(f), func_ty)
            }
            _ => {
                // check this type matches
                let this_ty = self.this_ty.clone();
                self.type_check(call.span, &this_ty, &func_ty.this_ty)?;

                (Callee::Expr(expr), func_ty)
            }
        }
    }
};

// type arguments
if call.type_args.is_some() {
    todo!("generics")
}

// if it is not a member call, we have to check
if !callee.is_member() {
    let this_ty = self.this_ty.clone();
    self.type_check(call.span, &this_ty, &callee_ty.this_ty)?;
}

// reference argument types
let expected_arguments: &[Type] = &callee_ty.params;

let mut args = Vec::new();
```

```
// since var args is not supported, length of arguments is fixed
if call.args.len() != callee_ty.params.len() {
    return Err(Error::syntax_error(
        call.span,
        format!(
            "expected {} arguments, {} were given",
            callee_ty.params.len(),
            call.args.len()
        ),
    ));
}

// handle arguments
for (i, arg) in call.args.iter().enumerate() {
    // spread ... is present
    if let Some(spread) = arg.spread {
        return Err(Error::syntax_error(
            spread,
            "variable arguments not supported",
        ));
    }

    // translate argument
    let (a, arg_ty) = self.translate_expr(&arg.expr, expected_arguments.get(i))?;

    // check argument fulfills type
    self.type_check(arg.span(), &arg_ty, &expected_arguments[i])?;

    if arg_ty.eq(&expected_arguments[i]) {
        // push expression to arguments
        args.push(a);
```

```
        } else {
            // convert value to type
            args.push(Expr::Cast(Box::new(a), expected_arguments[i].clone()))
        }
    }

    return Ok((
        Expr::Call {
            callee: Box::new(callee),
            args: args,
            optional: false,
        },
        callee_ty.return_ty,
    ));
}

pub fn translate_cond(&mut self, cond: &swc::CondExpr) -> Result<(Expr, Type)> {
    let (mut test, test_ty) = self.translate_expr(&cond.test, None)?;
    let (cons, cons_ty) = self.translate_expr(&cond.cons, None)?;
    let (alt, alt_ty) = self.translate_expr(&cond.alt, None)?;

    if test_ty != Type::Bool {
        // cast it to bool
        test = Expr::Cast(Box::new(test), Type::Bool);
    }

    return Ok((
        Expr::Ternary {
            test: Box::new(test),
            left: Box::new(cons),
            right: Box::new(alt),
        },
```

```
        cons_ty.union(alt_ty),
    ));
}

pub fn translate_var_load(&mut self, ident: &swc::Ident) -> Result<(Expr, Type)> {
    match self.context.find(&ident.sym) {
        Some(Binding::Var {
            writable,
            redeclarable: _,
            id,
            ty,
        }) => {
            if !*writable {
                return Err(Error::syntax_error(
                    ident.span,
                    "constant is not assignable",
                ));
            }

            return Ok((
                Expr::VarLoad {
                    span: ident.span,
                    variable: *id,
                },
                ty.clone(),
            ));
        }
        Some(Binding::Using { id, ty, .. }) => {
            return Ok((
                Expr::VarLoad {
                    span: ident.span,
                    variable: *id,
```

```
                },
                ty.clone(),
            ))
        }
        Some(Binding::Function(f)) => {
            // copy the id to avoid borrowing self
            let id = *f;
            // get the functio type
            let ty = self
                .context
                .functions
                .get(&id)
                .expect("invalid function id")
                .ty();
            // return expression
            return Ok((Expr::Function(id), Type::Function(Box::new(ty))));
        }
        None => {
            return Err(Error::syntax_error(
                ident.span,
                format!("undefined identifier '{}'", ident.sym),
            ))
        }
        _ => {
            return Err(Error::syntax_error(
                ident.span,
                format!("identifier '{}' is not a variable", ident.sym),
            ))
        }
    }
}
```

```
pub fn translate_member(&mut self, member: &swc::MemberExpr) -> Result<(Expr, Type)> {
    let (obj, obj_ty) = self.translate_expr(&member.obj, None)?;

    let prop = match &member.prop {
        swc::MemberProp::Computed(c) => self.translate_computed_prop_name(&c.expr)?,
        swc::MemberProp::Ident(id) => {
            PropNameOrExpr::PropName(PropName::Ident(id.sym.to_string()))
        }
        swc::MemberProp::PrivateName(id) => {
            if self.this_ty == obj_ty {
                PropNameOrExpr::PropName(PropName::Private(id.id.sym.to_string()))
            } else {
                return Err(Error::syntax_error(
                    id.span,
                    "cannot access privite properties outside of method",
                ));
            }
        }
    };

    match prop {
        PropNameOrExpr::PropName(name) => {
            if let Some(member_ty) = self.type_has_property(&obj_ty, &name, false) {
                return Ok((
                    Expr::Member {
                        object: Box::new(obj),
                        key: PropNameOrExpr::PropName(name),
                        optional: false,
                    },
                    member_ty,
                ));
            } else {
```

```
        return Err(Error::syntax_error(
            member.span,
            format!("type has no property '{}'", name),
        ));
    }
}
PropNameOrExpr::Expr(mut e, ty) => {
    match &obj_ty {
        Type::Map(k, v) => {
            self.type_check(member.span, &ty, k)?;

            if &ty != k.as_ref() {
                e = Box::new(Expr::Cast(e, k.as_ref().clone()));
            }
            return Ok((
                Expr::Member {
                    object: Box::new(obj),
                    key: PropNameOrExpr::Expr(e, ty),
                    optional: false,
                },
                v.as_ref().clone(),
            ));
        }
        Type::Array(elem) => match ty {
            Type::Int | Type::LiteralInt(_) | Type::Number | Type::LiteralNumber(_) => {
                return Ok((
                    Expr::Member {
                        object: Box::new(obj),
                        key: PropNameOrExpr::Expr(e, ty),
                        optional: false,
                    },
                    elem.as_ref().clone(),
```

```
        ));
    }
    _ => {
        return Err(Error::syntax_error(
            member.span,
            "array can only be indexed by number",
        ))
    }
},
Type::Tuple(elems) => match ty {
    Type::Int | Type::LiteralInt(_) | Type::Number | Type::LiteralNumber(_) => {
        return Ok((
            Expr::Member {
                object: Box::new(obj),
                key: PropNameOrExpr::Expr(e, ty),
                optional: false,
            },
            Type::Union(elems.clone()),
        ));
    }
    _ => {
        return Err(Error::syntax_error(
            member.span,
            "tuple can only be indexed by number",
        ));
    }
},
_ => {
    return Err(Error::syntax_error(
        member.span,
        "type " is not indexable, property must be literal",
    ))
```

```
            }
        };
    }
  }
}

pub fn translate_unary_expr(&mut self, u: &swc::UnaryExpr) -> Result<(Expr, Type)> {
  let (mut expr, ty) = self.translate_expr(&u.arg, None)?;
  match u.op {
    swc::UnaryOp::Bang => {
      if ty != Type::Bool {
        // cast type to bool
        expr = Expr::Cast(Box::new(expr), Type::Bool);
      }

      return Ok((
        Expr::Unary {
          op: crate::ast::UnaryOp::LogicalNot,
          value: Box::new(expr),
        },
        Type::Bool,
      ));
    }
    swc::UnaryOp::Delete => {
      return Err(Error::syntax_error(u.span, "'delete' is not allowed"))
    }
    swc::UnaryOp::Minus | swc::UnaryOp::Plus => {
      match ty {
        Type::Number
        | Type::LiteralNumber(_)
        | Type::Int
        | Type::LiteralInt(_)
```

```
        | Type::Bigint
        | Type::LiteralBigint(_) => {
            // return unary expression
            return Ok((
                Expr::Unary {
                    op: if u.op == swc::UnaryOp::Minus {
                        crate::ast::UnaryOp::Minus
                    } else {
                        crate::ast::UnaryOp::Plus
                    },
                    value: Box::new(expr),
                },
                match ty {
                    Type::Int => Type::Int,
                    Type::Bigint => Type::Bigint,
                    _ => Type::Number,
                },
            ));
        }
        _ => {
            return Err(Error::syntax_error(
                u.span,
                "right-hand side must be one of 'number', 'bigint' or 'boolean'",
            ))
        }
    }
}
swc::UnaryOp::Tilde => {
    match ty {
        Type::Number => {}
        Type::Int => {}
        Type::LiteralNumber(_) => {}
```

```
        Type::LiteralInt(_) => {}
        Type::Bigint => {}
        Type::LiteralBigint(_) => {}
        Type::Bool => {}
        Type::LiteralBool(_) => {}
        _ => {
            return Err(Error::syntax_error(
                u.span,
                "right-hand side must be one of 'number', 'bigint' or 'boolean'",
            ));
        }
    }

    if ty == Type::Number {
        expr = Expr::Cast(Box::new(expr), Type::Int)
    }
    if let Type::LiteralNumber(_) = ty {
        expr = Expr::Cast(Box::new(expr), Type::Int);
    }
    if let Type::LiteralBigint(_) = ty {
        expr = Expr::Cast(Box::new(expr), Type::Bigint);
    }

    // return expression
    return Ok((
        Expr::Unary {
            op: crate::ast::UnaryOp::BitNot,
            value: Box::new(expr),
        },
        Type::Int,
    ));
}
```

```
swc::UnaryOp::TypeOf => {
    let ty_s = match ty {
        Type::AnyObject
        | Type::Null
        | Type::Object(_)
        | Type::LiteralObject(_)
        | Type::Regex
        | Type::Array(_)
        | Type::Function(_)
        | Type::Map(_, _)
        | Type::Promise(_)
        | Type::Tuple(_)
        | Type::Iterator(_) => "object",
        Type::Bigint | Type::LiteralBigint(_) => "bigint",
        Type::Bool | Type::LiteralBool(_) => "boolean",
        Type::Enum(_)
        | Type::Int
        | Type::Number
        | Type::LiteralInt(_)
        | Type::LiteralNumber(_) => "number",
        Type::String | Type::LiteralString(_) => "string",
        Type::Symbol => "symbol",
        Type::Undefined => "undefined",
        // these type cannot be known at compile time
        Type::Any
        | Type::Alias(_)
        | Type::Generic(_)
        | Type::Interface(_)
        | Type::Union(_) => {
            // runtime reflect
            return Ok((
                Expr::Unary {
```

```
                    op: crate::ast::UnaryOp::Typeof,
                    value: Box::new(expr),
                },
                Type::String,
            ));
        }
    };

    // return the literal string of type
    return Ok((Expr::String(ty_s.to_string()), Type::String));
  }
  swc::UnaryOp::Void => {
    // simply return undefined
    return Ok((
      Expr::Unary {
        op: crate::ast::UnaryOp::Void,
        value: Box::new(expr),
      },
      Type::Undefined,
    ));
  }
 }
}

pub fn translate_update_expr(&mut self, u: &swc::UpdateExpr) -> Result<(Expr, Type)> {
  let (expr, ty) = self.translate_expr(&u.arg, None)?;

  let op = match u.op {
    swc::UpdateOp::MinusMinus => {
      if u.prefix {
        crate::ast::UpdateOp::PrefixSub
      } else {
```

```
            crate::ast::UpdateOp::SuffixSub
        }
    }
    swc::UpdateOp::PlusPlus => {
        if u.prefix {
            crate::ast::UpdateOp::PrefixAdd
        } else {
            crate::ast::UpdateOp::SuffixAdd
        }
    }
};

match ty {
    Type::Int
    | Type::LiteralInt(_)
    | Type::Number
    | Type::LiteralNumber(_)
    | Type::Bigint
    | Type::LiteralBigint(_) => {}
    _ => {
        return Err(Error::syntax_error(
            u.span,
            "operand must have type 'number' or 'bigint'",
        ))
    }
}

match expr {
    Expr::Member {
        object,
        key,
        optional,
```

```
} => {
    if optional {
        return Err(Error::syntax_error(
            u.span,
            "invalid left-hand side assignment",
        ));
    }

    return Ok((
        Expr::MemberUpdate {
            op: op,
            object: object,
            key: key,
        },
        ty,
    ));
}
Expr::VarLoad { variable, span: _ } => {
    return Ok((
        Expr::VarUpdate {
            op: op,
            variable: variable,
        },
        ty,
    ))
}
_ => {
    return Err(Error::syntax_error(
        u.span,
        "invalid left-hand side assignment",
    ))
}
```

```
        };
    }

    pub fn translate_super_prop_expr(&mut self, s: &swc::SuperPropExpr) -> Result<(Expr, Type)>
{
        if self.super_class.is_none() {
            return Err(Error::syntax_error(
                s.span,
                "'super' keyword unexpected here",
            ));
        }

        let prop = match &s.prop {
            swc::SuperProp::Computed(c) => self.translate_computed_prop_name(&c.expr)?,
            swc::SuperProp::Ident(id) => {
                PropNameOrExpr::PropName(PropName::Ident(id.sym.to_string()))
            }
        };

        let prop = match prop {
            PropNameOrExpr::PropName(p) => p,
            PropNameOrExpr::Expr(..) => {
                return Err(Error::syntax_error(
                    s.span,
                    "super property must be literal",
                ))
            }
        };

        let super_class = self.super_class.unwrap();

        // if in constructor, super means the class itself
```

```
if self.is_in_constructor {
    // get the class
    if let Some(cl) = self.context.classes.get(&super_class) {
        // find static property
        if let Some((vid, ty)) = cl.static_properties.get(&prop) {
            return Ok((
                Expr::VarLoad {
                    span: s.span,
                    variable: *vid,
                },
                ty.clone(),
            ));
        }
        // find static functions
        if let Some((fid, ty)) = cl.static_methods.get(&prop) {
            return Ok((Expr::Function(*fid), Type::Function(Box::new(ty.clone()))));
        }

        if let Some(_) = cl.static_generic_methods.get(&prop) {
            return Err(Error::syntax_error(s.span, "missing type arguments"));
        }
    } else {
        // the class should be defined
        unreachable!()
    }

    // the super class has no static property
    return Err(Error::syntax_error(
        s.span,
        format!("super has no property '{}'", prop),
    ));
}
```

```
    // context is in method
    if let Some(ty) = self.type_has_property(&Type::Object(super_class), &prop, false) {
        // return member expression
        return Ok((
            Expr::Member {
                object: Box::new(
                    // cast this to super
                    Expr::Cast(Box::new(Expr::This), Type::Object(super_class)),
                ),
                key: PropNameOrExpr::PropName(prop),
                optional: false,
            },
            ty,
        ));
    }

    return Err(Error::syntax_error(
        s.span,
        format!("super has no property '{}'", prop),
    ));
}

pub fn translate_new_expr(&mut self, n: &swc::NewExpr) -> Result<(Expr, Type)> {
    let ident = match n.callee.as_ident() {
        Some(ident) => ident,
        None => return Err(Error::syntax_error(n.callee.span(), "expected identifier")),
    };

    let mut arguments = Vec::new();

    match self.context.find(&ident.sym) {
```

```
Some(Binding::Class(class_id)) => {
    // should have no type arguments
    if let Some(args) = &n.type_args {
        return Err(Error::syntax_error(args.span, "expected 0 type arguments"));
    }

    let class_id = *class_id;

    let c = self.context.classes.get(&class_id).expect("invalid class");

    if let Some((_const_id, const_ty)) = &c.constructor {
        let params = const_ty.params.clone();

        if let Some(args) = &n.args {
            if args.len() != const_ty.params.len() {
                return Err(Error::syntax_error(
                    n.span,
                    format!(
                        "expected {} arguments, {} were given",
                        const_ty.params.len(),
                        args.len()
                    ),
                ));
            }

            for (i, arg) in args.iter().enumerate() {
                if let Some(spread) = arg.spread {
                    return Err(Error::syntax_error(
                        spread,
                        "spread argument is not supported",
                    ));
                }
```

```
                let (arg, _) = self.translate_expr(&arg.expr, params.get(i))?;
                arguments.push(arg);
            }
        } else {
            if const_ty.params.len() != 0 {
                return Err(Error::syntax_error(
                    n.span,
                    format!("expected {} arguments", const_ty.params.len()),
                ));
            }
        };
    } else {
        if n.args.as_ref().is_some_and(|a| a.len() != 0) {
            return Err(Error::syntax_error(n.span, "expected 0 arguments"));
        }
    };

    return Ok((
        Expr::New {
            class: class_id,
            args: arguments,
        },
        Type::Object(class_id),
    ));
}
Some(Binding::GenericClass(_class_id)) => {
    todo!("generic class")
}
_ => return Err(Error::syntax_error(ident.span, "expected class")),
};
}
```

```
pub fn translate_seq_expr(
    &mut self,
    s: &swc::SeqExpr,
    expected_ty: Option<&Type>,
) -> Result<(Expr, Type)> {
    if let Some(first) = s.exprs.get(0) {
        let first = self.translate_expr(
            &first,
            if s.exprs.len() == 1 {
                expected_ty
            } else {
                None
            },
        )?;

        if let Some(second) = s.exprs.get(1) {
            let second = self.translate_expr(
                &second,
                if s.exprs.len() == 2 {
                    expected_ty
                } else {
                    None
                },
            )?;

            let mut expr = Expr::Seq(Box::new(first.0), Box::new(second.0));
            let mut ty = second.1;

            let mut i = 2;
            while let Some(e) = s.exprs.get(i) {
                i += 1;
```

```
                let next = self.translate_expr(
                    e,
                    if i == s.exprs.len() {
                        expected_ty
                    } else {
                        None
                    },
                )?;

                expr = Expr::Seq(Box::new(expr), Box::new(next.0));
                ty = next.1;
            }

            return Ok((expr, ty));
        } else {
            return Ok(first);
        }
    };

    return Ok((Expr::Undefined, Type::Undefined));
}

pub fn translate_lit(
    &mut self,
    lit: &swc::Lit,
    expected_ty: Option<&Type>,
) -> Result<(Expr, Type)> {
    match lit {
        swc::Lit::BigInt(b) => {
            let i = b.value.to_i128().expect("i128 overflow");
            Ok((Expr::Bigint(i), Type::LiteralBigint(i)))
        }
```

```
        swc::Lit::Bool(b) => Ok((Expr::Bool(b.value), Type::LiteralBool(b.value))),
        swc::Lit::JSXText(_) => unimplemented!(),
        swc::Lit::Null(_) => Ok((Expr::Null, Type::Null)),
        swc::Lit::Num(n) => {
            if expected_ty == Some(&Type::Number) {
                return Ok((Expr::Number(n.value), Type::Number));
            }
            if n.value.is_finite() && n.value as i32 as f64 == n.value {
                return Ok((Expr::Int(n.value as i32), Type::LiteralInt(n.value as i32)));
            }

            return Ok((Expr::Number(n.value), Type::LiteralNumber(n.value)));
        }
        swc::Lit::Str(s) => Ok((
            Expr::String(s.value.to_string()),
            Type::LiteralString(s.value.as_str().into()),
        )),
        // todo: regex
        swc::Lit::Regex(_r) => Ok((Expr::Regex(), Type::Regex)),
    }
}

pub fn translate_optchain(&mut self, n: &swc::OptChainExpr) -> Result<(Expr, Type)> {
    match n.base.as_ref() {
        swc::OptChainBase::Member(m) => {
            let (mut expr, ty) = self.translate_member(m)?;
            if !n.optional {
                return Ok((expr, ty));
            }

            // set optional to true
            if let Expr::Member { optional, .. } = &mut expr {
```

```
        *optional = true;
    } else {
        unreachable!()
    }

    return Ok((expr, ty.union(Type::Undefined)));
}
swc::OptChainBase::Call(c) => {
    let mut callee = None;
    let mut func_ty = None;

    // function call
    if let Some(ident) = c.callee.as_ident() {
        match self.context.find(&ident.sym) {
            Some(Binding::Function(f)) => {
                if let Some(args) = &c.type_args {
                    return Err(Error::syntax_error(
                        args.span,
                        "expected 0 type arguments",
                    ));
                }

                let id = *f;
                let ty = self
                    .context
                    .functions
                    .get(&id)
                    .expect("invalid function")
                    .ty();

                self.type_check(ident.span, &self.this_ty, &ty.this_ty)?;
```

```
            callee = Some(Callee::Function(id));
            func_ty = Some(ty);
        }
        Some(Binding::GenericFunction(_id)) => {
            todo!("generic function")
        }
        _ => {}
    };
}

// expression
if callee.is_none() {
    let (expr, ty) = self.translate_expr(&c.callee, None)?;

    let func = match &ty {
        Type::Function(func_ty) => func_ty.as_ref().clone(),
        Type::Union(u) => {
            let mut func = None;

            for ty in u.iter() {
                match ty {
                    Type::Null | Type::Undefined => {}
                    Type::Function(f) => {
                        if func.is_some() {
                            return Err(Error::syntax_error(
                                c.callee.span(),
                                "type " is not callable",
                            ));
                        }
                        func = Some(f.as_ref().clone());
                    }
                    _ => {
```

```
                return Err(Error::syntax_error(
                    c.callee.span(),
                    "type " is not callable",
                ))
            }
        }
    }

    func.unwrap()
  }
  Type::Undefined => {
    // will never call
    return Ok((expr, Type::Undefined));
  }
  Type::Null => {
    // will never call
    return Ok((expr, Type::Null));
  }
  _ => {
    return Err(Error::syntax_error(
      c.callee.span(),
      "type " is not callable",
    ))
  }
};

// check this type matches
self.type_check(c.span, &self.this_ty, &func.this_ty)?;

// member call
if let Expr::Member {
  object,
```

```
        key,
        optional,
    } = expr
    {
        if optional {
            return Err(Error::syntax_error(c.span, "callee cannot be optional"));
        }
        callee = Some(Callee::Member {
            object: *object,
            prop: key,
        });
    } else {
        callee = Some(Callee::Expr(expr));
    }
    func_ty = Some(func);
};

let callee = callee.unwrap();
let func_ty = func_ty.unwrap();

let mut args = Vec::new();
let mut arg_tys = Vec::new();

// translate arguments
for (i, arg) in c.args.iter().enumerate() {
    if let Some(spread) = arg.spread {
        return Err(Error::syntax_error(
            spread,
            "variabl arguments not supported",
        ));
    }
    let (expr, ty) = self.translate_expr(&arg.expr, func_ty.params.get(i))?;
```

```
                args.push(expr);
                arg_tys.push(ty);
            }

            // length must be the same
            if args.len() != func_ty.params.len() {
                return Err(Error::syntax_error(
                    c.span,
                    format!(
                        "expected {} arguments, {} were given",
                        func_ty.params.len(),
                        args.len()
                    ),
                ));
            }

            // return call expression
            return Ok((
                Expr::Call {
                    callee: Box::new(callee),
                    args: args,
                    optional: true,
                },
                func_ty.return_ty,
            ));
        }
    }
}

pub fn translate_prop_name(&mut self, name: &swc::PropName) -> Result<PropNameOrExpr>
{
    match name {
```

```
            swc::PropName::BigInt(b) => Ok(PropNameOrExpr::PropName(crate::PropName::String(
                b.value.to_string(),
            ))),
            swc::PropName::Ident(id) => Ok(PropNameOrExpr::PropName(crate::PropName::Ident(
                id.sym.to_string(),
            ))),
            swc::PropName::Num(n) => {
                if n.value as i32 as f64 == n.value {
                    Ok(PropNameOrExpr::PropName(crate::PropName::Int(
                        n.value as i32,
                    )))
                } else {
                    Ok(PropNameOrExpr::PropName(crate::PropName::String(
                        n.value.to_string(),
                    )))
                }
            }
            swc::PropName::Str(s) => Ok(PropNameOrExpr::PropName(crate::PropName::String(
                s.value.to_string(),
            ))),
            swc::PropName::Computed(c) => self.translate_computed_prop_name(&c.expr),
        }
    }

    pub fn translate_computed_prop_name(&mut self, expr: &swc::Expr) ->
Result<PropNameOrExpr> {
        match expr {
            swc::Expr::PrivateName(n) => {
                return Ok(PropNameOrExpr::PropName(PropName::Private(
                    n.id.sym.to_string(),
                )))
            }
```

```
swc::Expr::Lit(l) => {
    match l {
        swc::Lit::BigInt(b) => {
            return Ok(PropNameOrExpr::PropName(crate::PropName::String(
                b.value.to_string(),
            )))
        }
        swc::Lit::Bool(b) => {
            return Ok(PropNameOrExpr::PropName(crate::PropName::String(
                b.value.to_string(),
            )))
        }
        swc::Lit::JSXText(j) => {
            return Err(Error::syntax_error(j.span, "JSXText is not allowed"))
        }
        swc::Lit::Null(_) => {
            return Ok(PropNameOrExpr::PropName(crate::PropName::String(
                "null".to_string(),
            )))
        }
        swc::Lit::Num(n) => {
            if n.value as i32 as f64 == n.value {
                return Ok(PropNameOrExpr::PropName(crate::PropName::Int(
                    n.value as i32,
                )));
            } else {
                return Ok(PropNameOrExpr::PropName(crate::PropName::String(
                    n.value.to_string(),
                )));
            };
        }
        swc::Lit::Regex(r) => {
```

```
            return Ok(PropNameOrExpr::PropName(crate::PropName::String(format!(
                "/{}/{}",
                r.exp, r.flags
            ))))
        }
        swc::Lit::Str(s) => {
            return Ok(PropNameOrExpr::PropName(crate::PropName::String(
                s.value.to_string(),
            )))
        }
    };
}
swc::Expr::Member(mem) => match mem.obj.as_ref() {
    swc::Expr::Ident(id) => {
        if id.sym.as_ref() == "Symbol" {
            if let Some(prop) = mem.prop.as_ident() {
                match prop.sym.as_ref() {
                    "asyncIterator" => {
                        return Ok(PropNameOrExpr::PropName(crate::PropName::Symbol(
                            crate::Symbol::AsyncIterator,
                        )))
                    }
                    "hasInstance" => {
                        return Ok(PropNameOrExpr::PropName(PropName::Symbol(
                            crate::Symbol::HasInstance,
                        )))
                    }
                    "isConcatSpreadable" => {
                        return Ok(PropNameOrExpr::PropName(PropName::Symbol(
                            crate::Symbol::IsConcatSpreadable,
                        )))
                    }
```

```
"iterator" => {
    return Ok(PropNameOrExpr::PropName(PropName::Symbol(
        crate::Symbol::Iterator,
    )))
}
"match" => {
    return Ok(PropNameOrExpr::PropName(PropName::Symbol(
        crate::Symbol::Match,
    )))
}
"matchAll" => {
    return Ok(PropNameOrExpr::PropName(PropName::Symbol(
        crate::Symbol::MatchAll,
    )))
}
"replace" => {
    return Ok(PropNameOrExpr::PropName(PropName::Symbol(
        crate::Symbol::Replace,
    )))
}
"search" => {
    return Ok(PropNameOrExpr::PropName(PropName::Symbol(
        crate::Symbol::Search,
    )))
}
"species" => {
    return Ok(PropNameOrExpr::PropName(PropName::Symbol(
        crate::Symbol::Species,
    )))
}
"split" => {
    return Ok(PropNameOrExpr::PropName(PropName::Symbol(
```

```
                        crate::Symbol::Split,
                    )))
                }
                "toStringTag" => {
                    return Ok(PropNameOrExpr::PropName(PropName::Symbol(
                        crate::Symbol::ToStringTag,
                    )))
                }
                "unscopables" => {
                    return Ok(PropNameOrExpr::PropName(PropName::Symbol(
                        crate::Symbol::Unscopables,
                    )))
                }
                "dispose" => {
                    return Ok(PropNameOrExpr::PropName(PropName::Symbol(
                        crate::Symbol::Dispose,
                    )))
                }
                "asyncDispose" => {
                    return Ok(PropNameOrExpr::PropName(PropName::Symbol(
                        crate::Symbol::AsyncDispose,
                    )))
                }
                _ => {}
            }
        }
    }
    _ => {}
  },
  _ => {}
};
```

```
        let (expr, ty) = self.translate_expr(expr, None)?;


        return Ok(PropNameOrExpr::Expr(Box::new(expr), ty));
    }
}
```

5.2.14 native-ts-hir/transform/context.rs

```
use std::collections::HashMap;

use crate::ast::*;
use crate::common::{
AliasId, ClassId, EnumId, FunctionId, GenericId, InterfaceId, ModuleId,
VariableId,
};

#[derive(Clone)]
pub enum Binding {
/// a variable, can be let, var or const
Var {
/// const is not writable
writable: bool,
/// var can be redeclared
redeclarable: bool,
id: VariableId,
ty: Type,
},
Using {
id: VariableId,
ty: Type,
is_await: bool,
},

GenericFunction(FunctionId),
Function(FunctionId),

GenericClass(ClassId),
Class(ClassId),

GenericInterface(InterfaceId),
Interface(InterfaceId),

Enum(EnumId),
```

```
GenericTypeAlias(AliasId),
TypeAlias(AliasId),

NameSpace(ModuleId),

Generic(GenericId),
}

pub struct Scope {
/// the id of function where the scope belongs to
pub function_id: FunctionId,
/// generic classes
pub bindings: HashMap<String, Binding>,
}

pub struct GenericRegister<ID, BASE> {
pub resolved: HashMap<u64, ID>,
pub ty: BASE,
}

pub struct Context {
pub generic_classes: HashMap<ClassId, GenericRegister<ClassId, ()>>,
pub classes: HashMap<ClassId, ClassType>,

pub generic_functions: HashMap<FunctionId, GenericRegister<FunctionId,
GenericFunction>>,
pub functions: HashMap<FunctionId, Function>,

pub generic_interfaces: HashMap<InterfaceId, GenericRegister<InterfaceId,
()>>,
pub interfaces: HashMap<InterfaceId, InterfaceType>,

pub enums: HashMap<EnumId, EnumType>,

pub generic_alias: HashMap<AliasId, ()>,
pub alias: HashMap<AliasId, Type>,

global_func: FunctionId,
scopes: Vec<Scope>,
}

impl Context {
pub fn new() -> Self {
```

```
// main function id
let function_id = FunctionId::new();

// construct scope
let global_scope = Scope {
function_id,
bindings: Default::default(),
};

// construct context
let mut s = Self {
generic_classes: Default::default(),
classes: Default::default(),
generic_functions: Default::default(),
functions: Default::default(),
generic_interfaces: Default::default(),
interfaces: Default::default(),
enums: Default::default(),
generic_alias: Default::default(),
alias: Default::default(),
global_func: function_id,
scopes: vec![global_scope],
};

// insert main function
s.functions.insert(
function_id,
Function {
this_ty: Type::Any,
params: Vec::new(),
return_ty: Type::Undefined,
variables: Default::default(),
captures: Vec::new(),
stmts: Vec::new(),
},
);
return s;
}

pub fn new_function(&mut self, id: FunctionId) {
self.scopes.push(Scope {
function_id: id,
bindings: HashMap::new(),
});
```

```
if !self.functions.contains_key(&id) {
self.functions.insert(
id,
Function {
this_ty: Type::Any,
params: Vec::new(),
return_ty: Type::Undefined,
variables: HashMap::new(),
captures: Default::default(),
stmts: Vec::new(),
},
);
}
}

pub fn end_function(&mut self) -> FunctionId {
let poped_scope = self.scopes.pop().expect("failed to close scope");

// check that the last scope is not owned by current function
if let Some(scope) = self.scopes.last() {
assert!(
scope.function_id != poped_scope.function_id,
"improper closing scope"
);
}

let func = self
.functions
.get_mut(&poped_scope.function_id)
.expect("invalid function");

for (_name, binding) in &poped_scope.bindings {
match binding {
Binding::Var { id, ty: _, .. } | Binding::Using { id, .. } => {
func.stmts.push(Stmt::DropVar(*id));
}
_ => {}
}
}

return poped_scope.function_id;
}
```

```
pub fn func(&mut self) -> &mut Function {
let scope = self.scopes.last().expect("invalid scope");

return self
.functions
.get_mut(&scope.function_id)
.expect("invalid function");
}

pub fn new_scope(&mut self) {
let func = self.scopes.last().unwrap().function_id;

self.scopes.push(Scope {
function_id: func,
bindings: HashMap::new(),
});
}

pub fn end_scope(&mut self) {
let poped_scope = self.scopes.pop().expect("failed to pop scope");

let func = self
.functions
.get_mut(&poped_scope.function_id)
.expect("invalid function");

for (_name, binding) in &poped_scope.bindings {
match binding {
Binding::Var { id, ty: _, .. } | Binding::Using { id, .. } => {
func.stmts.push(Stmt::DropVar(*id));
}
_ => {}
}
}
}

pub fn declare_global(&mut self, id: VariableId, ty: Type) {
self.functions
.get_mut(&self.global_func)
.expect("invalid function")
.variables
.insert(
id,
VariableDesc {
```

```
ty: ty,
is_heap: false,
is_captured: false,
},
);
}

pub fn declare(&mut self, name: &str, binding: Binding) -> bool {
let scope = self.scopes.last_mut().expect("invalid scope");

if let Some(bind) = scope.bindings.get(name) {
if let Binding::Var {
redeclarable,
id: varid,
..
} = bind
{
if *redeclarable {
// it is only valid if new binding is both writable and redeclarable
match &binding {
Binding::Var {
writable: true,
redeclarable: true,
..
} => {
// copy the old variable id
let varid = *varid;
// replace the current binding
scope.bindings.insert(name.to_string(), binding);
// drop the variable
self.func().stmts.push(Stmt::DropVar(varid));
return true;
}
_ => {}
}
}
};

return false;
}

match &binding {
Binding::Var { id, ty, .. } | Binding::Using { id, ty, .. } => {
let id = *id;
```

```
let ty = ty.clone();

self.func().variables.insert(
id,
VariableDesc {
ty: ty,
is_heap: false,
is_captured: false,
},
);
}
_ => {}
}

self.scopes
.last_mut()
.expect("invalide scope")
.bindings
.insert(name.to_string(), binding);

return true;
}

pub fn find(&mut self, name: &str) -> Option<&Binding> {
let current_func_id = self.scopes.last().unwrap().function_id;

for scope in self.scopes.iter().rev() {
if let Some(bind) = scope.bindings.get(name) {
match bind {
Binding::Var { id, .. } => {
// set variable to heap
if scope.function_id != current_func_id {
// get the function that owns the
let func = self
.functions
.get_mut(&scope.function_id)
.expect("invalid function");

// set the variable to heap
let desc = func.variables.get_mut(&id).expect("invalide varaibel id");
// set variable to a heap variable
desc.is_heap = true;
}
```

```
// capture variable

return Some(bind);
}
Binding::Using { .. } => {
// only the owned scope can see the variable
if scope.function_id != current_func_id {
return None;
}
return Some(bind);
}
_ => return Some(bind),
}
}
}

return None;
}

pub fn get_func_id(&self, name: &str) -> FunctionId {
match self.scopes.last().unwrap().bindings.get(name) {
Some(Binding::Function(id)) => *id,
Some(Binding::GenericFunction(id)) => *id,
_ => unreachable!(),
}
}
pub fn get_class_id(&self, name: &str) -> ClassId {
match self.scopes.last().unwrap().bindings.get(name) {
Some(Binding::Class(id)) => *id,
Some(Binding::GenericClass(id)) => *id,
_ => unreachable!(),
}
}
pub fn get_interface_id(&self, name: &str) -> InterfaceId {
match self.scopes.last().unwrap().bindings.get(name) {
Some(Binding::Interface(id)) => *id,
Some(Binding::GenericInterface(id)) => *id,
_ => unreachable!(),
}
}
}
```

5.2.15 native-ts-hir/transform/class.rs

```
use native_js_common::error::Error;
```

```rust
use native_ts_parser::swc_core::common::{Span, Spanned};
use native_ts_parser::swc_core::ecma::ast as swc;

use super::Transformer;

use crate::ast::{
ClassType, Expr, FuncType, FunctionParam, PropNameOrExpr, PropertyDesc,
Stmt, Type,
};
use crate::common::{ClassId, FunctionId, VariableId};
use crate::transform::context::Binding;

type Result<T> = std::result::Result<T, Error<Span>>;

impl Transformer {
/// this function only translates type definitions and does not translate any
initialiser or methods
pub fn translate_class_ty(&mut self, id: ClassId, class: &swc::Class) ->
Result<ClassType> {
let mut class_ty = ClassType::default();

if let Some(_type_params) = &class.type_params {
todo!("generic class")
}

// translate super class
if let Some(super_expr) = &class.super_class {
// translate super args
let super_args = if let Some(super_type_args) = &class.super_type_params {
self.translate_type_args(&super_type_args)?
} else {
Vec::new()
};
let super_ty = self.translate_expr_type(&super_expr, &super_args)?;

match super_ty {
Type::Object(class_id) => {
class_ty.extends = Some(class_id);
}
_ => {
return Err(Error::syntax_error(
super_expr.span(),
"class can only extend a class",
```

```
))
}
}
}

for member in &class.body {
match member {
swc::ClassMember::AutoAccessor(a) => {
return Err(Error::syntax_error(a.span, "auto accessor not supported"))
}
swc::ClassMember::Empty(_) => {}
// ignore private method as it is not exposed
swc::ClassMember::PrivateMethod(_) => {}
// ignore private prop as it is not exposed
swc::ClassMember::PrivateProp(_) => {}
// ignored as it is not evaluated
swc::ClassMember::StaticBlock(_) => {}
// index signature will not be supported
swc::ClassMember::TsIndexSignature(i) => {
return Err(Error::syntax_error(
i.span,
"index signature not allowed, use a map instead",
))
}
swc::ClassMember::Constructor(c) => {
// create function id
let dummy_id = FunctionId::new();
// create function type
let mut constructor_ty = FuncType {
this_ty: Type::Object(id),
params: Vec::new(),
var_arg: false,
return_ty: Type::Undefined,
};

// translate params
for p in &c.params {
match p {
swc::ParamOrTsParamProp::Param(p) => {
// identifier binding
if let Some(ident) = p.pat.as_ident() {
// translate type annotation
if let Some(ann) = &ident.type_ann {
// translate type
```

```
let mut ty = self.translate_type(&ann.type_ann)?;

if ident.optional {
ty = ty.union(Type::Any);
}
// push param to type
constructor_ty.params.push(ty);
} else {
// missing type annotation
return Err(Error::syntax_error(
ident.span,
"missing type annotation",
));
}
} else if let Some(r) = p.pat.as_rest() {
// variable argument
return Err(Error::syntax_error(
r.dot3_token,
"variable arguments not supported",
));
} else {
// destructive binding not supported
return Err(Error::syntax_error(
p.span,
"destructive param not supported",
));
}
}
swc::ParamOrTsParamProp::TsParamProp(p) => {
match &p.param {
swc::TsParamPropParam::Assign(a) => {
// default argument not supported
return Err(Error::syntax_error(
a.span,
"default param not supported",
));
}
swc::TsParamPropParam::Ident(ident) => {
// translate type annotation
if let Some(ann) = &ident.type_ann {
// translate type
let mut ty = self.translate_type(&ann.type_ann)?;

if ident.optional {
```

```
ty = ty.union(Type::Any);
}
// push param type
constructor_ty.params.push(ty);
} else {
// missing type annotation
return Err(Error::syntax_error(
ident.span,
"missing type annotation",
));
}
}
}
}
}
}

// set constructor type
class_ty.constructor = Some((dummy_id, constructor_ty));
}
swc::ClassMember::ClassProp(prop) => {
let ty = if let Some(ann) = &prop.type_ann {
let ty = self.translate_type(&ann.type_ann)?;
// optional
if prop.is_optional {
ty.union(Type::Undefined)
} else {
ty
}
} else {
Type::Any
};

let name = self.translate_prop_name(&prop.key)?;
let name = match name {
PropNameOrExpr::Expr(_, _) => {
return Err(Error::syntax_error(
prop.key.span(),
"dynamic propname is not allowed",
))
}
PropNameOrExpr::PropName(p) => p,
};
```

```
if prop.is_static {
let id = VariableId::new();
self.context.declare_global(id, ty.clone());
class_ty.static_properties.insert(name, (id, ty));
} else {
class_ty.properties.insert(
name,
PropertyDesc {
ty: ty,
readonly: prop.readonly,
initialiser: None,
},
);
}
}
swc::ClassMember::Method(m) => {
let mut func_ty = self.translate_function_ty(&m.function)?;
let prop = self.translate_prop_name(&m.key)?;
let prop = match prop {
PropNameOrExpr::Expr(..) => {
return Err(Error::syntax_error(
m.key.span(),
"dynamic prop name not allowed",
))
}
PropNameOrExpr::PropName(p) => p,
};

if m.is_optional {
return Err(Error::syntax_error(
m.span,
"class method cannot be optional",
));
}

if m.is_static {
class_ty
.static_methods
.insert(prop, (FunctionId::new(), func_ty));
} else {
func_ty.this_ty = Type::Object(id);
class_ty.methods.insert(prop, (FunctionId::new(), func_ty));
}
}
```

```
}
}

return Ok(class_ty);
}

pub fn translate_class(&mut self, id: ClassId, name: String, class: &swc::Class) -
> Result<()> {
let mut class_ty = self
.context
.classes
.get(&id)
.cloned()
.unwrap_or(ClassType::default());

class_ty.name = name;

if let Some(_type_params) = &class.type_params {
todo!("generic class")
}

// translate super class
if let Some(super_expr) = &class.super_class {
// translate super args
let super_args = if let Some(super_type_args) = &class.super_type_params {
self.translate_type_args(&super_type_args)?
} else {
Vec::new()
};
let super_ty = self.translate_expr_type(&super_expr, &super_args)?;

match super_ty {
Type::Object(class_id) => {
class_ty.extends = Some(class_id);
}
_ => {
return Err(Error::syntax_error(
super_expr.span(),
"class can only extend a class",
))
}
}
}
}
```

```
for i in &class.implements {
// translate super args
let type_args = if let Some(type_args) = &i.type_args {
self.translate_type_args(&type_args)?
} else {
Vec::new()
};
let impl_ty = self.translate_expr_type(&i.expr, &type_args)?;

match impl_ty {
Type::Interface(iface) => {
class_ty.implements.push(iface);
// pend a future type check
self.type_checks.push(super::TypeCheck {
span: i.span,
ty: Type::Object(id),
fulfills: Type::Interface(iface),
});
}
_ => {
return Err(Error::syntax_error(
i.span,
"class can only implement interfaces",
))
}
}
}

for member in &class.body {
match member {
swc::ClassMember::AutoAccessor(a) => {
// auto accessor not supported
return Err(Error::syntax_error(a.span, "auto accessor not supported"));
}
swc::ClassMember::TsIndexSignature(i) => {
// index signature not supported
return Err(Error::syntax_error(
i.span,
"index signature not allowed, use a map instead",
));
}
swc::ClassMember::Empty(_) => {}
swc::ClassMember::StaticBlock(block) => {
// translate block directly
```

```
self.translate_block_stmt(&block.body, None)?;
}
swc::ClassMember::ClassProp(prop) => {
let ty = if let Some(ann) = &prop.type_ann {
let ty = self.translate_type(&ann.type_ann)?;
// optional
if prop.is_optional {
ty.union(Type::Undefined)
} else {
ty
}
} else {
Type::Any
};

let name = self.translate_prop_name(&prop.key)?;
let name = match name {
PropNameOrExpr::Expr(_, _) => {
return Err(Error::syntax_error(
prop.key.span(),
"dynamic propname is not allowed",
))
}
PropNameOrExpr::PropName(p) => p,
};

let init = if let Some(e) = &prop.value {
let (e, _) = self.translate_expr(e, Some(&ty))?;
Some(e)
} else {
None
};

if prop.is_static {
let id = VariableId::new();
self.context.declare_global(id, ty.clone());
class_ty.static_properties.insert(name, (id, ty));

if let Some(init) = init {
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: id,
value: Box::new(init),
})));
```

```
}
} else {
class_ty.properties.insert(
name,
PropertyDesc {
ty: ty,
readonly: prop.readonly,
initialiser: init,
},
);
}
}
swc::ClassMember::PrivateProp(prop) => {
let ty = if let Some(ann) = &prop.type_ann {
let ty = self.translate_type(&ann.type_ann)?;
// optional
if prop.is_optional {
ty.union(Type::Undefined)
} else {
ty
}
} else {
Type::Any
};

let name = crate::PropName::Private(prop.key.id.sym.to_string());

let init = if let Some(e) = &prop.value {
let (e, _) = self.translate_expr(e, Some(&ty))?;
Some(e)
} else {
None
};

if prop.is_static {
let id = VariableId::new();
self.context.declare_global(id, ty.clone());
class_ty.static_properties.insert(name, (id, ty));

if let Some(init) = init {
self.context.func().stmts.push(Stmt::Expr(Box::new(Expr::VarAssign {
op: crate::ast::AssignOp::Assign,
variable: id,
value: Box::new(init),
```

```
})));
}
} else {
class_ty.properties.insert(
name,
PropertyDesc {
ty: ty,
readonly: prop.readonly,
initialiser: init,
},
);
}
}
swc::ClassMember::Constructor(c) => {
// use declared id or else create a new id
let constructor_id = class_ty
.constructor
.as_ref()
.map(|(id, _)| *id)
.unwrap_or(FunctionId::new());
// create function type
let mut constructor_ty = FuncType {
this_ty: Type::Object(id),
params: Vec::new(),
var_arg: false,
return_ty: Type::Undefined,
};

let old_this = core::mem::replace(&mut self.this_ty, Type::Object(id));
let old_return = core::mem::replace(&mut self.return_ty, Type::Undefined);
let old_is_constructor = core::mem::replace(&mut self.is_in_constructor, true);

self.context.new_function(constructor_id);

// translate params
for p in &c.params {
match p {
swc::ParamOrTsParamProp::Param(p) => {
// identifier binding
if let Some(ident) = p.pat.as_ident() {
// translate type annotation
if let Some(ann) = &ident.type_ann {
// translate type
let mut ty = self.translate_type(&ann.type_ann)?;
```

```
if ident.optional {
ty = ty.union(Type::Any);
}
// push param to type
constructor_ty.params.push(ty.clone());

// create new id
let param_id = VariableId::new();
// push param
self.context.func().params.push(FunctionParam {
ty: ty.clone(),
id: param_id,
});
self.context.declare(
&ident.sym,
Binding::Var {
writable: true,
redeclarable: true,
id: param_id,
ty: ty,
},
);
} else {
// missing type annotation
return Err(Error::syntax_error(
ident.span,
"missing type annotation",
));
}
} else if let Some(r) = p.pat.as_rest() {
// variable argument
return Err(Error::syntax_error(
r.dot3_token,
"variable arguments not supported",
));
} else {
// destructive binding not supported
return Err(Error::syntax_error(
p.span,
"destructive param not supported",
));
}
}
```

```
swc::ParamOrTsParamProp::TsParamProp(p) => {
match &p.param {
swc::TsParamPropParam::Assign(a) => {
// default argument not supported
return Err(Error::syntax_error(
a.span,
"default param not supported",
));
}
swc::TsParamPropParam::Ident(ident) => {
// translate type annotation
if let Some(ann) = &ident.type_ann {
// translate type
let mut ty = self.translate_type(&ann.type_ann)?;

if ident.optional {
ty = ty.union(Type::Any);
}
// push param type
constructor_ty.params.push(ty.clone());

// create new id
let param_id = VariableId::new();
// push param
self.context.func().params.push(FunctionParam {
ty: ty.clone(),
id: param_id,
});
self.context.declare(
&ident.sym,
Binding::Var {
writable: true,
redeclarable: true,
id: param_id,
ty: ty,
},
);
} else {
// missing type annotation
return Err(Error::syntax_error(
ident.span,
"missing type annotation",
));
}
```

```
}
}
}
}
}

if let Some(body) = &c.body {
self.translate_block_stmt(body, None)?;
} else {
return Err(Error::syntax_error(c.span, "missing constructor body"));
}

let func_id = self.context.end_function();
debug_assert!(func_id == constructor_id);

self.this_ty = old_this;
self.return_ty = old_return;
self.is_in_constructor = old_is_constructor;

// set constructor type
class_ty.constructor = Some((constructor_id, constructor_ty));
}
swc::ClassMember::Method(m) => {
let prop = self.translate_prop_name(&m.key)?;
let prop = match prop {
PropNameOrExpr::Expr(..) => {
return Err(Error::syntax_error(
m.key.span(),
"dynamic prop name not allowed",
))
}
PropNameOrExpr::PropName(p) => p,
};

let method_id = if m.is_static {
class_ty
.static_methods
.get(&prop)
.map(|(i, _)| *i)
.unwrap_or(FunctionId::new())
} else {
class_ty
.methods
.get(&prop)
```

```
.map(|(i, _)| *i)
.unwrap_or(FunctionId::new())
};

if m.is_optional {
return Err(Error::syntax_error(
m.span,
"class method cannot be optional",
));
}

self.translate_function(method_id, Some(Type::Object(id)), &m.function)?;
let func_ty = self
.context
.functions
.get(&method_id)
.expect("invalid function")
.ty();

if m.is_static {
class_ty.static_methods.insert(prop, (method_id, func_ty));
} else {
class_ty.methods.insert(prop, (method_id, func_ty));
}
}
swc::ClassMember::PrivateMethod(m) => {
let prop = crate::PropName::Private(m.key.id.sym.to_string());

let method_id = if m.is_static {
class_ty
.static_methods
.get(&prop)
.map(|(i, _)| *i)
.unwrap_or(FunctionId::new())
} else {
class_ty
.methods
.get(&prop)
.map(|(i, _)| *i)
.unwrap_or(FunctionId::new())
};

if m.is_optional {
return Err(Error::syntax_error(
```

```
m.span,
"class method cannot be optional",
));
}

self.translate_function(method_id, Some(Type::Object(id)), &m.function)?;
let func_ty = self
.context
.functions
.get(&method_id)
.expect("invalid function")
.ty();

if m.is_static {
class_ty.static_methods.insert(prop, (method_id, func_ty));
} else {
class_ty.methods.insert(prop, (method_id, func_ty));
}
}
}
}

self.context.classes.insert(id, class_ty);
return Ok(());
}
}
```

5.2.16 native-ts-hir/tests/loop_transform.rs

```
use native_ts_hir::ast::format::Formatter;
use native_ts_hir::transform::Transformer;

#[test]
fn test_for_in_loop() {
let s = r#"
for (let i in []){
i += (99)
}
"#;
let parser = native_ts_parser::Parser::new();
let m = parser
.parse_str("test".to_string(), s.to_string())
.expect("parse failed");
```

```
for (_id, module) in m.modules {
let mut t = Transformer::new();

let re = t.transform_module(&module.module).expect("parse error");
let mut formatter = Formatter::new(&re.table);
formatter.format_module(&re);

let formated = formatter.emit_string();
println!("{}", formated);
}
}

#[test]
fn test_for_of_loop() {
let s = r#"
for (let i of [0, 9, 8]){
i+=(99);
}
"#;
let parser = native_ts_parser::Parser::new();
let m = parser
.parse_str("test".to_string(), s.to_string())
.expect("parse failed");

for (_id, module) in m.modules {
let mut t = Transformer::new();

let re = t.transform_module(&module.module).expect("parse error");
let mut formatter = Formatter::new(&re.table);
formatter.format_module(&re);

let formated = formatter.emit_string();
println!("{}", formated);
}
}
```

5.2.17 native-ts-hir/tests/binary_search.rs

```
use native_ts_hir::ast::format::Formatter;
use native_ts_hir::transform::Transformer;

#[test]
fn binary_search() {
let s = "binarySearch([], 0);
function binarySearch(arr: number[], x: number): number
```

```
{
let l = 0;
let r = arr.length - 1;
let mid: number;
while (r >= l) {
mid = l + (r - l) / 2;
// If the element is present at the middle
// itself
if (arr[mid] == x)
return mid;
// If element is smaller than mid, then
// it can only be present in left subarray
if (arr[mid] > x)
r = mid - 1;

// Else the element can only be present
// in right subarray
else
l = mid + 1;
}
// We reach here when element is not
// present in array
return -1;
}";

let parser = native_ts_parser::Parser::new();
let m = parser
.parse_str("test".to_string(), s.to_string())
.expect("parse failed");

for (_id, module) in m.modules {
let mut t = Transformer::new();

let re = t.transform_module(&module.module).expect("parse error");
let mut formatter = Formatter::new(&re.table);
formatter.format_module(&re);

let formated = formatter.emit_string();
println!("{}", formated);
}
}
```

5.2.18 native-ts-hir/interpreter/mod.rs

use std::collections::{HashMap, HashSet};

```
use std::sync::Arc;

use parking_lot::RwLock;

use crate::ast::*;
use crate::common::{ClassId, EnumId, FunctionId, InterfaceId, ModuleId, VariableId};
use crate::symbol_table::SymbolTable;
use crate::PropName;
use crate::Symbol;

#[derive(Debug, Clone)]
pub enum Value {
    Undefined,
    Null,
    Bool(bool),
    Int(i32),
    Number(f64),
    /// loads an i128
    Bigint(i128),
    /// loads a string
    String(String),
    Symbol(Symbol),
    Regex(),
    Function(FunctionId),
    Clousure {
        id: FunctionId,
        captures: Arc<HashMap<VariableId, Arc<RwLock<Value>>>>,
    },
    Array(Arc<RwLock<Vec<Value>>>),
    Tuple(Arc<RwLock<Vec<Value>>>),
    Interface {
        id: InterfaceId,
```

```
        value: Box<Value>,
    },
    Union {
        ty: Box<[Type]>,
        value: Box<Value>,
    },
    Any(Box<Value>),
    AnyObject(Box<Value>),
    Object {
        id: ClassId,
        values: Arc<[(PropName, Value)]>,
    },
    DynObject {
        values: Arc<[(PropName, Value)]>,
    },
    Enum {
        id: EnumId,
        variant: usize,
    },
}

#[derive(Debug)]
enum StmtResult {
    Ok,
    Return(Value),
    Break(Option<String>),
    Continue(Option<String>),
    Error(Value),
}

struct Context {
    variables: HashMap<VariableId, Value>,
```

```
    heap_variables: HashMap<VariableId, Arc<RwLock<Value>>>,
}

impl Context {
    pub fn new() -> Self {
        Self {
            variables: HashMap::new(),
            heap_variables: HashMap::new(),
        }
    }
    pub fn declare(&mut self, id: VariableId, is_heap: bool) {
        if is_heap {
            self.heap_variables
                .insert(id, Arc::new(RwLock::new(Value::Undefined)));
        } else {
            self.variables.insert(id, Value::Undefined);
        }
    }

    pub fn read(&self, id: VariableId) -> Value {
        if let Some(v) = self.variables.get(&id) {
            return v.clone();
        }
        if let Some(v) = self.heap_variables.get(&id) {
            return v.read().clone();
        }
        panic!("invalid variable id")
    }
    pub fn write(&mut self, id: VariableId, value: Value) {
        if let Some(v) = self.variables.get_mut(&id) {
            *v = value;
            return;
```

```
        }
        if let Some(v) = self.heap_variables.get(&id) {
            *v.write() = value;
            return;
        }
        panic!("invalid variable id")
    }

    pub fn contains(&self, id: VariableId) -> bool {
        self.variables.contains_key(&id) || self.heap_variables.contains_key(&id)
    }

    pub fn remove(&mut self, id: VariableId) -> bool {
        if self.variables.remove(&id).is_none() {
            return self.variables.remove(&id).is_some();
        }

        return true;
    }
}

pub struct Interpreter {}

impl Interpreter {
    pub fn new() -> Self {
        Self {}
    }
    pub fn run(&self, program: &Program) -> Result<Value, Value> {
        let entry = program.modules.get(&program.entry).expect("invalid module");

        let mut runner = InterpreterRunning {
            table: &entry.table,
```

```
            modules: &program.modules,

            ran_modules: HashSet::new(),

            this: Value::Undefined,
            pc: Value::Undefined,
            stack: Vec::new(),
            context: Context::new(),
        };

        runner.run_module_with_dependencies(program.entry)
    }
}

struct InterpreterRunning<'a> {
    table: &'a SymbolTable,
    modules: &'a HashMap<ModuleId, Module>,

    ran_modules: HashSet<ModuleId>,

    this: Value,
    pc: Value,
    stack: Vec<Value>,

    context: Context,
}

impl<'a> InterpreterRunning<'a> {
    fn run_module_with_dependencies(&mut self, id: ModuleId) -> Result<Value, Value> {
        let module = self.modules.get(&id).expect("invalid module");

        for dep in &module.dependencies {
```

```
        if self.ran_modules.insert(*dep) {
            self.run_module_with_dependencies(*dep)?;
        }
    }

    self.run_module(module)
}

fn run_module(&mut self, module: &Module) -> Result<Value, Value> {
    let main = self
        .table
        .functions
        .get(&module.main_function)
        .expect("invalid function");

    self.run_function(
        main,
        Value::Any(Box::new(Value::Undefined)),
        &[],
        Context::new(),
    )?;

    return Ok(self.pc.clone());
}

fn run_function(
    &mut self,
    func: &Function,
    this: Value,
    args: &[Value],
    new_context: Context,
) -> Result<Value, Value> {
```

```
let old_context = core::mem::replace(&mut self.context, new_context);

// assert this type
self.assert_type(&this, &func.this_ty);

assert!(func.params.len() == args.len());

for (id, desc) in &func.variables {
    self.context.declare(*id, desc.is_heap);
}

for i in 0..args.len() {
    let param = &func.params[i];
    self.assert_type(&args[i], &param.ty);

    self.context.write(param.id, args[i].clone());
}

for (id, _) in &func.captures {
    if !self.context.heap_variables.contains_key(id) {
        self.context.heap_variables.insert(
            *id,
            old_context
                .heap_variables
                .get(id)
                .expect("missing capture")
                .clone(),
        );
    }
}

let mut cursor = 0;
```

```
    match self.run_stmts(&func.stmts, &mut cursor) {
        StmtResult::Ok => {
            // restore context
            self.context = old_context;
        }
        StmtResult::Continue(_) => panic!("invalid continue"),
        StmtResult::Break(_) => panic!("invalid break"),
        StmtResult::Return(r) => {
            // restore context
            self.context = old_context;
            return Ok(r);
        }
        StmtResult::Error(e) => {
            // restore context
            self.context = old_context;
            return Err(e);
        }
    }

    return Ok(Value::Undefined);
}

fn run_stmts(&mut self, stmts: &[Stmt], cursor: &mut usize) -> StmtResult {
    self.run_stmts_until(stmts, cursor, &|_| false)
}

#[inline(never)]
fn run_stmts_until(
    &mut self,
    stmts: &[Stmt],
    cursor: &mut usize,
```

```
    until: &dyn Fn(&Stmt) -> bool,
) -> StmtResult {
    while *cursor < stmts.len() {
        let stmt = &stmts[*cursor];
        *cursor += 1;

        if until(stmt) {
            return StmtResult::Ok;
        }

        let re = match stmt {
            Stmt::Block { label } => {
                let re = self.run_stmts_until(stmts, cursor, &|s| {
                    if let Stmt::EndBlock = s {
                        true
                    } else {
                        false
                    }
                });

                match re {
                    StmtResult::Break(l) => {
                        if let Some(a) = &l {
                            // the label does not match, fallout
                            if a != label {
                                return StmtResult::Break(l);
                            } else {
                                // a break occoured, loop until end is reached

                                // counter for block scope opened
                                let mut i = 0;
                                loop {
```

```
                    let s = &stmts[*cursor];
                    *cursor += 1;

                    if let Stmt::EndBlock = s {
                        // no interior scope is present
                        if i == 0 {
                            break;
                        } else {
                            // decrement interier scope
                            i -= 1;
                        }
                    };

                    // opens an interier scope
                    if let Stmt::Block { .. } = s {
                        i += 1;
                    }
                }
            }
        } else {
            return StmtResult::Break(l);
        }
    }
    StmtResult::Continue(_) => return re,
    StmtResult::Return(_) => return re,
    StmtResult::Error(_) => return re,
    StmtResult::Ok => (),
  };
}
Stmt::EndBlock => unreachable!(),
Stmt::Break(label) => return StmtResult::Break(label.clone()),
Stmt::Continue(label) => return StmtResult::Continue(label.clone()),
```

```
Stmt::Return(r) => {
    let re = self.run_expr(&r);

    match re {
        Ok(v) => return StmtResult::Return(v),
        Err(e) => return StmtResult::Error(e),
    }
}
Stmt::Throw(v) => {
    let re = self.run_expr(&v);
    let v = match re {
        Ok(v) => v,
        Err(v) => v,
    };

    return StmtResult::Error(v);
}
Stmt::DeclareClass(_)
| Stmt::DeclareFunction(_)
| Stmt::DeclareGenericClass(_)
| Stmt::DeclareGenericFunction(_)
| Stmt::DeclareGenericInterface(_)
| Stmt::DeclareInterface(_) => {
    // do nothing
}
Stmt::DeclareVar(v, _) => {
    // check if variable is already inserted
    assert!(self.context.contains(*v))
}
Stmt::DropVar(v) => {
    assert!(self.context.remove(*v));
}
```

```
Stmt::If { test } => {
    let value = match self.run_expr(&test) {
        Ok(v) => v,
        Err(e) => return StmtResult::Error(e),
    };

    self.assert_type(&value, &Type::Bool);

    let mut is_if_ran = false;

    // test value is true
    if self.to_bool(&value) {
        is_if_ran = true;

        // run statement in clause
        let re = self.run_stmts_until(stmts, cursor, &|s| {
            if let Stmt::EndIf = s {
                true
            } else {
                false
            }
        });

        match re {
            StmtResult::Ok => {}
            // does not accept breaks
            _ => return re,
        }
    } else {
        // loop until end if is reached

        // counter for if scope opened
```

```
    let mut i = 0;
    loop {
        let s = &stmts[*cursor];
        *cursor += 1;

        if let Stmt::EndIf = s {
            // no interior scope is present
            if i == 0 {
                break;
            } else {
                // decrement interier scope
                i -= 1;
            }
        };

        // opens an interier scope
        if let Stmt::If { .. } = s {
            i += 1;
        }
    }
};

// run the else clause
if let Some(Stmt::Else) = stmts.get(*cursor) {
    *cursor += 1;

    // the revious conditions are not met, else clause should run
    if !is_if_ran {
        let re = self.run_stmts_until(stmts, cursor, &|s| {
            if let Stmt::EndElse = s {
                true
            } else {
```

```
            false
        }
    });
    match re {
        StmtResult::Ok => {}
        _ => return re,
    }
} else {
    // else clause should not run
    // loop until end else is reached

    // counter for else scope opened
    let mut i = 0;
    loop {
        let s = &stmts[*cursor];
        *cursor += 1;

        if let Stmt::EndElse = s {
            // no interior scope is present
            if i == 0 {
                break;
            } else {
                // decrement interier scope
                i -= 1;
            }
        };

        // opens an interier scope
        if let Stmt::Else = s {
            i += 1;
        }
    }
```

```
        }
    };
}
Stmt::EndIf => unreachable!(),
Stmt::Else => unreachable!(),
Stmt::EndElse => unreachable!(),
// expression statement
Stmt::Expr(e) => {
    // run expression
    match self.run_expr(&e) {
        // ok
        Ok(v) => {
            // set pc to value
            self.pc = v;
        }
        // retunr error
        Err(e) => return StmtResult::Error(e),
    };
}
Stmt::Loop { label } => {
    // the loop
    loop {
        let mut new_cursor = *cursor;

        // execute statements within loop
        let re = self.run_stmts_until(stmts, &mut new_cursor, &|s| {
            if let Stmt::EndLoop = s {
                true
            } else {
                false
            }
        });
```

```
    println!("{:#?}", re);

    match re {
        StmtResult::Break(l) => {
            // if no label is specified, break anyways
            if l.is_none() || &l == label {
                break;
            } else {
                // label not match, break
                return StmtResult::Break(l);
            }
        }
        StmtResult::Continue(l) => {
            // label is not specidied or is none
            if l.is_none() || &l == label {
                continue;
            } else {
                return StmtResult::Continue(l);
            }
        }
        StmtResult::Return(_) => return re,
        StmtResult::Error(_) => return re,
        StmtResult::Ok => (),
    };
}

// loop until endloop is reached

// counter for loop scope opened
let mut i = 0;
loop {
```

```
        let s = &stmts[*cursor];
        *cursor += 1;

        if let Stmt::EndLoop = s {
            // no interior scope is present
            if i == 0 {
                break;
            } else {
                // decrement interier scope
                i -= 1;
            }
        };

        // opens an interier scope
        if let Stmt::Loop { .. } = s {
            i += 1;
        }
    }
}
Stmt::EndLoop => unreachable!(),
Stmt::Try => {
    let re = self.run_stmts_until(stmts, cursor, &|s| {
        if let Stmt::EndTry = s {
            true
        } else {
            false
        }
    });

    let mut error = None;
    match re {
        StmtResult::Ok => {}
```

```
    StmtResult::Error(e) => {
        error = Some(e);
    }
    _ => return re,
};

// an error means that endtry was not reached
if error.is_some() {
    // loop until endtry is reached

    // counter for try scope opened
    let mut i = 0;
    loop {
        let s = &stmts[*cursor];
        *cursor += 1;

        if let Stmt::EndTry = s {
            // no interior scope is present
            if i == 0 {
                break;
            } else {
                // decrement interier scope
                i -= 1;
            }
        };

        // opens an interier scope
        if let Stmt::Try = s {
            i += 1;
        }
    }
}
```

```
let mut error_in_catch = None;

// run the catch clause if error occoured
if let Some(error) = error {
    if let Some(Stmt::Catch(vid, ty)) = stmts.get(*cursor) {
        *cursor += 1;

        let error = self.cast(&error, ty);
        // insert binding
        self.context.declare(*vid, false);
        self.context.write(*vid, error);

        let re = self.run_stmts_until(stmts, cursor, &|s| {
            if let Stmt::EndCatch = s {
                true
            } else {
                false
            }
        });

        match re {
            StmtResult::Ok => {}
            StmtResult::Error(e) => {
                error_in_catch = Some(e);
            }
            _ => return re,
        }
    }
}

// run the finaliser no matter what
```

```
    if let Some(Stmt::Finally) = stmts.get(*cursor) {
        *cursor += 1;
        let re = self.run_stmts_until(stmts, cursor, &|s| {
            if let Stmt::EndFinally = s {
                true
            } else {
                false
            }
        });

        match re {
            StmtResult::Ok => {}
            _ => return re,
        }
    }

    // an error is thrown when catching
    if let Some(e) = error_in_catch {
        return StmtResult::Error(e);
    };
}
Stmt::EndTry => unreachable!(),
Stmt::Catch(_, _) => unreachable!(),
Stmt::EndCatch => unreachable!(),
Stmt::Finally => unreachable!(),
Stmt::EndFinally => unreachable!(),
Stmt::Switch(value) => {
    // run the expression
    let value = match self.run_expr(&value) {
        Ok(v) => v,
        Err(e) => return StmtResult::Error(e),
    };
```

```
let mut is_case_matched = false;

// run all the switch cases
while let Some(Stmt::SwitchCase(test)) = stmts.get(*cursor) {
    *cursor += 1;

    // run the expression
    let test = match self.run_expr(&test) {
        Ok(v) => v,
        Err(e) => return StmtResult::Error(e),
    };

    // if no case was matched, try to match case
    if !is_case_matched {
        is_case_matched = self.strict_equal(&value, &test);
    };

    // if case is matched, execute statements
    // this may be an effect of fallthrough
    if is_case_matched {
        let re = self.run_stmts_until(stmts, cursor, &|s| {
            if let Stmt::EndSwitchCase = s {
                true
            } else {
                false
            }
        });

        match re {
            StmtResult::Break(label) => {
                // break from switch
```

```
            if label.is_none() {
                break;
            } else {
                return StmtResult::Break(label);
            }
        }
        StmtResult::Ok => {}
        _ => return re,
    };
} else {
    // case is not matched, loop until end switch is reached

    // counter for switch case scope opened
    let mut i = 0;
    loop {
        let s = &stmts[*cursor];
        *cursor += 1;

        if let Stmt::EndSwitchCase = s {
            // no interior scope is present
            if i == 0 {
                break;
            } else {
                // decrement interier scope
                i -= 1;
            }
        };

        // opens an interier scope
        if let Stmt::SwitchCase(_) = s {
            i += 1;
        }
```

```
        }
    }
}

// run the default case if no case is matched
if !is_case_matched {
    if let Some(Stmt::DefaultCase) = stmts.get(*cursor) {
        *cursor += 1;

        let re = self.run_stmts_until(stmts, cursor, &|s| {
            if let Stmt::EndDefaultCase = s {
                true
            } else {
                false
            }
        });

        match re {
            StmtResult::Ok => {}
            _ => return re,
        }
    }
};

// loop until switch end

// counter for switch scope opened
let mut i = 0;
loop {
    let s = &stmts[*cursor];
    *cursor += 1;
```

```
                if let Stmt::EndSwitch = s {
                    // no interior scope is present
                    if i == 0 {
                        break;
                    } else {
                        // decrement interier scope
                        i -= 1;
                    }
                };

                // opens an interier scope
                if let Stmt::Switch(_) = s {
                    i += 1;
                }
            }
        }
        Stmt::EndSwitch => unreachable!(),
        Stmt::SwitchCase(_) => unreachable!(),
        Stmt::EndSwitchCase => unreachable!(),
        Stmt::DefaultCase => unreachable!(),
        Stmt::EndDefaultCase => unreachable!(),
    };
  }

  return StmtResult::Ok;
}

fn run_expr(&mut self, expr: &Expr) -> Result<Value, Value> {
  match expr {
    Expr::Undefined => Ok(Value::Undefined),
    Expr::Null => Ok(Value::Null),
    Expr::Int(i) => Ok(Value::Int(*i)),
```

```
Expr::Number(f) => Ok(Value::Number(*f)),
Expr::Bigint(b) => Ok(Value::Bigint(*b)),
Expr::Bool(b) => Ok(Value::Bool(*b)),
Expr::String(s) => Ok(Value::String(s.clone())),
Expr::Symbol(s) => Ok(Value::Symbol(*s)),
Expr::Regex() => Ok(Value::Regex()),
Expr::Function(f) => Ok(Value::Function(*f)),
Expr::This => Ok(self.this.clone()),
Expr::Array { values } => {
    let mut array = Vec::new();
    for v in values {
        array.push(self.run_expr(v)?);
    }
    Ok(Value::Array(Arc::new(RwLock::new(array))))
}
Expr::Tuple { values } => {
    let mut tuple = Vec::new();
    for v in values {
        tuple.push(self.run_expr(v)?);
    }
    Ok(Value::Tuple(Arc::new(RwLock::new(tuple))))
}
Expr::Object { props } => {
    let mut obj = Vec::new();
    for (p, e) in props {
        let v = self.run_expr(e)?;
        obj.push((p.clone(), v));
    }
    Ok(Value::DynObject { values: obj.into() })
}
Expr::New { class, args } => {
    todo!()
```

```
}
Expr::Call {
    callee,
    args,
    optional,
} => {
    let (this, mut callee) = match callee.as_ref() {
        Callee::Expr(e) => (self.this.clone(), self.run_expr(e)?),
        Callee::Function(f) => (self.this.clone(), Value::Function(*f)),
        Callee::Member { object, prop } => {
            todo!()
        }
        Callee::Super(s) => {
            todo!()
        }
    };

    let mut arguments = Vec::new();

    for arg in args {
        arguments.push(self.run_expr(arg)?);
    }

    if *optional {
        match self.assert_not_null(callee) {
            Ok(v) => callee = v,
            Err(_) => return Ok(Value::Undefined),
        }
    }

    match callee {
        Value::Function(id) => {
```

```
            let func = self.table.functions.get(&id).expect("invalid function");

            return self.run_function(func, this, &arguments, Context::new());
        }
        Value::Clousure { id, captures } => {
            let func = self.table.functions.get(&id).expect("invalid function");

            let mut ctx = Context::new();
            for (vid, cap) in captures.iter() {
                ctx.heap_variables.insert(*vid, cap.clone());
            }

            return self.run_function(func, this, &arguments, ctx);
        }
        _ => panic!("callee is not a function"),
    }
}
Expr::Push(v) => {
    let v = self.run_expr(v)?;
    self.stack.push(v.clone());
    return Ok(v);
}
Expr::ReadStack => Ok(self.stack.last().expect("stack underflow").clone()),
Expr::Pop => Ok(self.stack.pop().expect("stack underflow")),
Expr::Member {
    object,
    key,
    optional,
} => {
    let obj = self.run_expr(&object)?;

    let v = match key {
```

```
        PropNameOrExpr::PropName(p) => self.get_property(&obj, p),
        PropNameOrExpr::Expr(e, _) => {
            let key = self.run_expr(e)?;
            self.get_property_expr(&obj, &key)?
        }
    };

    if let Some(v) = v {
        Ok(v)
    } else {
        if *optional {
            Ok(Value::Undefined)
        } else {
            panic!("missing property")
        }
    }
}
Expr::MemberAssign {
    op,
    object,
    key,
    value,
} => {
    let obj = self.run_expr(&object)?;
    let value = self.run_expr(&value)?;

    if op == &AssignOp::Assign {
        match key {
            PropNameOrExpr::PropName(p) => self.set_property(&obj, p, value.clone()),
            PropNameOrExpr::Expr(e, _) => {
                let key = self.run_expr(e)?;
                self.set_property_expr(&obj, key, value.clone());
```

```
        }
    };
    return Ok(value);
}

enum PorV {
    P(PropName),
    V(Value),
}

let key = match &key {
    PropNameOrExpr::PropName(p) => PorV::P(p.clone()),
    PropNameOrExpr::Expr(e, _) => {
        let key = self.run_expr(e)?;
        PorV::V(key)
    }
};

let old = match &key {
    PorV::P(key) => self.get_property(&obj, &key),
    PorV::V(v) => self.get_property_expr(&obj, &v)?,
}
.expect("invalid property");

let value = match op {
    AssignOp::Assign => unreachable!(),
    AssignOp::AddAssign => self.add(old, value),
    AssignOp::SubAssign => self.sub(old, value),
    AssignOp::MulAssign => self.mul(old, value),
    AssignOp::DivAssign => self.div(old, value),
    AssignOp::ExpAssign => self.exp(old, value)?,
    AssignOp::ModAssign => self.rem(old, value),
```

```
            AssignOp::LShiftAssign => self.lshift(old, value),
            AssignOp::RShiftAssign => self.rshift(old, value),
            AssignOp::ZeroFillRShiftAssign => self.zero_fill_rshift(old, value),
            AssignOp::BitAndAssign => self.bitand(old, value),
            AssignOp::BitOrAssign => self.bitor(old, value),
            AssignOp::BitXorAssign => self.bixor(old, value),
            AssignOp::OrAssign => self.or(old, value),
            AssignOp::AndAssign => self.and(old, value),
            AssignOp::NullishAssign => self.nullish(old, value),
        };

        match key {
            PorV::P(p) => self.set_property(&obj, &p, value.clone()),
            PorV::V(key) => {
                self.set_property_expr(&obj, key, value.clone());
            }
        };

        return Ok(value);
    }
    Expr::MemberUpdate { op, object, key } => {
        let obj = self.run_expr(&object)?;

        enum PorV {
            P(PropName),
            V(Value),
        }

        let key = match key {
            PropNameOrExpr::PropName(p) => PorV::P(p.clone()),
            PropNameOrExpr::Expr(e, _) => {
                let key = self.run_expr(e)?;
```

```
        PorV::V(key)
    }
};

let old = match &key {
    PorV::P(key) => self.get_property(&obj, &key),
    PorV::V(v) => self.get_property_expr(&obj, &v)?,
}
.expect("invalid property");

let (rv, nv) = match op {
    UpdateOp::PrefixAdd => match old {
        Value::Int(i) => (Value::Int(i + 1), Value::Int(i + 1)),
        Value::Number(n) => (Value::Number(n + 1.0), Value::Number(n + 1.0)),
        Value::Bigint(n) => (Value::Bigint(n + 1), Value::Bigint(n + 1)),
        _ => panic!(),
    },
    UpdateOp::PrefixSub => match old {
        Value::Int(i) => (Value::Int(i - 1), Value::Int(i - 1)),
        Value::Number(n) => (Value::Number(n - 1.0), Value::Number(n - 1.0)),
        Value::Bigint(n) => (Value::Bigint(n - 1), Value::Bigint(n - 1)),
        _ => panic!(),
    },
    UpdateOp::SuffixAdd => match old {
        Value::Int(i) => (Value::Int(i), Value::Int(i + 1)),
        Value::Number(n) => (Value::Number(n), Value::Number(n + 1.0)),
        Value::Bigint(n) => (Value::Bigint(n), Value::Bigint(n + 1)),
        _ => panic!(),
    },
    UpdateOp::SuffixSub => match old {
        Value::Int(i) => (Value::Int(i), Value::Int(i - 1)),
        Value::Number(n) => (Value::Number(n), Value::Number(n - 1.0)),
```

```
            Value::Bigint(n) => (Value::Bigint(n), Value::Bigint(n - 1)),
            _ => panic!(),
        },
    };

    match key {
        PorV::P(p) => self.set_property(&obj, &p, nv),
        PorV::V(key) => {
            self.set_property_expr(&obj, key, nv);
        }
    };

    return Ok(rv);
}
Expr::VarAssign {
    op,
    variable,
    value,
} => {
    let value = self.run_expr(&value)?;

    if let AssignOp::Assign = op {
        self.context.write(*variable, value.clone());
        return Ok(value);
    }

    let old = self.context.read(*variable);

    let value = match op {
        AssignOp::Assign => unreachable!(),
        AssignOp::AddAssign => self.add(old, value),
        AssignOp::SubAssign => self.sub(old, value),
```

```
        AssignOp::MulAssign => self.mul(old, value),
        AssignOp::DivAssign => self.div(old, value),
        AssignOp::ExpAssign => self.exp(old, value)?,
        AssignOp::ModAssign => self.rem(old, value),
        AssignOp::LShiftAssign => self.lshift(old, value),
        AssignOp::RShiftAssign => self.rshift(old, value),
        AssignOp::ZeroFillRShiftAssign => self.zero_fill_rshift(old, value),
        AssignOp::BitAndAssign => self.bitand(old, value),
        AssignOp::BitOrAssign => self.bitor(old, value),
        AssignOp::BitXorAssign => self.bixor(old, value),
        AssignOp::OrAssign => self.or(old, value),
        AssignOp::AndAssign => self.and(old, value),
        AssignOp::NullishAssign => self.nullish(old, value),
    };

    self.context.write(*variable, value.clone());

    return Ok(value);
}
Expr::VarLoad { span: _, variable } => return Ok(self.context.read(*variable)),
Expr::VarUpdate { op, variable } => {
    let old = self.context.read(*variable);

    let (rv, nv) = match op {
        UpdateOp::PrefixAdd => match old {
            Value::Int(i) => (Value::Int(i + 1), Value::Int(i + 1)),
            Value::Number(n) => (Value::Number(n + 1.0), Value::Number(n + 1.0)),
            Value::Bigint(n) => (Value::Bigint(n + 1), Value::Bigint(n + 1)),
            _ => panic!(),
        },
        UpdateOp::PrefixSub => match old {
            Value::Int(i) => (Value::Int(i - 1), Value::Int(i - 1)),
```

```
            Value::Number(n) => (Value::Number(n - 1.0), Value::Number(n - 1.0)),
            Value::Bigint(n) => (Value::Bigint(n - 1), Value::Bigint(n - 1)),
            _ => panic!(),
        },
        UpdateOp::SuffixAdd => match old {
            Value::Int(i) => (Value::Int(i), Value::Int(i + 1)),
            Value::Number(n) => (Value::Number(n), Value::Number(n + 1.0)),
            Value::Bigint(n) => (Value::Bigint(n), Value::Bigint(n + 1)),
            _ => panic!(),
        },
        UpdateOp::SuffixSub => match old {
            Value::Int(i) => (Value::Int(i), Value::Int(i - 1)),
            Value::Number(n) => (Value::Number(n), Value::Number(n - 1.0)),
            Value::Bigint(n) => (Value::Bigint(n), Value::Bigint(n - 1)),
            _ => panic!(),
        },
    };

    self.context.write(*variable, nv);

    return Ok(rv);
}
Expr::Bin { op, left, right } => {
    let a = self.run_expr(&left)?;
    let b = self.run_expr(&right)?;

    let v = match op {
        BinOp::Add => self.add(a, b),
        BinOp::Sub => self.sub(a, b),
        BinOp::Mul => self.mul(a, b),
        BinOp::Div => self.div(a, b),
        BinOp::Mod => self.rem(a, b),
```

```
            BinOp::BitAnd => self.bitand(a, b),
            BinOp::BitOr => self.bitor(a, b),
            BinOp::BitXor => self.bixor(a, b),
            BinOp::RShift => self.rshift(a, b),
            BinOp::LShift => self.lshift(a, b),
            BinOp::URShift => self.zero_fill_rshift(a, b),
            BinOp::Exp => self.exp(a, b)?,
            BinOp::EqEq => Value::Bool(self.equals(&a, &b)),
            BinOp::EqEqEq => Value::Bool(self.strict_equal(&a, &b)),
            BinOp::NotEq => Value::Bool(!self.equals(&a, &b)),
            BinOp::NotEqEq => Value::Bool(!self.strict_equal(&a, &b)),
            BinOp::And => self.and(a, b),
            BinOp::Or => self.or(a, b),
            BinOp::Gt => self.gt(a, b),
            BinOp::Gteq => self.gteq(a, b),
            BinOp::Lt => self.lt(a, b),
            BinOp::Lteq => self.lteq(a, b),
            BinOp::Nullish => self.nullish(a, b),
            BinOp::In => Value::Bool(self.get_property_expr(&a, &b)?.is_some()),
        };

        return Ok(v);
    }
    Expr::Unary { op, value } => {
        let value = self.run_expr(&value)?;

        let v = match op {
            UnaryOp::Void => Value::Undefined,
            UnaryOp::LogicalNot => Value::Bool(!self.to_bool(&value)),
            UnaryOp::BitNot => match value {
                Value::Int(i) => Value::Int(!i),
                Value::Number(n) => Value::Int(!(n as i32)),
```

```
            Value::Bigint(i) => Value::Bigint(!i),
            _ => panic!(),
        },
        UnaryOp::Plus => match value {
            Value::Int(i) => Value::Int(i),
            Value::Number(n) => Value::Number(n),
            Value::Bigint(i) => Value::Bigint(i),
            _ => panic!(),
        },
        UnaryOp::Minus => match value {
            Value::Int(i) => Value::Int(-i),
            Value::Number(n) => Value::Number(-n),
            Value::Bigint(i) => Value::Bigint(-i),
            _ => panic!(),
        },
        UnaryOp::Typeof => {
            let s = match value {
                Value::Int(_) | Value::Number(_) => "number",
                Value::Bigint(_) => "bigint",
                Value::Null => "null",
                Value::Undefined => "undefined",
                Value::Bool(_) => "boolean",
                Value::String(_) => "string",
                Value::Symbol(_) => "symbol",
                _ => "object",
            };
            Value::String(s.to_string())
        }
    };

    return Ok(v);
}
```

```
Expr::Ternary { test, left, right } => {
    let test = self.run_expr(&test)?;
    let left = self.run_expr(&left)?;
    let right = self.run_expr(&right)?;

    if self.to_bool(&test) {
        Ok(left)
    } else {
        Ok(right)
    }
}
Expr::Seq(a, b) => {
    let _ = self.run_expr(&a)?;
    let b = self.run_expr(&b)?;

    return Ok(b);
}
Expr::Await(p) => {
    let p = self.run_expr(&p)?;
    todo!()
}
Expr::Yield(y) => {
    let y = self.run_expr(&y)?;
    todo!()
}
Expr::Closure(c) => {
    let func = self.table.functions.get(c).expect("invalid function");

    let mut cap = HashMap::new();

    for (vid, _) in &func.captures {
        let v = self
```

```
                .context
                .heap_variables
                .get(vid)
                .expect("invalid variable");
            cap.insert(*vid, v.clone());
        }

        return Ok(Value::Clousure {
            id: *c,
            captures: Arc::new(cap),
        });
    }
    Expr::Cast(v, ty) => {
        let v = self.run_expr(&v)?;
        return Ok(self.cast(&v, ty));
    }
    Expr::AssertNonNull(v) => {
        let v = self.run_expr(&v)?;

        return Ok(self.assert_not_null(v)?);
    }
  }
}

fn cast(&mut self, value: &Value, ty: &Type) -> Value {
  match value {
    Value::Any(value) => self.cast(value, ty),
    Value::AnyObject(value) => self.cast(value, ty),
    Value::Interface { value, .. } => self.cast(value, ty),
    Value::Union { value, .. } => self.cast(value, ty),
    Value::Bigint(b) => match ty {
      Type::Bigint => value.clone(),
```

```
    Type::Any => Value::Any(Box::new(value.clone())),
    Type::Bool => Value::Bool(*b != 0),
    Type::Int => Value::Int(*b as i32),
    Type::Number => Value::Number(*b as f64),
    Type::String => Value::String(b.to_string()),
    Type::Interface(id) => Value::Interface {
        id: *id,
        value: Box::new(value.clone()),
    },
    Type::Union(u) => Value::Union {
        ty: u.clone(),
        value: Box::new(value.clone()),
    },
    _ => panic!("invalid cast"),
},
Value::Int(i) => match ty {
    Type::Int => value.clone(),
    Type::Any => Value::Any(Box::new(value.clone())),
    Type::Bool => Value::Bool(*i != 0),
    Type::Number => Value::Number(*i as f64),
    Type::Bigint => Value::Bigint(*i as i128),
    Type::String => Value::String(i.to_string()),
    Type::Interface(id) => Value::Interface {
        id: *id,
        value: Box::new(value.clone()),
    },
    Type::Union(u) => Value::Union {
        ty: u.clone(),
        value: Box::new(value.clone()),
    },
    _ => panic!("invalid cast"),
},
```

```
Value::Number(f) => match ty {
    Type::Number => value.clone(),
    Type::Any => Value::Any(Box::new(value.clone())),
    Type::Bool => Value::Bool(!f.is_nan() && *f != 0.0),
    Type::Int => Value::Int(*f as i32),
    Type::Bigint => Value::Bigint(*f as i128),
    Type::String => Value::String(f.to_string()),
    Type::Interface(id) => Value::Interface {
        id: *id,
        value: Box::new(value.clone()),
    },
    Type::Union(u) => Value::Union {
        ty: u.clone(),
        value: Box::new(value.clone()),
    },
    _ => panic!("invalid cast"),
},
Value::Bool(b) => match ty {
    Type::Bool => Value::Bool(*b),
    Type::Any => Value::Any(Box::new(value.clone())),
    Type::Int => Value::Int(*b as i32),
    Type::Number => Value::Number(if *b { 1.0 } else { 0.0 }),
    Type::Bigint => Value::Bigint(*b as i128),
    Type::String => Value::String(b.to_string()),
    Type::Interface(id) => Value::Interface {
        id: *id,
        value: Box::new(value.clone()),
    },
    Type::Union(u) => Value::Union {
        ty: u.clone(),
        value: Box::new(value.clone()),
    },
```

```
    _ => panic!("invalid cast"),
},
Value::Tuple(t) => match ty {
    Type::Array(_) => Value::Array(t.clone()),
    Type::Any => Value::Any(Box::new(value.clone())),
    Type::AnyObject => Value::AnyObject(Box::new(Value::Tuple(t.clone()))),
    Type::Interface(id) => Value::Interface {
        id: *id,
        value: Box::new(value.clone()),
    },
    Type::Union(u) => Value::Union {
        ty: u.clone(),
        value: Box::new(value.clone()),
    },
    _ => panic!("invalid cast"),
},
Value::Object { id, values } => match ty {
    Type::Object(super_class) => {
        todo!()
    }
    Type::LiteralObject(o) => {
        todo!()
    }
    Type::Any => Value::Any(Box::new(value.clone())),
    Type::AnyObject => Value::AnyObject(Box::new(value.clone())),
    Type::Interface(id) => Value::Interface {
        id: *id,
        value: Box::new(value.clone()),
    },
    Type::Union(u) => Value::Union {
        ty: u.clone(),
        value: Box::new(value.clone()),
```

```
            },
            _ => panic!("invalid cast"),
        },
        _ => match ty {
            Type::Any => Value::Any(Box::new(value.clone())),
            Type::AnyObject => Value::AnyObject(Box::new(value.clone())),
            Type::Interface(id) => Value::Interface {
                id: *id,
                value: Box::new(value.clone()),
            },
            Type::Union(u) => Value::Union {
                ty: u.clone(),
                value: Box::new(value.clone()),
            },
            _ => panic!("invalid cast"),
        },
    }
}

fn get_property(&self, value: &Value, prop: &PropName) -> Option<Value> {
    match value {
        Value::Array(a) => match prop {
            PropName::Ident(s) => match s.as_str() {
                "length" => Some(Value::Int(a.read().len() as i32)),
                _ => None,
            },
            _ => todo!(),
        },
        _ => todo!(),
    }
}
```

```
fn get_property_expr(&self, value: &Value, key: &Value) -> Result<Option<Value>, Value> {
    match value {
        Value::Any(value) => self.get_property_expr(value, key),
        Value::AnyObject(value) => self.get_property_expr(value, key),
        Value::Interface { value, .. } => self.get_property_expr(value, key),
        Value::Union { value, .. } => self.get_property_expr(value, key),
        Value::Array(a) => match key {
            Value::Int(i) => {
                if let Some(v) = a.read().get(*i as usize) {
                    Ok(Some(v.clone()))
                } else {
                    Err(Value::String(format!("index out of range: {}", *i)))
                }
            }
            Value::Number(i) => {
                if let Some(v) = a.read().get(*i as usize) {
                    Ok(Some(v.clone()))
                } else {
                    Err(Value::String(format!("index out of range: {}", *i)))
                }
            }
            _ => Ok(None),
        },
        Value::Tuple(a) => match key {
            Value::Int(i) => {
                if let Some(v) = a.read().get(*i as usize) {
                    Ok(Some(v.clone()))
                } else {
                    Err(Value::String(format!("index out of range: {}", *i)))
                }
            }
            Value::Number(i) => {
```

```
                if let Some(v) = a.read().get(*i as usize) {
                    Ok(Some(v.clone()))
                } else {
                    Err(Value::String(format!("index out of range: {}", *i)))
                }
            }
            _ => Ok(None),
        },
        _ => Ok(None),
    }
}

fn set_property(&self, obj: &Value, prop: &PropName, value: Value) {
    todo!()
}

fn set_property_expr(&self, obj: &Value, key: Value, value: Value) {
    todo!()
}

fn strict_equal(&self, left: &Value, right: &Value) -> bool {
    match right {
        Value::Any(v) => return self.strict_equal(left, v),
        Value::AnyObject(v) => return self.strict_equal(left, &v),
        Value::Interface { value, .. } => return self.strict_equal(left, &value),
        Value::Union { value, .. } => return self.strict_equal(left, &value),
        _ => {}
    }
    match left {
        Value::Any(v) => return self.strict_equal(v, right),
        Value::AnyObject(v) => return self.strict_equal(&v, right),
        Value::Interface { value, .. } => return self.strict_equal(&value, right),
```

```
Value::Union { value, .. } => return self.strict_equal(&value, right),
Value::Bigint(i) => match right {
    Value::Bigint(n) => i == n,
    _ => false,
},
Value::Bool(a) => match right {
    Value::Bool(b) => a == b,
    _ => false,
},
Value::Clousure {
    id: id1,
    captures: cap1,
} => match right {
    Value::Clousure {
        id: id2,
        captures: cap2,
    } => id1 == id2 && Arc::as_ptr(cap1) == Arc::as_ptr(cap2),
    _ => false,
},

Value::Enum {
    id: id1,
    variant: var1,
} => match right {
    Value::Enum {
        id: id2,
        variant: var2,
    } => *id1 == *id2 && *var1 == *var2,
    _ => false,
},
Value::Function(f1) => match right {
    Value::Function(f2) => *f1 == *f2,
```

```
    _ => false,
},
Value::Int(i) => match right {
    Value::Int(n) => i == n,
    Value::Number(n) => *i as f64 == *n,
    _ => false,
},
Value::Number(i) => match right {
    Value::Int(n) => (*n as f64) == *i,
    Value::Number(n) => i == n,
    _ => false,
},
Value::Null => match right {
    Value::Null => true,
    _ => false,
},
Value::DynObject { values: o1 } => match right {
    Value::DynObject { values: o2 } => Arc::as_ptr(o1) == Arc::as_ptr(o2),
    Value::Object { values: o2, .. } => Arc::as_ptr(o1) == Arc::as_ptr(o2),
    _ => false,
},
Value::Object { values: o1, .. } => match right {
    Value::Object { values: o2, .. } => Arc::as_ptr(o1) == Arc::as_ptr(o2),
    Value::DynObject { values: o2 } => Arc::as_ptr(o1) == Arc::as_ptr(o2),
    _ => false,
},
Value::Regex() => match right {
    Value::Regex() => true,
    _ => false,
},
Value::String(s1) => match right {
    Value::String(s2) => s1 == s2,
```

```
            _ => false,
        },
        Value::Symbol(s1) => match right {
            Value::Symbol(s2) => s1 == s2,
            _ => false,
        },
        Value::Undefined => match right {
            Value::Undefined => true,
            _ => false,
        },
        Value::Array(a) => match right {
            Value::Array(b) => Arc::as_ptr(a) == Arc::as_ptr(b),
            Value::Tuple(b) => Arc::as_ptr(a) == Arc::as_ptr(b),
            _ => false,
        },
        Value::Tuple(a) => match right {
            Value::Array(b) => Arc::as_ptr(a) == Arc::as_ptr(b),
            Value::Tuple(b) => Arc::as_ptr(a) == Arc::as_ptr(b),
            _ => false,
        },
    }
}

fn equals(&self, a: &Value, b: &Value) -> bool {
    println!("{:?} == {:?}", a, b);
    self.strict_equal(a, b)
}

fn get_type(&self, value: &Value) -> Type {
    todo!()
}
```

```
fn assert_type(&self, value: &Value, ty: &Type) {
    //assert!(&self.get_type(value) == ty)
}

fn is_nullable(&self, value: &Value) -> bool {
    match value {
        Value::Null => true,
        Value::Undefined => true,
        Value::Any(v) => self.is_nullable(v),
        Value::Interface { value, .. } => self.is_nullable(&value),
        Value::Union { value, .. } => self.is_nullable(&value),
        _ => false,
    }
}

fn assert_not_null(&self, value: Value) -> Result<Value, Value> {
    match value {
        Value::Null | Value::Undefined => {
            Err(Value::String("assert not null failed".to_string()))
        }
        Value::Any(a) => self.assert_not_null(*a),
        Value::Interface { value, .. } => self.assert_not_null(*value),
        Value::Union { ty, value } => {
            if self.is_nullable(&value) {
                return Err(Value::String("assert not null failed".to_string()));
            }

            let mut tys = ty.to_vec();
            tys.retain(|t| t != &Type::Null && t != &Type::Undefined);

            if tys.len() == 1 {
                return Ok(*value);
```

```
            }
            return Ok(Value::Union {
                ty: tys.into(),
                value: value,
            });
        }
        _ => Ok(value),
    }
}

fn add(&self, a: Value, b: Value) -> Value {
    match a {
        Value::Int(i) => match b {
            Value::Int(n) => Value::Int(i + n),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Number(i + n),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bigint(i + n),
            _ => panic!(),
        },
        Value::String(i) => match b {
            Value::String(n) => Value::String(i + n.as_str()),
            _ => panic!(),
        },
        _ => panic!(),
    }
}
```

```
fn sub(&self, a: Value, b: Value) -> Value {
    match a {
        Value::Int(i) => match b {
            Value::Int(n) => Value::Int(i - n),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Number(i - n),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bigint(i - n),
            _ => panic!(),
        },
        _ => panic!(),
    }
}

fn mul(&self, a: Value, b: Value) -> Value {
    match a {
        Value::Int(i) => match b {
            Value::Int(n) => Value::Int(i * n),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Number(i * n),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bigint(i * n),
            _ => panic!(),
        },
```

```
        _ => panic!(),
    }
}
fn div(&self, a: Value, b: Value) -> Value {
    match a {
        Value::Int(i) => match b {
            Value::Int(n) => Value::Number((i as f64) / (n as f64)),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Number(i / n),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bigint(i / n),
            _ => panic!(),
        },
        _ => panic!(),
    }
}

fn exp(&self, a: Value, b: Value) -> Result<Value, Value> {
    match a {
        Value::Int(i) => match b {
            Value::Int(n) => {
                if n < 0 {
                    Ok(Value::Number((i as f64).powi(n)))
                } else {
                    Ok(Value::Int(i.pow(n as u32)))
                }
            }
            //Value::Number(n) => {
```

```
            //    Ok(Value::Number((i as f64).powf(n)))
            //}
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Ok(Value::Number(i.powf(n))),
            //Value::Int(n) => Ok(Value::Number(i.powi(n))),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => {
                if n < 0 || n > u32::MAX as _ {
                    Err(Value::String(
                        "RangeError: Exponent must be positive".to_string(),
                    ))
                } else {
                    if let Some(v) = i.checked_pow(i as u32) {
                        Ok(Value::Bigint(v))
                    } else {
                        Err(Value::String(
                            "RangeError: Exponent must be positive".to_string(),
                        ))
                    }
                }
            }
            _ => panic!(),
        },
        _ => panic!(),
    }
}

fn rem(&self, a: Value, b: Value) -> Value {
```

```
    println!("{:?} % {:?}", a, b);
    match a {
        Value::Int(i) => match b {
            Value::Int(b) => Value::Int(i % b),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Number(i % n),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bigint(i % n),
            _ => panic!(),
        },
        _ => panic!(),
    }
}

fn lshift(&self, a: Value, b: Value) -> Value {
    match a {
        Value::Int(i) => match b {
            Value::Int(b) => Value::Int(i << b),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Int((i as i32) << (n as i32)),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bigint(i << n),
            _ => panic!(),
        },
```

```
        _ => panic!(),
    }
}

fn rshift(&self, a: Value, b: Value) -> Value {
    match a {
        Value::Int(i) => match b {
            Value::Int(b) => Value::Int(i >> b),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Int((i as i32) >> (n as i32)),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bigint(i >> n),
            _ => panic!(),
        },
        _ => panic!(),
    }
}

fn zero_fill_rshift(&self, a: Value, b: Value) -> Value {
    match a {
        Value::Int(i) => match b {
            Value::Int(b) => Value::Int(((i as u32) >> (b as u32)) as i32),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Int(((i as i32 as u32) >> (n as i32 as u32)) as i32),
            _ => panic!(),
        },
```

```
            Value::Bigint(i) => match b {
                Value::Bigint(n) => Value::Bigint((i as u128 >> n as u128) as i128),
                _ => panic!(),
            },
            _ => panic!(),
        }
    }

    fn bitand(&self, a: Value, b: Value) -> Value {
        match a {
            Value::Int(i) => match b {
                Value::Int(b) => Value::Int(i & b),
                _ => panic!(),
            },
            Value::Number(i) => match b {
                Value::Number(n) => Value::Int((i as i32) & (n as i32)),
                _ => panic!(),
            },
            Value::Bigint(i) => match b {
                Value::Bigint(n) => Value::Bigint(i & n),
                _ => panic!(),
            },
            _ => panic!(),
        }
    }
    fn bitor(&self, a: Value, b: Value) -> Value {
        match a {
            Value::Int(i) => match b {
                Value::Int(b) => Value::Int(i | b),
                _ => panic!(),
            },
            Value::Number(i) => match b {
```

```
            Value::Number(n) => Value::Int((i as i32) | (n as i32)),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bigint(i | n),
            _ => panic!(),
        },
        _ => panic!(),
    }
}

fn bixor(&self, a: Value, b: Value) -> Value {
    match a {
        Value::Int(i) => match b {
            Value::Int(b) => Value::Int(i ^ b),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Int((i as i32) ^ (n as i32)),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bigint(i ^ n),
            _ => panic!(),
        },
        _ => panic!(),
    }
}

fn gt(&self, a: Value, b: Value) -> Value {
    match a {
        Value::Int(i) => match b {
```

```
            Value::Int(n) => Value::Bool(i > n),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Bool(i > n),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bool(i > n),
            _ => panic!(),
        },
        _ => panic!(),
    }
}

fn gteq(&self, a: Value, b: Value) -> Value {
    println!("{:?} >= {:?}", a, b);
    match a {
        Value::Int(i) => match b {
            Value::Int(n) => Value::Bool(i >= n),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Bool(i >= n),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bool(i >= n),
            _ => panic!(),
        },
        _ => panic!(),
    }
```

```
}

fn lt(&self, a: Value, b: Value) -> Value {
    match a {
        Value::Int(i) => match b {
            Value::Int(n) => Value::Bool(i < n),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Bool(i < n),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bool(i < n),
            _ => panic!(),
        },
        _ => panic!(),
    }
}

fn lteq(&self, a: Value, b: Value) -> Value {
    match a {
        Value::Int(i) => match b {
            Value::Int(n) => Value::Bool(i <= n),
            _ => panic!(),
        },
        Value::Number(i) => match b {
            Value::Number(n) => Value::Bool(i <= n),
            _ => panic!(),
        },
        Value::Bigint(i) => match b {
            Value::Bigint(n) => Value::Bool(i <= n),
```

```
            _ => panic!(),
        },
        _ => panic!(),
    }
}

fn to_bool(&self, value: &Value) -> bool {
    match value {
        Value::Any(v) => self.to_bool(v),
        Value::AnyObject(v) => self.to_bool(v),
        Value::Array(_) => true,
        Value::Bigint(i) => *i != 0,
        Value::Bool(b) => *b,
        Value::Number(i) => !i.is_nan() && *i != 0.0,
        Value::Int(i) => *i != 0,
        Value::Function(_) => true,
        Value::Clousure { .. } => true,
        Value::Object { .. } => true,
        Value::DynObject { .. } => true,
        Value::Regex() => true,
        Value::Symbol(_) => true,
        Value::String(s) => !s.is_empty(),
        Value::Interface { value, .. } => self.to_bool(&value),
        Value::Tuple(_) => true,
        Value::Union { value, .. } => self.to_bool(&value),
        Value::Null => false,
        Value::Undefined => false,
        Value::Enum { .. } => true,
    }
}

fn and(&self, a: Value, b: Value) -> Value {
```

```
        return Value::Bool(self.to_bool(&a) && self.to_bool(&b));
    }

    fn or(&self, a: Value, b: Value) -> Value {
        if self.to_bool(&a) {
            return a;
        } else {
            return b;
        }
    }

    fn nullish(&self, a: Value, b: Value) -> Value {
        if self.is_nullable(&a) {
            return b;
        } else {
            return a;
        }
    }
}
```

5.3 native-ts-mir

5.3.1 ntive-ts-mir/lib.rs

```
extern crate alloc;
```

```
pub mod builder;
mod context;
mod function;
pub mod mir;
pub mod passes;
pub mod runtime;
pub mod types;
pub mod backend;
mod util;
mod value;

pub use builder::{Block, Builder, StackSlot};
pub use context::{Context, Linkage};
pub use function::Function;
pub use types::Type;
pub use value::Value;
```

5.3.2 native-ts-mir/value.rs

```
use std::marker::PhantomData;

use paste::paste;

pub use crate::types::*;
use crate::util::ValueID;

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Value<'ctx, 'func, T: MarkerType<'ctx>> {
pub(crate) id: ValueID,
pub(crate) ty: T,
pub(crate) _mark: PhantomData<(&'func (), &'ctx ())>,
}

impl<'ctx, 'func, T: MarkerType<'ctx>> Value<'ctx, 'func, T> {
pub fn ty(&self) -> &T {
&self.ty
}
pub fn id(&self) -> ValueID {
self.id
}
}

macro_rules! impl_into_auto {
($ty:ty) => {
```

```
impl<'ctx, 'func> Into<Value<'ctx, 'func, Auto<'ctx>>> for Value<'ctx, 'func,
$ty> {
fn into(self) -> Value<'ctx, 'func, Auto<'ctx>> {
Value {
id: self.id,
ty: Auto {
inner: self.ty.to_type(),
},
_mark: PhantomData,
}
}
}
};
}

impl_into_auto!(Void);
impl_into_auto!(U8);
impl_into_auto!(U16);
impl_into_auto!(U32);
impl_into_auto!(U64);
impl_into_auto!(Usize);
impl_into_auto!(I8);
impl_into_auto!(I16);
impl_into_auto!(I32);
impl_into_auto!(I64);
impl_into_auto!(Isize);
impl_into_auto!(F32);
impl_into_auto!(F64);
impl_into_auto!(Aggregate<'ctx>);
impl_into_auto!(Interface<'ctx>);

impl<'ctx, 'func, T: MarkerType<'ctx>> Into<Value<'ctx, 'func, Auto<'ctx>>>
for Value<'ctx, 'func, Pointer<T>>
{
fn into(self) -> Value<'ctx, 'func, Auto<'ctx>> {
Value {
id: self.id,
ty: Auto {
inner: self.ty.to_type(),
},
_mark: PhantomData,
}
}
}
```

```
impl<'ctx, 'func, Arg: FunctionArgs<'ctx>, R: MarkerType<'ctx>>
Into<Value<'ctx, 'func, Auto<'ctx>>>
for Value<'ctx, 'func, Function<'ctx, Arg, R>>
{
fn into(self) -> Value<'ctx, 'func, Auto<'ctx>> {
Value {
id: self.id,
ty: Auto {
inner: self.ty.to_type(),
},
_mark: PhantomData,
}
}
}
impl<'ctx, 'func, T: MarkerType<'ctx>> Into<Value<'ctx, 'func, Auto<'ctx>>>
for Value<'ctx, 'func, Array<T>>
{
fn into(self) -> Value<'ctx, 'func, Auto<'ctx>> {
Value {
id: self.id,
ty: Auto {
inner: self.ty.to_type(),
},
_mark: PhantomData,
}
}
}
impl<'ctx, 'func, T: MarkerType<'ctx>> Into<Value<'ctx, 'func, Auto<'ctx>>>
for Value<'ctx, 'func, Future<T>>
{
fn into(self) -> Value<'ctx, 'func, Auto<'ctx>> {
Value {
id: self.id,
ty: Auto {
inner: self.ty.to_type(),
},
_mark: PhantomData,
}
}
}
impl<'ctx, 'func, T: ScalarMarkerType, const N: usize> Into<Value<'ctx, 'func,
Auto<'ctx>>>
for Value<'ctx, 'func, SIMD<T, N>>
where
```

```
simd::LaneCount<N>: simd::SupportedLaneCount,
{
fn into(self) -> Value<'ctx, 'func, Auto<'ctx>> {
Value {
id: self.id,
ty: Auto {
inner: self.ty.to_type(),
},
_mark: PhantomData,
}
}
}

macro_rules! from_auto {
($ty:ident) => {
paste! {
pub fn [<into_ $ty:lower>](&self) -> Value<'ctx, 'func, $ty>{
if let Type::$ty = self.ty.to_type(){
return Value{
id: self.id,
ty: $ty,
_mark: PhantomData
}
}
panic!(stringify!(value is not $ty))
}
}
};
}

impl<'ctx, 'func, T: MarkerType<'ctx>> Value<'ctx, 'func, T> {
pub fn into_auto(&self) -> Value<'ctx, 'func, Auto<'ctx>> {
Value {
id: self.id,
ty: Auto {
inner: self.ty.to_type(),
},
_mark: PhantomData,
}
}
}

impl<'ctx, 'func> Value<'ctx, 'func, Function<'ctx, AutoArgs<'ctx>,
Auto<'ctx>>>{
```

```
pub fn into_function<A: FunctionArgs<'ctx>, R: MarkerType<'ctx>>(&self,
args: A, return_: R) -> Value<'ctx, 'func, Function<'ctx, A, R>>{
if self.ty.args.len() != args.len() {
panic!("value is function, but wrong number of arguments provided")
}
for (i, ty) in self.ty.args.0.iter().enumerate() {
if ty != &args.get(i) {
panic!("value is function, but argument {} has wrong type", i)
}
}
if self.ty.return_.inner != return_.to_type() {
panic!("value is function, but return type does not match")
}

return Value {
id: self.id,
ty: Function {
args,
return_,
_mark: PhantomData,
},
_mark: PhantomData,
};
}
}

impl<'ctx, 'func> Value<'ctx, 'func, Auto<'ctx>> {
from_auto!(Void);
from_auto!(U8);
from_auto!(U16);
from_auto!(U32);
from_auto!(U64);
from_auto!(Usize);
from_auto!(I8);
from_auto!(I16);
from_auto!(I32);
from_auto!(I64);
from_auto!(Isize);
pub fn into_aggregate(&self) -> Value<'ctx, 'func, Aggregate<'ctx>> {
if let Type::Aggregate(id) = self.ty.inner {
return Value {
id: self.id,
ty: Aggregate(id),
_mark: PhantomData,
```

```
};
}
panic!("value is not aggregate")
}
pub fn into_interface(&self) -> Value<'ctx, 'func, Interface<'ctx>> {
if let Type::Interface(id) = self.ty.inner {
return Value {
id: self.id,
ty: Interface(id),
_mark: PhantomData,
};
}
panic!("value is not interface")
}
pub fn into_pointer<T: MarkerType<'ctx>>(&self, ty: T) -> Value<'ctx, 'func,
Pointer<T>> {
if let Type::Pointer(t) = &self.ty.inner {
if t.as_ref() == &ty.to_type() {
return Value {
id: self.id,
ty: Pointer { pointee: ty },
_mark: PhantomData,
};
}
}
panic!("value is not pointer")
}

pub fn into_smart_pointer<T: MarkerType<'ctx>>(&self, ty: T) -> Value<'ctx,
'func, Smart<T>> {
if let Type::Pointer(t) = &self.ty.inner {
if t.as_ref() == &ty.to_type() {
return Value {
id: self.id,
ty: Smart { pointee: ty },
_mark: PhantomData,
};
}
}
panic!("value is not pointer")
}

pub fn into_function<Arg: FunctionArgs<'ctx>, R: MarkerType<'ctx>>(
&self,
```

```
args: Arg,
return_: R,
) -> Value<'ctx, 'func, Function<'ctx, Arg, R>> {
if let Type::Function(f) = &self.ty.inner {
if f.params.len() != args.len() {
panic!("value is function, but wrong number of arguments provided")
}
for (i, ty) in f.params.iter().enumerate() {
if ty != &args.get(i) {
panic!("value is function, but argument {} has wrong type", i)
}
}
if f.return_ != return_.to_type() {
panic!("value is function, but return type does not match")
}

return Value {
id: self.id,
ty: Function {
args,
return_,
_mark: PhantomData,
},
_mark: PhantomData,
};
}
panic!("value is not function")
}

pub fn into_array<T: MarkerType<'ctx>>(&self, ty: T) -> Value<'ctx, 'func,
Array<T>> {
if let Type::Array(a) = &self.ty.inner {
if &a.0 != &ty.to_type() {
panic!("value is array but element type does not match")
}
return Value {
id: self.id,
ty: Array {
element: ty,
length: a.1,
},
_mark: PhantomData,
};
}
```

```
panic!("value is not array")
}

pub fn into_future<T: MarkerType<'ctx>>(&self, ty: T) -> Value<'ctx, 'func,
Future<T>> {
if let Type::Future(t) = &self.ty.inner {
if t.as_ref() == &ty.to_type() {
return Value {
id: self.id,
ty: Future { value: ty },
_mark: PhantomData,
};
}
}
panic!("value is not future")
}
pub fn into_generator<Y: MarkerType<'ctx>, RE: MarkerType<'ctx>, R:
MarkerType<'ctx>>(&self, yield_ty: Y, resume_ty: RE, return_ty: R) ->
Value<'ctx, 'func, Generator<Y, RE, R>> {
if let Type::Generator(gen) = &self.ty.inner {
if &gen.0 != &yield_ty.to_type() || &gen.1 != &resume_ty.to_type() || &gen.2 !=
&return_ty.to_type(){
panic!("value is generator but type does not match")
}

return Value{
id: self.id,
ty: Generator {
yield_: yield_ty,
resume: resume_ty,
return_: return_ty
},
_mark: PhantomData
}
}
panic!("value is not generator")
}

pub fn into_simd<I: ScalarMarkerType, const N: usize>(&self) -> Value<'ctx,
'func, SIMD<I, N>>
where
simd::LaneCount<N>: simd::SupportedLaneCount,
{
let valid = match &self.ty.inner {
```

```
Type::SIMDx2(t) => t == &I::TY && N == 2,
Type::SIMDx4(t) => t == &I::TY && N == 4,
Type::SIMDx8(t) => t == &I::TY && N == 8,
Type::SIMDx16(t) => t == &I::TY && N == 16,
Type::SIMDx32(t) => t == &I::TY && N == 32,
Type::SIMDx64(t) => t == &I::TY && N == 64,
Type::SIMDx128(t) => t == &I::TY && N == 128,
Type::SIMDx256(t) => t == &I::TY && N == 256,
_ => false,
};
if valid {
return Value {
id: self.id,
ty: SIMD::default(),
_mark: PhantomData,
};
}
panic!("value is not SIMD")
}
}
```

5.3.3 native-ts-mir/utils.rs

```
use core::marker::PhantomData;
use core::sync::atomic::{AtomicUsize, Ordering};
use std::hash::Hash;
use std::hash::Hasher;

#[repr(C)]
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub enum Size {
Fixed(usize),
PointerSize,
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct AggregateID<'ctx> {
pub(super) id: usize,
pub(super) _mark: PhantomData<&'ctx ()>,
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct InterfaceID<'ctx> {
pub(super) id: usize,
pub(super) _mark: PhantomData<&'ctx ()>,
```

```
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct BlockID(usize);

impl BlockID {
pub(crate) fn new() -> Self {
static COUNT: AtomicUsize = AtomicUsize::new(1);
Self(COUNT.fetch_add(1, Ordering::SeqCst))
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct ValueID(pub(crate) usize);

impl ValueID {
pub(crate) fn new() -> Self {
static COUNT: AtomicUsize = AtomicUsize::new(1);
Self(COUNT.fetch_add(1, Ordering::SeqCst))
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct StackSlotID(pub(crate) usize);

#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct FunctionID<'ctx> {
pub(crate) id: usize,
pub(super) _mark: PhantomData<&'ctx ()>,
}


#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct Ident(u64);

impl Ident {
/// from string
pub fn from_str(s: &str) -> Self {
let mut hasher = std::collections::hash_map::DefaultHasher::new();

hasher.write_u8(1);
s.hash(&mut hasher);

return Self(hasher.finish());
}
```

```
/// from index
pub fn from_index(i: usize) -> Self {
let mut hasher = std::collections::hash_map::DefaultHasher::new();
hasher.write_u8(0);
hasher.write_usize(i);

Self(hasher.finish())
}
}
```

5.3.4 native-ts-mir/runtime.rs

```
use crate::util::{AggregateID, FunctionID};

#[derive(Debug, Default)]
pub struct Runtime<'ctx> {
pub memory_management: MemoryManagement<'ctx>,
pub async_runtime: Option<AsyncRuntime<'ctx>>,
}

#[derive(Debug)]
pub enum MemoryManagement<'ctx> {
/// manual memory management
Manual,
/// automatic reference counting
Arc {
/// should have type fn(usize) -> *mut u8
malloc: FunctionID<'ctx>,
/// pointer may contain header
ptr_offset: isize,
/// should have type fn(*mut u8)
increment_count: FunctionID<'ctx>,
/// should have type fn(*mut u8)
decrement_count: FunctionID<'ctx>,
},
/// garbage collector
GarbageCollect {
/// should have type fn(usize) -> *mut u8
malloc: FunctionID<'ctx>,
/// should have type fn(obj: *mut u8, offset: usize, child: *mut u8)
write_barrier: FunctionID<'ctx>,
/// pointer may contain header
ptr_offset: isize,
/// should have type fn()
```

```
safepoint: FunctionID<'ctx>,
},
}

impl<'ctx> Default for MemoryManagement<'ctx> {
fn default() -> Self {
return Self::Manual;
}
}

#[derive(Debug)]
pub struct AsyncRuntime<'ctx> {
/// the type of a future
pub future_type: AggregateID<'ctx>,
/// create a future.
/// should have type fn() -> future_type
pub create_future: FunctionID<'ctx>,
/// resolve the future. The first argument is the pointer to the result.
/// should have type fn(furute_type, *mut u8)
pub resolve_future: FunctionID<'ctx>,
/// awaits a future, returns the result pointer, null if future is not resolved.
/// should have type fn(future_type) -> *mut u8
pub await_future: FunctionID<'ctx>,
}
```

5.3.5 native-ts-mir/mir.rs

```
use crate::util::*;

#[repr(u8)]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum ICond {
EQ,
NE,
GT,
GTEQ,
LT,
LTEQ,
}

#[repr(u8)]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum FCond {
/// ordered and equal
OEQ,
```

```
/// ordered and greater than
OGT,
/// ordered and greater than or equal
OGE,
/// ordered and less than
OLT,
/// ordered and less than or equal
OLE,
/// ordered and not equal
ONE,
/// ordered (no NaN)
ORD,
/// unordered or equal
UEQ,
/// unordered or greater than
UGT,
/// unordered or greater than or equal
UGE,
/// unordered or less than
ULT,
/// unordered or less than or equal
ULE,
/// unordered or nor equal
UNE,
/// unordered (either NaN)
UNO,
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum Ordering {
Acquire,
Release,
AcqRel,
SeqCst,
}

#[derive(Debug, Clone)]
pub struct SwitchCase {
pub test: i64,
pub block: BlockID,
pub block_args: Vec<ValueID>,
}

#[derive(Debug, Clone)]
```

```
pub enum MIR<'ctx> {
/// read param
ReadParam(usize, ValueID),
Uconst(u128, ValueID),
Iconst(i128, ValueID),
F64const(f64, ValueID),
F32const(f32, ValueID),

/// data stored in raw bytes,
/// convert back when use
Vconst(Box<[u8]>, ValueID),

/// negative value
Neg(ValueID, ValueID),
/// absolute value
Abs(ValueID, ValueID),

Add(ValueID, ValueID, ValueID),
Sub(ValueID, ValueID, ValueID),
Mul(ValueID, ValueID, ValueID),
Exp(ValueID, ValueID, ValueID),
Rem(ValueID, ValueID, ValueID),
Div(ValueID, ValueID, ValueID),
Shl(ValueID, ValueID, ValueID),
Shr(ValueID, ValueID, ValueID),

/// bitand
Bitand(ValueID, ValueID, ValueID),
/// bitor
BitOr(ValueID, ValueID, ValueID),
Bitxor(ValueID, ValueID, ValueID),
/// bitnot
Bitnot(ValueID, ValueID),
/// bit reverse
Bitrev(ValueID, ValueID),
/// swap the order of bytes
Bitswap(ValueID, ValueID),
/// count number of ones
BitOnes(ValueID, ValueID),
/// count number of leading zeros
BitLeadingZeros(ValueID, ValueID),
/// count number of trailing zeros
BitTrailingZeros(ValueID, ValueID),
/// bitcast a value, both type must have the same size
```

```
Bitcast(ValueID, ValueID),

/// compare two integers, returns a bool
Icmp(ICond, ValueID, ValueID, ValueID),
/// compare two floats, returns a bool
Fcmp(FCond, ValueID, ValueID, ValueID),

/// return the minimum value.
/// when value type is float, if either value is NaN, NaN is returned.
Min(ValueID, ValueID, ValueID),
/// return the maximum value.
/// when value type is float, if either value is NaN, NaN is returned.
Max(ValueID, ValueID, ValueID),
/// if test value is true, return left side, otherwise right.
Select(ValueID, ValueID, ValueID, ValueID),
BitSelect(ValueID, ValueID, ValueID, ValueID),

// float operations
Sqrt(ValueID, ValueID),
Sin(ValueID, ValueID),
Cos(ValueID, ValueID),
Powi(ValueID, ValueID, ValueID),
Powf(ValueID, ValueID, ValueID),
Floor(ValueID, ValueID),
Ceil(ValueID, ValueID),
Round(ValueID, ValueID),

/// converts int to float
IntToFloat(ValueID, ValueID),
/// converts float to int
FloatToInt(ValueID, ValueID),
/// converts from one int type to another
IntCast(ValueID, ValueID),
/// converts from f64 to f32 or f32 to f64
FloatCast(ValueID, ValueID),

/// extract an element from vector
ExtractElement(ValueID, u8, ValueID),
/// insert element to vector
InsertElement(ValueID, ValueID, u8, ValueID),

/// creates an aggregate structure value
Aggregate(Box<[ValueID]>, ValueID),
/// converts an aggregate pointer to interface
```

```
Interface(ValueID, ValueID),
/// extracts a field from either aggregate or interface
ExtractValue(ValueID, Ident, ValueID),
/// inserts a value to field to either aggregate or interface
InsertValue(ValueID, Ident, ValueID),

/// converts one interface to another
AggregateToInterface(ValueID, InterfaceID<'ctx>, ValueID),
/// converts one interface to another
InterfaceToInterface(ValueID, InterfaceID<'ctx>, ValueID),

CreateStackSlot(StackSlotID, ValueID),
/// loads from the stack
StackLoad(StackSlotID, u64, ValueID),
/// stores to the stack
StackStore(StackSlotID, u64, ValueID),

// (slot, result)
/// get the location of stackslot
StackPtr(StackSlotID, ValueID),

// (pointer, result)
/// loads a value from location
Load(ValueID, ValueID),
// (pointer, value)
/// stores a value to location
Store(ValueID, ValueID),

/// calculates the pointer to elements with offsets.
ElementPtr(ValueID, Box<[usize]>),

/// fence
AtomicFence(Ordering),
/// (pointer, cmp, new, sucess ordering, failure ordering, loaded value, sucess)
///
/// compare exchange
AtomicCompareExchange(
Box<(
ValueID,
ValueID,
ValueID,
Ordering,
Ordering,
ValueID,
```

```
ValueID,
)>,
),

/// unconditionally branch to a block
Jump(BlockID),
/// branch if zero
Brz(ValueID, BlockID, BlockID),
/// branch if not zero
Brnz(ValueID, BlockID, BlockID),
Switch(ValueID, Box<[SwitchCase]>),
/// return a value or void
Return(Option<ValueID>),

/// function call
Call {
id: FunctionID<'ctx>,
args: Box<[ValueID]>,
return_: ValueID,
},
CallIndirect {
func: ValueID,
args: Box<[ValueID]>,
return_: ValueID,
},

/// allocate a smart pointer
Malloc(ValueID, ValueID),
/// if memory management is manual, free the pointer.
/// otherwise, this is noop
Free(ValueID),
/// await for a future
AsyncAwait(ValueID, ValueID),
/// yield from a generator and wait for resume
Yield(ValueID, ValueID),

/// generator, resume, result
///
/// returns an enum of yield type or return type
GeneratorNext(ValueID, ValueID, ValueID),
}
```

5.3.6 native-ts-mir/function.rs

```
use std::marker::PhantomData;

use crate::context::GeneratorDesc;
use crate::mir::MIR;
use crate::types::*;
use crate::util::*;

pub(crate) struct SSA<'ctx>{
pub id: ValueID,
pub ty: Type<'ctx>
}

pub(crate) struct BlockDesc<'ctx> {
pub(crate) id: BlockID,
pub(crate) inst: Vec<MIR<'ctx>>,
}

impl<'ctx> BlockDesc<'ctx> {
pub fn new(id: BlockID, params: &[Type<'ctx>]) -> Self {
let mut param_values = Vec::new();
let mut values = Vec::new();

for i in 0..params.len() {
let id = ValueID::new();

param_values.push(id);
values.push((id, params[i].clone()));
}

Self {
id,
inst: Vec::new(),
}
}
}

pub struct Function<'ctx> {
pub(crate) params: Vec<Type<'ctx>>,
pub(crate) return_: Type<'ctx>,
pub(crate) is_async: bool,
pub(crate) is_generator: Option<GeneratorDesc<'ctx>>,
pub(crate) map_ssa_func: Vec<(FunctionID<'ctx>, ValueID)>,
pub(crate) blocks: Vec<BlockDesc<'ctx>>,
```

```
pub(crate) stackslots: Vec<Type<'ctx>>,
pub(crate) _mark: PhantomData<&'ctx ()>,
}

impl<'ctx> Function<'ctx> {
pub fn new(params: &[Type<'ctx>], return_ty: Type<'ctx>, is_async: bool,
is_generator: Option<GeneratorDesc<'ctx>>) -> Self{
Self {
params: params.to_vec(),
return_: return_ty,
is_async: is_async,
is_generator: is_generator,
map_ssa_func: Vec::new(),
blocks: Vec::new(),
stackslots: Vec::new(),
_mark: PhantomData
}
}

/// return true if function is async
pub fn is_async(&self) -> bool{
self.is_async
}

/// return true if fucntion is generator
pub fn is_generator(&self) -> bool{
self.is_generator.is_some()
}

/// return the resume type if function is generator
pub fn generator_resume_type(&self) -> Option<Type<'ctx>>{
self.is_generator.as_ref().map(|w|w.resume_type.clone())
}

/// return the yield type if function is generator
pub fn generator_yield_type(&self) -> Option<Type<'ctx>>{
self.is_generator.as_ref().map(|w|w.yield_type.clone())
}
}
```

5.3.7 native-ts-mir/context.rs

```
use core::marker::PhantomData;

use crate::{
```

```
function::Function,
types::{
aggregate::{AggregateDesc, InterfaceDesc},
FunctionType,
},
util::{AggregateID, FunctionID, InterfaceID},
Type, backend::Backend,
};

#[repr(C)]
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord)]
pub enum Linkage {
/// private symbol
Private,
/// internal
Internal,
AvailableExternally,
LinkOnce,
Weak,
Common,
Appending,
ExternWeak,
LinkOnceOdr,
WeakOdr,
External,
/// export to dll
DLLExport,
/// import from dll
DLLImport
}

pub(crate) struct FunctionDesc<'ctx> {
pub(crate) name: Option<String>,
pub(crate) is_async: bool,
pub(crate) is_generator: Option<GeneratorDesc<'ctx>>,

pub(crate) function: Option<Function<'ctx>>,
pub(crate) ty: FunctionType<'ctx>,
pub(crate) linkage: Option<Linkage>,
}

pub struct GeneratorDesc<'ctx>{
pub resume_type: Type<'ctx>,
pub yield_type: Type<'ctx>,
```

```
}

pub struct Context {
pub(crate) aggregates: Vec<AggregateDesc<'static>>,
pub(crate) interfaces: Vec<InterfaceDesc<'static>>,

pub(crate) functions: Vec<FunctionDesc<'static>>,
}

impl Context {
pub fn new() -> Self {
Self {
aggregates: Vec::new(),
interfaces: Vec::new(),

functions: Vec::new(),
}
}

/// define an aggregate type
pub fn declare_aggregate<'ctx>(&mut self, desc: AggregateDesc<'ctx>) ->
AggregateID {
// check if aggregate already declared
for (i, agg) in self.aggregates.iter().enumerate(){
if agg.hash == desc.hash{
return AggregateID{
id: i,
_mark: PhantomData
}
}
}

// get the id
let id = self.aggregates.len();
// push aggregate
self.aggregates.push(unsafe { core::mem::transmute(desc) });
return AggregateID {
id: id,
_mark: PhantomData,
};
}

/// get the aggreagate descriptor
```

```
pub fn get_aggregate<'ctx>(&'ctx self, id: AggregateID<'ctx>) ->
&AggregateDesc {
self.aggregates.get(id.id).expect("invalid aggregate id")
}

/// define an interface type
pub fn declare_interface<'ctx>(&'ctx mut self, desc: InterfaceDesc<'ctx>) ->
InterfaceID {
// check if interface already declared
for (i, iface) in self.interfaces.iter().enumerate(){
if iface.hash == desc.hash{
return InterfaceID{
id: i,
_mark: PhantomData
}
}
}

// get the id
let id = self.interfaces.len();
// push the interface
self.interfaces.push(unsafe { core::mem::transmute(desc) });
// return wrapped id
return InterfaceID {
id: id,
_mark: PhantomData,
};
}

/// get the interface descriptor
pub fn get_interface<'ctx>(&'ctx self, id: InterfaceID<'ctx>) -> &InterfaceDesc
{
self.interfaces.get(id.id).expect("invalid interface id")
}

/// declare a function
pub fn declare_function<'ctx, S: Into<String>>(
&'ctx mut self,
name: Option<S>,
params: &[Type<'ctx>],
return_ty: Type<'ctx>,
is_async: bool,
is_generator: Option<GeneratorDesc<'ctx>>,
linkage: Option<Linkage>,
```

```
) -> FunctionID {
let id = self.functions.len();

self.functions.push(FunctionDesc {
name: name.map(|s| s.into()),
is_async: is_async,
is_generator: unsafe{core::mem::transmute(is_generator)},
function: None,
ty: unsafe {
core::mem::transmute(FunctionType {
params: params.into(),
return_: return_ty,
})
},
linkage
});
return FunctionID {
id: id,
_mark: PhantomData,
};
}

/// define a function
pub fn define_function<'ctx>(&'ctx mut self, id: FunctionID<'ctx>, func:
Function<'ctx>) {
if let Some(desc) = self.functions.get_mut(id.id){
if desc.is_async != func.is_async{
panic!("function is not async but declared as async")
}
if let Some(gen) = &desc.is_generator{
if let Some(fgen) = &func.is_generator{
if fgen.resume_type != gen.resume_type{
panic!("generator resume type mismatch")
}
if fgen.yield_type != gen.yield_type{
panic!("generator yield type mismatch")
}
} else{
panic!("function is not generator but declared as generator")
}
}
if desc.ty.params.as_ref() != &func.params{
panic!("function params does not match")
}
```

```
if desc.ty.return_ != func.return_{
panic!("function return type mismatch")
}
desc.function = Some(unsafe{core::mem::transmute(func)});
} else{
panic!("invalid function id")
}
}

/// get the function if defined
pub fn get_function<'ctx>(&'ctx self, id: FunctionID<'ctx>) -> Option<&'ctx
Function> {
self.functions.get(id.id).expect("invalid function id").function.as_ref()
}

/// get the function if defined
pub fn get_function_by_name<'ctx>(&'ctx self, name: &str) -> Option<&'ctx
Function>{
for f in &self.functions{
if let Some(n) = &f.name{
if name == n{
return f.function.as_ref()
}
}
}

return None
}

pub fn get_function_type<'ctx>(&'ctx self, id: FunctionID<'ctx>) -> &'ctx
FunctionType {
&self.functions.get(id.id).expect("invalid function id").ty
}

pub fn create_function<'ctx>(&'ctx self, params: &[Type<'ctx>], return_ty:
Type<'ctx>, is_async: bool, is_generator: Option<GeneratorDesc<'ctx>>) ->
Function{
Function{
params: params.to_vec(),
return_: return_ty,
is_async,
is_generator,
map_ssa_func: Vec::new(),
blocks: Vec::new(),
```

```
stackslots: Vec::new(),
_mark: PhantomData
}
}

pub fn compile<B: Backend>(&self, mut backend: B) -> Result<B::Output,
String>{
backend.compile(self)
}
}
```

5.3.8 native-ts-mir/builder.rs

```
use std::cell::RefCell;
use std::marker::PhantomData;

use types::{Auto, AutoArgs, Enum, FunctionType, Future, Generator,
PointerMarkerType, Smart, ValueIndex};

use crate::function::{BlockDesc, Function};
use crate::mir::{FCond, ICond, Ordering, MIR};
use crate::types::simd::{LaneCount, SupportedLaneCount};
use crate::types::{
Aggregate, FieldedMarkerType, FloatMarkerType, FloatMathMarkerType,
IntMarkerType,
IntMathMarkerType, Interface, IntoFloatMarkerType, IntoIntMarkerType,
IntoScalarMarkerType,
MarkerType, MathMarkerType, Pointer, ScalarMarkerType, Type, I32, I8, SIMD,
};
use crate::util::{AggregateID, BlockID, FunctionID, Ident, InterfaceID,
StackSlotID, ValueID};
pub use crate::Value;
use crate::{types, Context};

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Block<'func> {
id: BlockID,
_mark: PhantomData<&'func ()>,
}

#[derive(Clone)]
pub struct StackSlot<'ctx, 'func, T: MarkerType<'ctx>> {
id: StackSlotID,
ty: T,
_mark: PhantomData<(&'func (), &'ctx ())>,
```

```
}

pub struct Builder<'ctx, 'func>
where
'ctx: 'func,
{
ctx: &'ctx Context,
func: RefCell<&'func mut Function<'ctx>>,
current_block: RefCell<Option<&'func mut BlockDesc<'ctx>>>,
}

impl<'ctx, 'func> Builder<'ctx, 'func>
where
'ctx: 'func,
{
pub fn new(ctx: &'ctx Context, func: &'func mut Function<'ctx>) -> Self{
Self {
ctx: ctx,
func: RefCell::new(func),
current_block: RefCell::new(None)
}
}
pub fn create_block(&self) -> Block<'func> {
let id = BlockID::new();
self.func.borrow_mut().blocks.push(BlockDesc::new(id, &[]));

return Block {
id,
_mark: PhantomData,
};
}

pub fn switch_to_block(&self, block: Block<'func>) {
let mut f = self
.func
.borrow_mut();
let block: &mut BlockDesc =
f
.blocks
.iter_mut()
.rev()
.find(|b| b.id == block.id)
.expect("Trying to get instruction builder without declaring block");
```

```
self.current_block.replace(unsafe { core::mem::transmute(block) });
}

pub fn global_function(&self, func_id: FunctionID<'ctx>) -> Value<'ctx, 'func,
types::Function<'ctx, AutoArgs<'ctx>, Auto<'ctx>>>{
let f = &self.ctx.functions[func_id.id];
let params = f.ty.params.clone();
let mut return_ty = f.ty.return_.clone();

if f.is_async{
return_ty = Type::Future(Box::new(return_ty));
}

if let Some(gen) = &f.is_generator{
return_ty = Type::Generator(Box::new((
if f.is_async{
Type::Future(Box::new(gen.yield_type.clone()))
} else{
gen.yield_type.clone()
},
gen.resume_type.clone(),
return_ty
)));
}

let id = ValueID::new();
self.func.borrow_mut().map_ssa_func.push((func_id, id));

return Value {
id,
ty: types::Function{
args: AutoArgs(params),
return_: Auto { inner: return_ty.clone(),
},
_mark: PhantomData
}, _mark: PhantomData
}
}

/// # panic
pub fn inst<'builder>(&'builder mut self) -> InstBuilder<'ctx, 'func, 'builder> {
if self.current_block.get_mut().is_none(){
panic!("missing entry block")
}
```

```
InstBuilder {
builder: self,
block: unsafe{core::mem::transmute_copy(&self.current_block)},
_mark: PhantomData,
}
}
}
pub struct InstBuilder<'ctx, 'func, 'builder>
where
'ctx: 'func,
'func: 'builder,
{
builder: &'builder Builder<'ctx, 'func>,
block: RefCell<&'builder mut crate::function::BlockDesc<'ctx>>,
_mark: PhantomData<&'func ()>,
}

impl<'ctx, 'func, 'builder> InstBuilder<'ctx, 'func, 'builder>
where
'ctx: 'func,
{
fn new_ssa(&self, _ty: Type<'ctx>) -> ValueID{
ValueID::new()
}

/// read from param
pub fn param(&self, index: usize) -> Option<Value<'ctx, 'func, Auto<'ctx>>>{
if let Some(ty) = self.builder.func.borrow().params.get(index){
let ty = ty.clone();
let id = self.new_ssa(ty.clone());

self.block.borrow_mut().inst.push(MIR::ReadParam(index, id));

return Some(Value {
id: id,
ty: Auto { inner: ty },
_mark: PhantomData
})
}
return None;
}

/// an integer constant
```

```
pub fn iconst<I: IntoIntMarkerType>(&self, value: I) -> Value<'ctx, 'func,
I::Marker> {
let id = self.new_ssa(I::Marker::default().to_type());
self.block.borrow_mut().inst.push(MIR::Iconst(value.to_i128(), id));

return Value {
id: id,
ty: I::Marker::default(),
_mark: PhantomData,
};
}

/// a floating point constant
pub fn fconst<F: IntoFloatMarkerType>(&self, value: F) -> Value<'ctx, 'func,
F::Marker> {
let id = self.new_ssa(F::Marker::default().to_type());

if core::mem::size_of::<F>() == 8 {
let v = unsafe { *(&value as *const F as *const f64) };
self.block.borrow_mut().inst.push(MIR::F64const(v, id));
} else {
debug_assert!(core::mem::size_of::<F>() == 4);
let v = unsafe { *(&value as *const F as *const f32) };
self.block.borrow_mut().inst.push(MIR::F32const(v, id));
}
return Value {
id,
ty: F::Marker::default(),
_mark: PhantomData,
};
}

/// an simd constant
pub fn vconst<T: IntoScalarMarkerType, const N: usize>(
&self,
values: [T; N],
) -> Value<'ctx, 'func, SIMD<T::Marker, N>>
where
crate::types::simd::LaneCount<N>: crate::types::simd::SupportedLaneCount,
{
let id = self.new_ssa(SIMD::<T::Marker, N>::default().to_type());
unsafe {
let layout = alloc::alloc::Layout::for_value(&values);
let ptr = alloc::alloc::alloc(layout);
```

```
core::ptr::copy_nonoverlapping(
&values as *const [T; N] as *const u8,
ptr,
layout.size(),
);
let slice = core::slice::from_raw_parts_mut(ptr, layout.size());
let data = Box::from_raw(slice);
self.block.borrow_mut().inst.push(MIR::Vconst(data, id));

return Value {
id,
ty: SIMD::<T::Marker, N>::default(),
_mark: PhantomData,
};
};
}

/// negative value
pub fn neg<T: MathMarkerType>(
&self,
value: Value<'ctx, 'func, T>,
) -> Value<'ctx, 'func, T> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Neg(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}

/// absolute value
pub fn abs<T: MathMarkerType>(
&self,
value: Value<'ctx, 'func, T>,
) -> Value<'ctx, 'func, T> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Abs(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
```

```
};
}

pub fn add<T: MathMarkerType>(
&self,
a: Value<'ctx, 'func, T>,
b: Value<'ctx, 'func, T>,
) -> Value<'ctx, 'func, T> {
let id = self.new_ssa(a.ty.to_type());

self.block.borrow_mut().inst.push(MIR::Add(a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
pub fn sub<T: MathMarkerType>(
&self,
a: Value<'ctx, 'func, T>,
b: Value<'ctx, 'func, T>,
) -> Value<'ctx, 'func, T> {
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Sub(a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}

/// multiply
pub fn mul<T: MathMarkerType>(
&self,
a: Value<'ctx, 'func, T>,
b: Value<'ctx, 'func, T>,
) -> Value<'ctx, 'func, T> {
let id = self.new_ssa(a.ty.to_type());

self.block.borrow_mut().inst.push(MIR::Mul(a.id, b.id, id));

return Value {
```

```
id,
ty: a.ty,
_mark: PhantomData,
};
}

/// exponent
pub fn exp<T: MathMarkerType>(
&self,
a: Value<'ctx, 'func, T>,
b: Value<'ctx, 'func, T>,
) -> Value<'ctx, 'func, T> {
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Exp(a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}

/// remainder
pub fn rem<T: MathMarkerType>(
&self,
a: Value<'ctx, 'func, T>,
b: Value<'ctx, 'func, T>,
) -> Value<'ctx, 'func, T> {
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Rem(a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
/// division
pub fn div<T: MathMarkerType>(
&self,
a: Value<'ctx, 'func, T>,
b: Value<'ctx, 'func, T>,
) -> Value<'ctx, 'func, T> {
let id = self.new_ssa(a.ty.to_type());
```

```
self.block.borrow_mut().inst.push(MIR::Div(a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
/// shift left
pub fn shl<I: IntMathMarkerType>(
&self,
a: Value<'ctx, 'func, I>,
b: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Shl(a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
/// shift right
pub fn shr<I: IntMathMarkerType>(
&self,
a: Value<'ctx, 'func, I>,
b: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Shr(a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
/// bitwise and
pub fn bitand<I: IntMathMarkerType>(
&self,
a: Value<'ctx, 'func, I>,
b: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
```

```
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Bitand(a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
/// bitwise or
pub fn bitor<I: IntMathMarkerType>(
&self,
a: Value<'ctx, 'func, I>,
b: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst.push(MIR::BitOr(a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
/// bitwise xor
pub fn bitxor<I: IntMathMarkerType>(
&self,
a: Value<'ctx, 'func, I>,
b: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Bitxor(a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
/// bitwise not
pub fn bitnot<I: IntMathMarkerType>(
&self,
value: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
```

```
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Bitnot(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// bitwise reverse
pub fn bitrev<I: IntMathMarkerType>(
&self,
value: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Bitrev(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// bitwise swap
pub fn bitswap<I: IntMathMarkerType>(
&self,
value: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Bitswap(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// count one bits
pub fn count_ones<I: IntMathMarkerType>(
&self,
value: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::BitOnes(value.id, id));
```

```
return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// count leading zero bits
pub fn leading_zeros<I: IntMathMarkerType>(
&self,
value: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::BitLeadingZeros(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// count trailing zero bits
pub fn trailing_zeros<I: IntMathMarkerType>(
&self,
value: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::BitTrailingZeros(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// cast a type to another
pub fn bitcast<T: MarkerType<'ctx>, U: MarkerType<'ctx>>(
&self,
value: Value<'ctx, 'func, T>,
ty: U,
) -> Value<'ctx, 'func, U> {
let id = self.new_ssa(ty.to_type());
self.block.borrow_mut().inst.push(MIR::Bitcast(value.id, id));
```

```
return Value {
id,
ty: ty,
_mark: PhantomData,
};
}

pub fn icmp<I: IntMathMarkerType>(
&self,
cond: ICond,
a: Value<'ctx, 'func, I>,
b: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Icmp(cond, a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
pub fn fcmp<F: FloatMathMarkerType>(
&self,
cond: FCond,
a: Value<'ctx, 'func, F>,
b: Value<'ctx, 'func, F>,
) -> Value<'ctx, 'func, F> {
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Fcmp(cond, a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}

pub fn min<I: MathMarkerType>(
&self,
a: Value<'ctx, 'func, I>,
b: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(a.ty.to_type());
```

```
self.block.borrow_mut().inst.push(MIR::Min(a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
pub fn max<I: MathMarkerType>(
&self,
a: Value<'ctx, 'func, I>,
b: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Max(a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
pub fn select<I: IntMathMarkerType>(
&self,
test: Value<'ctx, 'func, I8>,
a: Value<'ctx, 'func, I>,
b: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Select(test.id, a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
/// bitwise select
pub fn bitselect<I: IntMathMarkerType>(
&self,
test: Value<'ctx, 'func, I>,
a: Value<'ctx, 'func, I>,
b: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, I> {
```

```
let id = self.new_ssa(a.ty.to_type());
self.block.borrow_mut().inst
.push(MIR::BitSelect(test.id, a.id, b.id, id));

return Value {
id,
ty: a.ty,
_mark: PhantomData,
};
}
/// sqroot
pub fn sqrt<F: FloatMathMarkerType>(
&self,
value: Value<'ctx, 'func, F>,
) -> Value<'ctx, 'func, F> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Sqrt(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// sin in radiant
pub fn sin<F: FloatMathMarkerType>(
&self,
value: Value<'ctx, 'func, F>,
) -> Value<'ctx, 'func, F> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Sin(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// cos in radiant
pub fn cos<F: FloatMathMarkerType>(
&self,
value: Value<'ctx, 'func, F>,
) -> Value<'ctx, 'func, F> {
let id = self.new_ssa(value.ty.to_type());
```

```
self.block.borrow_mut().inst.push(MIR::Cos(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// power to integer
pub fn powi<F: FloatMarkerType>(
&self,
value: Value<'ctx, 'func, F>,
exponent: Value<'ctx, 'func, I32>,
) -> Value<'ctx, 'func, F> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Powi(value.id, exponent.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// power to float
pub fn powf<F: FloatMathMarkerType>(
&self,
value: Value<'ctx, 'func, F>,
exponent: Value<'ctx, 'func, F>,
) -> Value<'ctx, 'func, F> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Powf(value.id, exponent.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// floor
pub fn floor<F: FloatMarkerType>(
&self,
value: Value<'ctx, 'func, F>,
) -> Value<'ctx, 'func, F> {
let id = self.new_ssa(value.ty.to_type());
```

```
self.block.borrow_mut().inst.push(MIR::Floor(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// ceil
pub fn ceil<F: FloatMarkerType>(
&self,
value: Value<'ctx, 'func, F>,
) -> Value<'ctx, 'func, F> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Ceil(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
/// round
pub fn round<F: FloatMarkerType>(
&self,
value: Value<'ctx, 'func, F>,
) -> Value<'ctx, 'func, F> {
let id = self.new_ssa(value.ty.to_type());
self.block.borrow_mut().inst.push(MIR::Round(value.id, id));

return Value {
id,
ty: value.ty,
_mark: PhantomData,
};
}
pub fn int_to_float<F: IntoFloatMarkerType, I: IntMarkerType>(
&self,
value: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, F::Marker> {
let id = self.new_ssa(F::Marker::default().to_type());
self.block.borrow_mut().inst.push(MIR::IntToFloat(value.id, id));

return Value {
```

```
id,
ty: F::Marker::default(),
_mark: PhantomData,
};
}
pub fn float_to_int<I: IntoIntMarkerType, F: FloatMarkerType>(
&self,
value: Value<'ctx, 'func, F>,
) -> Value<'ctx, 'func, I::Marker> {
let id = self.new_ssa(I::Marker::default().to_type());
self.block.borrow_mut().inst.push(MIR::FloatToInt(value.id, id));

return Value {
id,
ty: I::Marker::default(),
_mark: PhantomData,
};
}
/// extend or reduce bits to cast integer to another
pub fn int_cast<U: IntoIntMarkerType, I: IntMarkerType>(
&self,
value: Value<'ctx, 'func, I>,
) -> Value<'ctx, 'func, U::Marker> {
let id = self.new_ssa(U::Marker::default().to_type());
self.block.borrow_mut().inst.push(MIR::IntCast(value.id, id));

return Value {
id,
ty: U::Marker::default(),
_mark: PhantomData,
};
}
/// cast between f64 and f32
pub fn float_cast<U: IntoFloatMarkerType, F: FloatMarkerType>(
&self,
value: Value<'ctx, 'func, F>,
) -> Value<'ctx, 'func, U::Marker> {
let id = self.new_ssa(U::Marker::default().to_type());
self.block.borrow_mut().inst.push(MIR::FloatCast(value.id, id));

return Value {
id,
ty: U::Marker::default(),
_mark: PhantomData,
```

```
};
}
/// extract an element from vector
pub fn extract_element<T: ScalarMarkerType, const N: usize>(
&self,
vector: Value<'ctx, 'func, SIMD<T, N>>,
index: u8,
) -> Value<'ctx, 'func, T>
where
LaneCount<N>: SupportedLaneCount,
{
if index as usize >= N {
panic!("index larger then lanes")
}
let id = self.new_ssa(T::default().to_type());
self.block.borrow_mut().inst
.push(MIR::ExtractElement(vector.id, index as _, id));

return Value {
id,
ty: T::default(),
_mark: PhantomData,
};
}
/// insert an element to vector
pub fn insert_element<T: ScalarMarkerType, const N: usize>(
&self,
vector: Value<'ctx, 'func, SIMD<T, N>>,
value: Value<'ctx, 'func, T>,
index: u8,
) -> Value<'ctx, 'func, SIMD<T, N>>
where
LaneCount<N>: SupportedLaneCount,
{
if index as usize >= N {
panic!("index larger then lanes")
}
let id = self.new_ssa(vector.ty.to_type());
self.block.borrow_mut().inst
.push(MIR::InsertElement(vector.id, value.id, index as _, id));

return Value {
id,
ty: SIMD::<T, N>::default(),
```

```
_mark: PhantomData,
};
}
/// construct an aggregate type
pub fn aggregate(
&self,
ty: AggregateID<'ctx>,
values: &[Value<'ctx, 'func, Auto<'ctx>>],
) -> Value<'ctx, 'func, Aggregate<'ctx>> {
let agg = self.builder.ctx.get_aggregate(ty);

if values.len() != agg.fields.len() {
panic!("invalid arguments")
}

for (i, (_key, ty)) in agg.fields.iter().enumerate() {
if &values[i].ty.inner != ty {
panic!("mismatch type")
}
}
let id = self.new_ssa(Type::Aggregate(ty));

self.block.borrow_mut().inst
.push(MIR::Aggregate(values.iter().map(|v| v.id).collect(), id));

return Value {
id,
ty: Aggregate(ty),
_mark: PhantomData,
};
}
pub fn aggregate_to_interface(
&self,
value: Value<'ctx, 'func, Smart<Aggregate<'ctx>>>,
iface: InterfaceID<'ctx>,
) -> Value<'ctx, 'func, Interface<'ctx>> {
let agg = self.builder.ctx.get_aggregate(value.ty.pointee.0);
let interface = self.builder.ctx.get_interface(iface);

if agg.fields.len() < interface.fields.len() {
panic!("aggregate type does not match interface")
}

for (ident, ty) in &interface.fields {
```

```
if agg
.fields
.iter()
.find(|(k, t)| k == ident && t == ty)
.is_none()
{
panic!("aggregate type does not match interface")
}
}
let id = self.new_ssa(Type::Interface(iface));
self.block.borrow_mut().inst
.push(MIR::AggregateToInterface(value.id, iface, id));

return Value {
id: id,
ty: Interface(iface),
_mark: PhantomData,
};
}
pub fn interface_to_interface(
&self,
value: Value<'ctx, 'func, Interface<'ctx>>,
iface: InterfaceID<'ctx>,
) -> Value<'ctx, 'func, Interface<'ctx>> {
// same interface, no need to map.
if value.ty.0 == iface {
return value;
}

let iface1 = self.builder.ctx.get_interface(value.ty.0);
let iface2 = self.builder.ctx.get_interface(iface);

for (key, ty) in &iface2.fields {
if !iface1
.fields
.iter()
.find(|(id, t)| id == key && t == ty)
.is_some()
{
panic!("interface mismatch")
}
}

let id = self.new_ssa(Type::Interface(iface));
```

```
// insert instruction
self.block.borrow_mut().inst
.push(MIR::InterfaceToInterface(value.id, iface, id));

return Value {
id,
ty: Interface(iface),
_mark: PhantomData,
};
}

/// extract a field value from any fielded types.
/// Accepts aggregate, interface, pointer to aggregate or pointer to interface.
pub fn extract_value<T: FieldedMarkerType<'ctx>>(
&self,
target: Value<'ctx, 'func, T>,
field: Ident,
) -> Value<'ctx, 'func, Auto<'ctx>> {
let ty = match target.ty.to_type() {
Type::Aggregate(id) => {
if let Some((_, ty)) = self
.builder
.ctx
.get_aggregate(id)
.fields
.iter()
.find(|(key, _)| key == &field)
{
ty.clone()
} else {
panic!("aggregate has no field")
}
}
Type::Interface(id) => {
if let Some((_, ty)) = self
.builder
.ctx
.get_interface(id)
.fields
.iter()
.find(|(key, _)| key == &field)
{
ty.clone()
```

```
} else {
panic!("interface has no field")
}
}
Type::SmartPointer(p) | Type::Pointer(p) => match p.as_ref() {
Type::Aggregate(id) => {
if let Some((_, ty)) = self
.builder
.ctx
.get_aggregate(*id)
.fields
.iter()
.find(|(key, _)| key == &field)
{
ty.clone()
} else {
panic!("aggregate has no field")
}
}
Type::Interface(id) => {
if let Some((_, ty)) = self
.builder
.ctx
.get_interface(*id)
.fields
.iter()
.find(|(key, _)| key == &field)
{
ty.clone()
} else {
panic!("interface has no field")
}
}
_ => unreachable!(),
},
_ => unreachable!(),
};

// the lifetime of type is phanom and can be safely transmuted
let ty: Type<'_> = unsafe { core::mem::transmute(ty.clone())};
let id = self.new_ssa(ty.clone());

// insert instruction
self.block.borrow_mut().inst
```

```
.push(MIR::ExtractValue(target.id, field, id));

return Value {
id,
ty: Auto {
inner: ty,
},
_mark: PhantomData,
};
}

/// inserts a value to a field
/// Accepts aggregate, interface, pointer to aggregate or pointer to interface.
///
/// if garbage collection is enabled, this will be lowered to a call to write barrier
pub fn insert_value<T: FieldedMarkerType<'ctx>, V: MarkerType<'ctx>>(
&self,
target: Value<'ctx, 'func, T>,
field: Ident,
value: Value<'ctx, 'func, V>,
) {
let ty = match target.ty.to_type() {
Type::Aggregate(id) => {
if let Some((_, ty)) = self
.builder
.ctx
.get_aggregate(id)
.fields
.iter()
.find(|(key, _)| key == &field)
{
ty.clone()
} else {
panic!("aggregate has no field")
}
}
Type::Interface(id) => {
if let Some((_, ty)) = self
.builder
.ctx
.get_interface(id)
.fields
.iter()
.find(|(key, _)| key == &field)
```

```
{
ty.clone()
} else {
panic!("interface has no field")
}
}
Type::SmartPointer(p) | Type::Pointer(p) => match p.as_ref() {
Type::Aggregate(id) => {
if let Some((_, ty)) = self
.builder
.ctx
.get_aggregate(*id)
.fields
.iter()
.find(|(key, _)| key == &field)
{
ty.clone()
} else {
panic!("aggregate has no field")
}
}
Type::Interface(id) => {
if let Some((_, ty)) = self
.builder
.ctx
.get_interface(*id)
.fields
.iter()
.find(|(key, _)| key == &field)
{
ty.clone()
} else {
panic!("interface has no field")
}
}
_ => unreachable!(),
},
_ => unreachable!(),
};

if value.ty.to_type() != ty {
panic!("type not match")
}
```

```
self.block.borrow_mut().inst
.push(MIR::InsertValue(target.id, field, value.id));

return;
}

/// allocate a new stack slot
pub fn create_stack_slot<T: MarkerType<'ctx>>(
&self,
initialiser: Value<'ctx, 'func, T>,
) -> StackSlot<'ctx, 'func, T> {
let mut f = self.builder.func.borrow_mut();
let id = f.stackslots.len();
f.stackslots.push(initialiser.ty.to_type());

drop(f);

self.block.borrow_mut().inst
.push(MIR::CreateStackSlot(StackSlotID(id), initialiser.id));

StackSlot {
id: StackSlotID(id),
ty: initialiser.ty,
_mark: PhantomData,
}
}

/// load value from the stack slot
pub fn stack_load<T: MarkerType<'ctx>>(
&self,
slot: StackSlot<'ctx, 'func, T>,
) -> Value<'ctx, 'func, T> {
let id = self.new_ssa(slot.ty.to_type());
self.block.borrow_mut().inst.push(MIR::StackLoad(slot.id, 0, id));

return Value {
id,
ty: slot.ty.clone(),
_mark: PhantomData,
};
}
/// write value to the stack slot
pub fn stack_store<T: MarkerType<'ctx>>(
&self,
```

```
slot: StackSlot<'ctx, 'func, T>,
value: Value<'ctx, 'func, T>,
) {
self.block.borrow_mut().inst.push(MIR::StackStore(slot.id, 0, value.id));
}

/// retrieve pointer to a stack slot
pub fn stack_pointer<T: MarkerType<'ctx>>(
&self,
slot: StackSlot<'ctx, 'func, T>,
) -> Value<'ctx, 'func, Pointer<T>> {
let id = self.new_ssa(Type::Pointer(Box::new(slot.ty.to_type())));
self.block.borrow_mut().inst.push(MIR::StackPtr(slot.id, id));

return Value {
id,
ty: Pointer {
pointee: slot.ty.clone(),
},
_mark: PhantomData,
};
}

pub fn load<T: MarkerType<'ctx>, P: PointerMarkerType<'ctx, T>>(
&self,
ptr: Value<'ctx, 'func, P>,
) -> Value<'ctx, 'func, T> {
let id = self.new_ssa(ptr.ty.pointee().to_type());
self.block.borrow_mut().inst.push(MIR::Load(ptr.id, id));

return Value {
id,
ty: ptr.ty.pointee().clone(),
_mark: PhantomData,
};
}

/// stores a value
pub fn store<T: MarkerType<'ctx>, P: PointerMarkerType<'ctx, T>>(
&self,
ptr: Value<'ctx, 'func, P>,
value: Value<'ctx, 'func, T>,
) {
self.block.borrow_mut().inst.push(MIR::Store(ptr.id, value.id));
```

```
}

/// memory fence operation
pub fn fence(&self, ordering: Ordering) {
self.block.borrow_mut().inst.push(MIR::AtomicFence(ordering));
}
/// returns the loaded value and success
pub fn compare_exchange<T: IntMarkerType>(
&self,
ptr: Value<'ctx, 'func, Pointer<T>>,
cmp: Value<'ctx, 'func, T>,
new: Value<'ctx, 'func, T>,
success_order: Ordering,
failure_order: Ordering,
) -> (Value<'ctx, 'func, T>, Value<'ctx, 'func, I8>) {
let loaded_id = self.new_ssa(T::default().to_type());
let success_id = self.new_ssa(Type::I8);

self.block.borrow_mut().inst.push(MIR::AtomicCompareExchange(Box::new((
ptr.id,
cmp.id,
new.id,
success_order,
failure_order,
loaded_id,
success_id,
))));

return (
Value {
id: loaded_id,
ty: T::default(),
_mark: PhantomData,
},
Value {
id: success_id,
ty: I8,
_mark: PhantomData,
},
);
}

/// unconditional jump
pub fn jump(&self, block: Block<'func>) {
```

```
self.block.borrow_mut().inst.push(MIR::Jump(block.id))
}

/// branch if zero
pub fn brz<I: IntMarkerType>(
&self,
test: Value<'ctx, 'func, I>,
then: Block<'ctx>,
else_: Block<'ctx>,
) {
self.block.borrow_mut().inst.push(MIR::Brz(test.id, then.id, else_.id));
}

/// branch if non zero
pub fn brnz<I: IntMarkerType>(
&self,
test: Value<'ctx, 'func, I>,
then: Block<'ctx>,
else_: Block<'ctx>,
) {
self.block.borrow_mut().inst.push(MIR::Brnz(test.id, then.id, else_.id));
}

/// returns from a function
pub fn return_<T: MarkerType<'ctx>>(&self, value: Option<Value<'ctx, 'func,
T>>) {
self.block.borrow_mut().inst.push(MIR::Return(value.map(|v| v.id)));
}

/// calls a function indirectly
///
/// # panic
pub fn call_indirect<Arg: types::FunctionArgs<'ctx>, R: MarkerType<'ctx>>(
&self,
func: Value<'ctx, 'func, types::Function<'ctx, Arg, R>>,
args: &Arg::ArgValues<'func>,
) -> Value<'ctx, 'func, R> {
let args_len = args.len();
let ty_len = func.ty.args.len();

if args_len != ty_len {
panic!("arguments not match")
};
```

```
let mut arg_values = Vec::new();

for i in 0..ty_len {
let arg = args.get(i);
if arg.ty.inner != func.ty.args.get(i) {
panic!("argument type not match")
}
arg_values.push(arg.id);
}

let id = self.new_ssa(func.ty.return_.to_type());

self.block.borrow_mut().inst.push(MIR::CallIndirect {
func: func.id,
args: arg_values.into_boxed_slice(),
return_: id,
});
return Value {
id,
ty: func.ty.return_,
_mark: PhantomData,
};
}

/// this function calls a function directly.
/// the return value has auto type, user must cast the value before use.
///
/// # panic
pub fn call_direct(
&self,
id: FunctionID<'ctx>,
args: &[Value<'ctx, 'func, Auto<'ctx>>],
) -> Value<'ctx, 'func, Auto<'ctx>> {
let ty: &FunctionType<'ctx> =
unsafe { core::mem::transmute(self.builder.ctx.get_function_type(id)) };

if ty.params.len() != args.len() {
panic!("number of arguments not match")
}
for (i, t) in ty.params.iter().enumerate() {
if &args[i].ty.inner != t {
panic!("argument type not match")
}
}
```

```
let return_id = self.new_ssa(ty.return_.clone());

self.block.borrow_mut().inst.push(MIR::Call {
id: FunctionID {
id: id.id,
_mark: PhantomData,
},
args: args.iter().map(|v| v.id).collect(),
return_: return_id,
});

return Value {
id: return_id,
ty: Auto {
inner: ty.return_.clone(),
},
_mark: PhantomData,
};
}

/// allocate a smart pointer.
///
/// smart pointers must be stored in a stack slot.
/// Its SSA values should not be passed around.
/// Instead, load the smart pointer from stackslot every time value is accessed.
pub fn malloc<T: MarkerType<'ctx>>(
&self,
value: Value<'ctx, 'func, T>,
) -> Value<'ctx, 'func, Smart<T>> {
let id = self.new_ssa(value.ty.to_type());

self.block.borrow_mut().inst.push(MIR::Malloc(value.id, id));

return Value {
id,
ty: Smart { pointee: value.ty },
_mark: PhantomData,
};
}

/// frees an allocation
///
/// if GC is choosen, a safepoint is inserted.
```

```
/// if Arc is choosen, this function does nothing.
pub fn free<T: MarkerType<'ctx>>(&self, ptr: Value<'ctx, 'func, Smart<T>>){
self.block.borrow_mut().inst.push(MIR::Free(ptr.id));
}

pub fn await_<T: MarkerType<'ctx>>(
&self,
future: Value<'ctx, 'func, Future<T>>,
) -> Value<'ctx, 'func, T> {
if !self.builder.func.borrow().is_async{
panic!("await in non async function")
}
let id = self.new_ssa(future.ty.value.to_type());

self.block.borrow_mut().inst.push(MIR::AsyncAwait(future.id, id));

return Value {
id,
ty: future.ty.value,
_mark: PhantomData,
};
}

/// yields from a generator
///
/// all the previous SSA values would be invalidated after this point.
/// Any value wanting to live across generator boundaries must be stored in
stackslot
pub fn yield_<T: MarkerType<'ctx>>(&self, value: Value<'ctx, 'func, T>) ->
Value<'ctx, 'func, Auto<'ctx>>{
if let Some(desc) = &self.builder.func.borrow().is_generator{
let resume_ty = desc.resume_type.clone();

let id = self.new_ssa(resume_ty.clone());

self.block.borrow_mut().inst.push(MIR::Yield(value.id, id));

return Value {
id,
ty: Auto { inner: resume_ty },
_mark: PhantomData
}
} else{
panic!("yield in non generator function")
```

```
}
}

/// resume a generator
pub fn generator_resume<Y: MarkerType<'ctx>, RE: MarkerType<'ctx>, R:
MarkerType<'ctx>, >(&self, generator: Value<'ctx, 'func, Generator<Y, RE,
R>>, resume: Value<'ctx, 'func, RE>) -> Value<'ctx, 'func, Enum<'ctx, (Y,
R)>>{
let ty = Type::Enum(Box::new([generator.ty.yield_.to_type(),
generator.ty.return_.to_type()]));
let id = self.new_ssa(ty);

self.block.borrow_mut().inst.push(MIR::GeneratorNext(generator.id, resume.id,
id));

return Value {
id: id,
ty: Enum {
variants: (generator.ty.yield_, generator.ty.return_),
_mark: PhantomData
},
_mark: PhantomData
}
}
}
```

5.3.9 native-ts-mir/types/mod.rs

```
pub mod aggregate;

use std::marker::PhantomData;

use crate::util::{AggregateID, InterfaceID};
use crate::Value;

#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub enum ScalarType {
I8,
I16,
I32,
I64,
U8,
U16,
```

```
U32,
U64,
F32,
F64,
}

impl ScalarType {
pub fn is_signed(&self) -> bool {
match self {
Self::U8 | Self::U16 | Self::U32 | Self::U64 => false,
_ => true,
}
}

pub fn is_float(&self) -> bool {
match self {
Self::F32 | Self::F64 => true,
_ => false,
}
}

pub fn size(&self) -> usize {
match self {
Self::U8 | Self::I8 => 1,
Self::U16 | Self::I16 => 2,
Self::U32 | Self::I32 => 4,
Self::U64 | Self::I64 => 8,
Self::F32 => 4,
Self::F64 => 8,
}
}
}

#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub struct FunctionType<'ctx> {
pub params: Box<[Type<'ctx>]>,
pub return_: Type<'ctx>,
}

#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub enum Type<'ctx> {
/// void
Void,
/// i8
```

```
I8,
/// i16
I16,
/// i32
I32,
/// i64
I64,
/// isize
Isize,
/// u8
U8,
/// u16
U16,
/// u32
U32,
/// u64
U64,
/// usize
Usize,
/// f32
F32,
/// f64
F64,
/// an aggregate type
Aggregate(AggregateID<'ctx>),
/// an interface type
Interface(InterfaceID<'ctx>),
/// a raw pointer
Pointer(Box<Type<'ctx>>),
/// a smart pointer
SmartPointer(Box<Type<'ctx>>),
/// a function, a safe wrapper around pointer
Function(Box<FunctionType<'ctx>>),
/// a fixed size array
Array(Box<(Type<'ctx>, u64)>),

/// a future
Future(Box<Type<'ctx>>),
/// a generator, yield, resume, return
Generator(Box<(Type<'ctx>, Type<'ctx>, Type<'ctx>)>),

Enum(Box<[Type<'ctx>]>),

SIMDx2(ScalarType),
```

```
SIMDx4(ScalarType),
SIMDx8(ScalarType),
SIMDx16(ScalarType),
SIMDx32(ScalarType),
SIMDx64(ScalarType),
SIMDx128(ScalarType),
SIMDx256(ScalarType),
}

impl<'ctx> Type<'ctx> {
pub const BOOL: Self = Self::I8;

/// returns the number of lanes in SIMD types,
pub fn lanes(&self) -> usize {
match self {
Self::SIMDx2(_) => 2,
Self::SIMDx4(_) => 4,
Self::SIMDx8(_) => 8,
Self::SIMDx16(_) => 16,
Self::SIMDx32(_) => 32,
Self::SIMDx64(_) => 64,
Self::SIMDx128(_) => 128,
Self::SIMDx256(_) => 256,
_ => 0,
}
}
}

mod seal {
pub trait Sealed {}
}

pub trait MarkerType<'ctx>: seal::Sealed + Clone {
fn to_type(&self) -> Type<'ctx>;
}

pub trait IntoMarkerType<'ctx> {
type Marker: MarkerType<'ctx>;
fn into(self) -> Self::Marker;
}

impl<'ctx, T: MarkerType<'ctx>> IntoMarkerType<'ctx> for T {
type Marker = Self;
fn into(self) -> Self::Marker {
```

```
return self;
}
}
impl<'ctx> IntoMarkerType<'ctx> for Type<'ctx> {
type Marker = Auto<'ctx>;
fn into(self) -> Self::Marker {
Auto { inner: self }
}
}

pub trait FieldedMarkerType<'ctx>: MarkerType<'ctx> {}
pub trait PointerMarkerType<'ctx, T: MarkerType<'ctx>>: MarkerType<'ctx>{
fn pointee(&self) -> &T;
}

pub trait IntMarkerType: seal::Sealed + for<'a> MarkerType<'a> + Default {
type Type: IntoIntMarkerType;
}
pub trait IntoIntMarkerType: seal::Sealed {
type Marker: IntMarkerType;
type Int;
fn to_i128(self) -> i128;
}
//impl<I:IntMarkerType> IntoIntMarkerType for I{
// type Marker = I;
// type Int = I::Type;
//}

pub trait FloatMarkerType: seal::Sealed + for<'a> MarkerType<'a> + Default {
type Type;
}
pub trait IntoFloatMarkerType: seal::Sealed {
type Marker: FloatMarkerType;
type Float;
}
//impl<F:FloatMarkerType> IntoFloatMarkerType for F{
// type Marker = F;
// type Float = F::Type;
//}
pub trait ScalarMarkerType: seal::Sealed + for<'a> MarkerType<'a> + Default
+ Copy {
const TY: ScalarType;
}
pub trait IntoScalarMarkerType: seal::Sealed {
```

```
type Marker: ScalarMarkerType;
}
pub trait MathMarkerType: seal::Sealed + for<'a> MarkerType<'a> {}
pub trait IntMathMarkerType: MathMarkerType {}
pub trait FloatMathMarkerType: MathMarkerType {}

macro_rules! impl_int_marker {
($n:ident, $t:ty) => {
#[derive(Debug, Default, Clone, Copy, PartialEq, Eq)]
pub struct $n;
impl seal::Sealed for $n {}
impl<'ctx> MarkerType<'ctx> for $n {
fn to_type(&self) -> Type<'ctx> {
return Type::$n;
}
}
impl ScalarMarkerType for $n {
const TY: ScalarType = ScalarType::$n;
}
impl IntoScalarMarkerType for $t {
type Marker = $n;
}
impl MathMarkerType for $n {}
impl IntMathMarkerType for $n {}
impl IntMarkerType for $n {
type Type = $t;
}
impl seal::Sealed for $t {}
impl IntoIntMarkerType for $t {
type Marker = $n;
type Int = $t;
fn to_i128(self) -> i128 {
self as i128
}
}
};
}
macro_rules! impl_float_marker {
($n:ident, $t:ty) => {
#[derive(Debug, Default, Clone, Copy, PartialEq, Eq)]
pub struct $n;
impl seal::Sealed for $n {}
impl<'ctx> MarkerType<'ctx> for $n {
fn to_type(&self) -> Type<'ctx> {
```

```
return Type::$n;
}
}
impl ScalarMarkerType for $n {
const TY: ScalarType = ScalarType::$n;
}
impl IntoScalarMarkerType for $t {
type Marker = $n;
}
impl MathMarkerType for $n {}
impl FloatMathMarkerType for $n {}
impl FloatMarkerType for $n {
type Type = $t;
}
impl seal::Sealed for $t {}
impl IntoFloatMarkerType for $t {
type Marker = $n;
type Float = $t;
}
};
}
macro_rules! impl_non_scalar_marker {
($n:ident, $t:ty) => {
#[derive(Debug, Default, Clone, Copy, PartialEq, Eq)]
pub struct $n;
impl seal::Sealed for $n {}
impl<'ctx> MarkerType<'ctx> for $n {
fn to_type(&self) -> Type<'ctx> {
return Type::$n;
}
}
impl MathMarkerType for $n {}
impl IntMathMarkerType for $n {}
impl IntMarkerType for $n {
type Type = $t;
}
impl seal::Sealed for $t {}
impl IntoIntMarkerType for $t {
type Marker = $n;
type Int = $t;
fn to_i128(self) -> i128 {
self as i128
}
}
```

```
};
}

#[derive(Debug, Default, Clone, Copy, PartialEq, Eq)]
pub struct Void;

impl seal::Sealed for Void {}

impl<'ctx> MarkerType<'ctx> for Void {
fn to_type(&self) -> Type<'ctx> {
Type::Void
}
}

impl_int_marker!(I8, i8);
impl_int_marker!(I16, i16);
impl_int_marker!(I32, i32);
impl_int_marker!(I64, i64);
impl_int_marker!(U8, u8);
impl_int_marker!(U16, u16);
impl_int_marker!(U32, u32);
impl_int_marker!(U64, u64);

impl_non_scalar_marker!(Usize, usize);
impl_non_scalar_marker!(Isize, isize);

impl_float_marker!(F64, f64);
impl_float_marker!(F32, f32);

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Function<
'ctx,
Arg: FunctionArgs<'ctx> = AutoArgs<'ctx>,
R: MarkerType<'ctx> = Auto<'ctx>,
> {
pub args: Arg,
pub return_: R,
pub(crate) _mark: PhantomData<&'ctx ()>,
}

impl<'ctx, Arg: FunctionArgs<'ctx>, R: MarkerType<'ctx>> seal::Sealed for
Function<'ctx, Arg, R> {}

impl<'ctx, Arg: FunctionArgs<'ctx>, R: MarkerType<'ctx>> MarkerType<'ctx>
```

```
for Function<'ctx, Arg, R>
{
fn to_type(&self) -> Type<'ctx> {
let len = self.args.len();
let mut params = Vec::with_capacity(len);
for i in 0..len {
params.push(self.args.get(i))
}
Type::Function(Box::new(FunctionType {
params: params.into_boxed_slice(),
return_: self.return_.to_type(),
}))
}
}

pub trait FunctionArgs<'ctx>: Clone {
type ArgValues<'func>: ValueIndex<'ctx, 'func> + ?Sized;
fn len(&self) -> usize;
fn get(&self, index: usize) -> Type<'ctx>;
}

pub trait ValueIndex<'ctx, 'func> {
fn len(&self) -> usize;
fn get(&self, index: usize) -> Value<'ctx, 'func, Auto<'ctx>>;
}

#[derive(Clone)]
pub struct AutoArgs<'ctx>(pub(crate) Box<[Type<'ctx>]>);

impl<'ctx> FunctionArgs<'ctx> for AutoArgs<'ctx> {
type ArgValues<'func> = [Value<'ctx, 'func, Auto<'ctx>>];
fn len(&self) -> usize {
self.0.len()
}
fn get(&self, index: usize) -> Type<'ctx> {
self.0[index].clone()
}
}

impl<'ctx, 'func> ValueIndex<'ctx, 'func> for [Value<'ctx, 'func, Auto<'ctx>>]
{
fn len(&self) -> usize {
self.len()
}
```

```
fn get(&self, index: usize) -> Value<'ctx, 'func, Auto<'ctx>> {
self[index].into_auto()
}
}

impl<'ctx> FunctionArgs<'ctx> for () {
type ArgValues<'func> = ();
fn len(&self) -> usize {
0
}
fn get(&self, _index: usize) -> Type<'ctx> {
unreachable!()
}
}

impl<'ctx, 'func> ValueIndex<'ctx, 'func> for () {
fn len(&self) -> usize {
0
}
fn get(&self, _index: usize) -> Value<'ctx, 'func, Auto<'ctx>> {
unreachable!()
}
}

impl<'ctx, T: MarkerType<'ctx>> FunctionArgs<'ctx> for T {
type ArgValues<'func> = Value<'ctx, 'func, T>;
fn len(&self) -> usize {
1
}
fn get(&self, index: usize) -> Type<'ctx> {
match index {
0 => self.to_type(),
_ => unreachable!(),
}
}
}

impl<'ctx, 'func, T: MarkerType<'ctx>> ValueIndex<'ctx, 'func> for Value<'ctx,
'func, T> {
fn len(&self) -> usize {
1
}
fn get(&self, index: usize) -> Value<'ctx, 'func, Auto<'ctx>> {
if index == 0 {
```

```
self.into_auto()
} else {
unreachable!()
}
}
}

macro_rules! impl_function_arg {
($($ty:ident, $idx:tt),*) => {
#[allow(unused)]
impl<'ctx, $($ty:MarkerType<'ctx>),*> FunctionArgs<'ctx> for ($($ty),*){
type ArgValues<'func> = ($(Value<'ctx, 'func, $ty>),*);
fn len(&self) -> usize {
$($idx + 1);*
}
fn get(&self, index: usize) -> Type<'ctx> {
$(
if $idx == index{
return self.$idx.to_type()
}
)*
unreachable!()
}
}
#[allow(unused)]
impl<'ctx, 'func, $($ty:MarkerType<'ctx>),*> ValueIndex<'ctx, 'func> for ($
(Value<'ctx, 'func, $ty>),*){
fn len(&self) -> usize{
$($idx + 1);*
}
fn get(&self, index:usize) -> Value<'ctx, 'func, Auto<'ctx>>{
$(
if $idx == index{
return self.$idx.into_auto()
}
)*
unreachable!()
}
}
};
}

impl_function_arg!(A, 0, B, 1);
impl_function_arg!(A, 0, B, 1, C, 2);
```

```
impl_function_arg!(A, 0, B, 1, C, 2, D, 3);
impl_function_arg!(A, 0, B, 1, C, 2, D, 3, E, 4);
impl_function_arg!(A, 0, B, 1, C, 2, D, 3, E, 4, F, 5);
impl_function_arg!(A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6);
impl_function_arg!(A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7);
impl_function_arg!(A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8);
impl_function_arg!(A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9);
impl_function_arg!(A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10);
impl_function_arg!(A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L,
11);
impl_function_arg!(A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L,
11, M, 12);
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13
);
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13, O,
14
);
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13, O,
14,
P, 15
);
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13, O,
14,
P, 15, Q, 16
);
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13, O,
14,
P, 15, Q, 16, R, 17
);
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13, O,
14,
P, 15, Q, 16, R, 17, S, 18
);
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13, O,
14,
P, 15, Q, 16, R, 17, S, 18, T, 19
);
```

```
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13, O,
14,
P, 15, Q, 16, R, 17, S, 18, T, 19, U, 20
);
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13, O,
14,
P, 15, Q, 16, R, 17, S, 18, T, 19, U, 20, V, 21
);
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13, O,
14,
P, 15, Q, 16, R, 17, S, 18, T, 19, U, 20, V, 21, W, 22
);
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13, O,
14,
P, 15, Q, 16, R, 17, S, 18, T, 19, U, 20, V, 21, W, 22, X, 23
);
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13, O,
14,
P, 15, Q, 16, R, 17, S, 18, T, 19, U, 20, V, 21, W, 22, X, 23, Y, 24
);
impl_function_arg!(
A, 0, B, 1, C, 2, D, 3, E, 4, F, 5, G, 6, H, 7, I, 8, J, 9, K, 10, L, 11, M, 12, N, 13, O,
14,
P, 15, Q, 16, R, 17, S, 18, T, 19, U, 20, V, 21, W, 22, X, 23, Y, 24, Z, 25
);

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Aggregate<'ctx>(pub AggregateID<'ctx>);

impl<'ctx> seal::Sealed for Aggregate<'ctx> {}

impl<'ctx> MarkerType<'ctx> for Aggregate<'ctx> {
fn to_type(&self) -> Type<'ctx> {
Type::Aggregate(self.0)
}
}
impl<'ctx> FieldedMarkerType<'ctx> for Aggregate<'ctx> {}
impl<'ctx> FieldedMarkerType<'ctx> for Pointer<Aggregate<'ctx>> {}
impl<'ctx> FieldedMarkerType<'ctx> for Smart<Aggregate<'ctx>> {}
```

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Interface<'ctx>(pub InterfaceID<'ctx>);

impl<'ctx> seal::Sealed for Interface<'ctx> {}

impl<'ctx> MarkerType<'ctx> for Interface<'ctx> {
fn to_type(&self) -> Type<'ctx> {
Type::Interface(self.0)
}
}
impl<'ctx> FieldedMarkerType<'ctx> for Interface<'ctx> {}
impl<'ctx> FieldedMarkerType<'ctx> for Pointer<Interface<'ctx>> {}
impl<'ctx> FieldedMarkerType<'ctx> for Smart<Interface<'ctx>> {}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Array<T> {
pub element: T,
pub length: u64,
}

impl<'ctx, T: MarkerType<'ctx>> seal::Sealed for Array<T> {}

impl<'ctx, T: MarkerType<'ctx>> MarkerType<'ctx> for Array<T> {
fn to_type(&self) -> Type<'ctx> {
Type::Array(Box::new((self.element.to_type(), self.length)))
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Pointer<T> {
pub pointee: T,
}

impl<'ctx, T> seal::Sealed for Pointer<T> where T: MarkerType<'ctx> {}

impl<'ctx, T> MarkerType<'ctx> for Pointer<T>
where
T: MarkerType<'ctx>,
{
fn to_type(&self) -> Type<'ctx> {
Type::Pointer(Box::new(self.pointee.to_type()))
}
}
```

```
impl<'ctx, T: MarkerType<'ctx>> PointerMarkerType<'ctx, T> for Pointer<T>{
fn pointee(&self) -> &T {
&self.pointee
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Smart<T> {
pub pointee: T,
}

impl<'ctx, T> seal::Sealed for Smart<T> where T: MarkerType<'ctx> {}

impl<'ctx, T> MarkerType<'ctx> for Smart<T>
where
T: MarkerType<'ctx>,
{
fn to_type(&self) -> Type<'ctx> {
Type::SmartPointer(Box::new(self.pointee.to_type()))
}
}

impl<'ctx, T: MarkerType<'ctx>> PointerMarkerType<'ctx, T> for Smart<T>{
fn pointee(&self) -> &T {
&self.pointee
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Future<T> {
pub value: T,
}

impl<'ctx, T: MarkerType<'ctx>> seal::Sealed for Future<T> {}

impl<'ctx, T: MarkerType<'ctx>> MarkerType<'ctx> for Future<T> {
fn to_type(&self) -> Type<'ctx> {
Type::Future(Box::new(self.value.to_type()))
}
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Generator<Y, RE, R> {
pub yield_: Y,
```

```
pub resume: RE,
pub return_: R,
}

impl<'ctx, Y: MarkerType<'ctx>, RE: MarkerType<'ctx>, R: MarkerType<'ctx>>
seal::Sealed
for Generator<Y, RE, R>
{
}

impl<'ctx, Y: MarkerType<'ctx>, RE: MarkerType<'ctx>, R: MarkerType<'ctx>>
MarkerType<'ctx>
for Generator<Y, RE, R>
{
fn to_type(&self) -> Type<'ctx> {
Type::Generator(Box::new((
self.yield_.to_type(),
self.resume.to_type(),
self.return_.to_type(),
)))
}
}

pub(crate) mod simd {
pub struct LaneCount<const N: usize>;

pub trait SupportedLaneCount {}

impl SupportedLaneCount for LaneCount<2> {}
impl SupportedLaneCount for LaneCount<4> {}
impl SupportedLaneCount for LaneCount<8> {}
impl SupportedLaneCount for LaneCount<16> {}
impl SupportedLaneCount for LaneCount<32> {}
impl SupportedLaneCount for LaneCount<64> {}
impl SupportedLaneCount for LaneCount<128> {}
impl SupportedLaneCount for LaneCount<256> {}
}

#[derive(Debug, Default, Clone, Copy, PartialEq, Eq)]
pub struct SIMD<T: ScalarMarkerType, const N: usize>(PhantomData<T>)
where
simd::LaneCount<N>: simd::SupportedLaneCount;

impl<T: ScalarMarkerType, const N: usize> seal::Sealed for SIMD<T, N> where
```

```
simd::LaneCount<N>: simd::SupportedLaneCount
{
}

impl<'ctx, T: ScalarMarkerType, const N: usize> MarkerType<'ctx> for SIMD<T,
N>
where
simd::LaneCount<N>: simd::SupportedLaneCount,
{
fn to_type(&self) -> Type<'ctx> {
match N {
2 => Type::SIMDx2(T::TY),
4 => Type::SIMDx4(T::TY),
8 => Type::SIMDx8(T::TY),
16 => Type::SIMDx16(T::TY),
32 => Type::SIMDx32(T::TY),
64 => Type::SIMDx64(T::TY),
128 => Type::SIMDx128(T::TY),
256 => Type::SIMDx256(T::TY),
_ => unreachable!(),
}
}
}

impl<T: ScalarMarkerType, const N: usize> MathMarkerType for SIMD<T, N>
where
simd::LaneCount<N>: simd::SupportedLaneCount
{
}

macro_rules! impl_simd_int_math {
($t:ty) => {
impl<const N: usize> IntMathMarkerType for SIMD<$t, N> where
simd::LaneCount<N>: simd::SupportedLaneCount
{
}
};
}

impl_simd_int_math!(I8);
impl_simd_int_math!(I16);
impl_simd_int_math!(I32);
impl_simd_int_math!(I64);
impl_simd_int_math!(U8);
```

```
impl_simd_int_math!(U16);
impl_simd_int_math!(U32);
impl_simd_int_math!(U64);

impl<const N: usize> FloatMathMarkerType for SIMD<F64, N> where
simd::LaneCount<N>: simd::SupportedLaneCount
{
}
impl<const N: usize> FloatMathMarkerType for SIMD<F32, N> where
simd::LaneCount<N>: simd::SupportedLaneCount
{
}

#[derive(Clone)]
pub struct Enum<'ctx, V: FunctionArgs<'ctx> = AutoArgs<'ctx>>{
pub variants: V,
pub _mark: PhantomData<&'ctx ()>
}

impl<'ctx, V: FunctionArgs<'ctx>> seal::Sealed for Enum<'ctx, V>{}

impl<'ctx, V: FunctionArgs<'ctx>> MarkerType<'ctx> for Enum<'ctx, V>{
fn to_type(&self) -> Type<'ctx> {
let mut v = Vec::new();

for i in 0..self.variants.len(){
v.push(self.variants.get(i))
}
Type::Enum(v.into_boxed_slice())
}
}

#[derive(Clone)]
pub struct Auto<'ctx> {
pub(crate) inner: Type<'ctx>,
}

impl<'ctx> seal::Sealed for Auto<'ctx> {}
impl<'ctx> MarkerType<'ctx> for Auto<'ctx> {
fn to_type(&self) -> Type<'ctx> {
self.inner.clone()
}
}
```

5.3.10 native-ts-mir/types/aggregate.rs

```
use std::hash::{Hash, Hasher};

use crate::util::Ident;

use super::Type;

#[derive(Debug, Default, PartialEq, Eq, Hash)]
pub struct AggregateDesc<'ctx> {
pub fields: Vec<(Ident, Type<'ctx>)>,
pub(crate) hash: u64,
}

impl<'ctx> AggregateDesc<'ctx> {
pub const fn new() -> Self {
Self {
fields: Vec::new(),
hash: 0
}
}
/// return true if has field
pub fn has_field(&self, name: Ident) -> bool {
self.fields.iter().find(|f| f.0 == name).is_some()
}
/// find index of a field
pub fn find_index(&self, name: Ident) -> Option<usize> {
self.fields
.iter()
.enumerate()
.find(|(_, f)| f.0 == name)
.map(|(i, _)| i)
}
/// add a field to aggregate
pub fn with_field(mut self, name: Ident, ty: Type<'ctx>) -> Self {
let mut state = ahash::AHasher::default();
state.write_u64(self.hash);
name.hash(&mut state);
ty.hash(&mut state);

self.fields.push((name, ty));
self.hash = state.finish();

return self;
}
```

```
}

#[derive(Debug, Default, PartialEq, Eq)]
pub struct EnumDesc<'ctx> {
pub varients: Vec<(Ident, Option<AggregateDesc<'ctx>>)>,
}

impl<'ctx> EnumDesc<'ctx> {
pub const fn new() -> Self {
Self {
varients: Vec::new(),
}
}
/// return true if have varient
pub fn has_varient(&self, name: Ident) -> bool {
self.varients.iter().find(|f| f.0 == name).is_some()
}
/// find the index of a varient
pub fn find_index(&self, varient: Ident) -> Option<usize> {
self.varients
.iter()
.enumerate()
.find(|(_, f)| f.0 == varient)
.map(|(i, _)| i)
}
/// add a varient to enum
pub fn with_varient(mut self, name: Ident, aggregate:
Option<AggregateDesc<'ctx>>) -> Self {
self.varients.push((name, aggregate));
return self;
}
}

pub struct InterfaceDesc<'ctx> {
pub fields: Vec<(Ident, Type<'ctx>)>,
pub(crate) hash: u64,
}

impl<'ctx> InterfaceDesc<'ctx> {
pub const fn new() -> Self {
Self {
fields: Vec::new(),
hash: 0
}
```

```
}
/// return true if interface has field
pub fn has_field(&self, name: Ident) -> bool {
self.fields.iter().find(|f| f.0 == name).is_some()
}
/// return index of field
pub fn find_index(&self, name: Ident) -> Option<usize> {
self.fields
.iter()
.enumerate()
.find(|(_, f)| f.0 == name)
.map(|(i, _)| i)
}
/// add field to interface
pub fn with_field(mut self, name: Ident, ty: Type<'ctx>) -> Self {
// calculate hash
let mut state = ahash::AHasher::default();
state.write_u64(self.hash);
name.hash(&mut state);
ty.hash(&mut state);

// update hash
self.hash = state.finish();
// push field
self.fields.push((name, ty));
return self;
}
}
```

5.3.11 native-ts-mir/backend/mod.rs

```
//pub mod llvm;

pub trait Backend{
type Output;
fn compile(&mut self, context: &crate::Context) -> Result<Self::Output,
String>;
}

pub struct ObjectFile{

}
```

5.3.12 native-ts-mir/backend/llvm/mod.rs

```
use std::collections::HashMap;



pub struct LLVMBackend{
context: llvm_sys::LLVMContext,
}

impl super::Backend for LLVMBackend{
fn compile(&mut self, context: &crate::Context) -> Result<super::ObjectFile,
String> {
let module = llvm_sys::core::LLVMModuleCreateWithNameInContext("",
&self.context);
let mut builder = LLVMBuilder{
ctx: module.get_context(),
module,
builder: self.context.create_builder(),

functions: Vec::new(),
};

builder.compile(context)?;

return Ok(todo!())
}
}

struct LLVMBuilder<'ctx>{
ctx: inkwell::context::ContextRef<'ctx>,
module: inkwell::module::Module<'ctx>,
builder: inkwell::builder::Builder<'ctx>,

functions: Vec<inkwell::values::FunctionValue<'ctx>>
}

impl<'ctx> LLVMBuilder<'ctx>{
pub fn compile(&mut self, context: &crate::Context) -> Result<(), String>{
for (id, func) in context.functions.iter().enumerate(){
let return_ty = self.translate_type(&func.ty.return_);
let mut params = Vec::new();
for t in func.ty.params.iter(){
```

```
params.push(self.translate_type(t));
}

let fn_ty = self.translate_function_type(return_ty, &params);

let f = self.module.add_function(
func.name.as_ref().map(|n|
{n.as_str()}).unwrap_or(itoa::Buffer::new().format(id)),
fn_ty,
match func.linkage{
Some(crate::Linkage::Appending) =>
Some(inkwell::module::Linkage::Appending),
Some(crate::Linkage::AvailableExternally) =>
Some(inkwell::module::Linkage::AvailableExternally),
Some(crate::Linkage::Common) => Some(inkwell::module::Linkage::Common),
Some(crate::Linkage::DLLExport) =>
Some(inkwell::module::Linkage::DLLExport),
Some(crate::Linkage::DLLImport) =>
Some(inkwell::module::Linkage::DLLImport),
Some(crate::Linkage::ExternWeak) =>
Some(inkwell::module::Linkage::ExternalWeak),
Some(crate::Linkage::External) => Some(inkwell::module::Linkage::External),
Some(crate::Linkage::Internal) => Some(inkwell::module::Linkage::Internal),
Some(crate::Linkage::LinkOnce) =>
Some(inkwell::module::Linkage::LinkOnceAny),
Some(crate::Linkage::LinkOnceOdr) =>
Some(inkwell::module::Linkage::LinkOnceODR),
Some(crate::Linkage::Private) => Some(inkwell::module::Linkage::Private),
Some(crate::Linkage::Weak) => Some(inkwell::module::Linkage::WeakAny),
Some(crate::Linkage::WeakOdr) =>
Some(inkwell::module::Linkage::WeakODR),
None => None
}
);

self.functions.push(f);
};


return Ok(())
}

pub fn translate_type(&self, ty: &crate::Type) ->
inkwell::types::AnyTypeEnum<'ctx>{
match ty{
```

```
crate::Type::Aggregate(a) => {

},
crate::Type::Array(a) => {
self.translate_type(ty)
}
}
}

pub fn translate_function_type(&self, return_ty:
inkwell::types::AnyTypeEnum<'ctx>, params:
&[inkwell::types::AnyTypeEnum<'ctx>]) ->
inkwell::types::FunctionType<'ctx>{

}
}
```

5.4 runtime

5.4.1 native-ts-runtime/lib.rs

```
/*
* Copyright 2023 YC. Lam. All rights reserved.
*
* This Source Code Form is subject to the terms of the Mozilla Public
* License, v. 2.0. If a copy of the MPL was not distributed with this
* file, You can obtain one at http://mozilla.org/MPL/2.0/.
```

```
*/

#![no_std]

extern crate alloc;

mod global_allocator;
pub mod asynchronous;
mod event_loop;
mod exception;
pub mod gc;
pub mod runtime;
pub mod types;
pub mod std;
```

5.4.2 native-ts-runtime/global_allocator.rs

```
#[global_allocator]
static ALLOCATOR: LibcAllocator = LibcAllocator;

pub struct LibcAllocator;

unsafe impl alloc::alloc::GlobalAlloc for LibcAllocator {
unsafe fn alloc(&self, layout: core::alloc::Layout) -> *mut u8 {
let size = layout.size();
libc::malloc(size) as *mut u8
}

unsafe fn dealloc(&self, ptr: *mut u8, _layout: core::alloc::Layout) {
libc::free(ptr as _);
}

unsafe fn realloc(
&self,
ptr: *mut u8,
_layout: core::alloc::Layout,
new_size: usize,
) -> *mut u8 {
libc::realloc(ptr as _, new_size) as _
}
}
```

5.4.3 native-ts-runtime/exception.rs

```rust
use core::ffi::{c_int, c_void};
use core::ptr;
use core::sync::atomic::{AtomicPtr, AtomicUsize, Ordering};

use gimli::{constants, NativeEndian};
use gimli::{EndianSlice, Error, Pointer, Reader};
use unwinding::abi::{UnwindAction, UnwindContext, UnwindReasonCode};

use crate::types::Any;

/// LAM\0NATS
pub const NATIVE_TS_EXCEPTION_CLASS: u64 = u64::from_ne_bytes([
'L' as u8, 'A' as u8, 'M' as u8, '\0' as u8, 'N' as u8, 'A' as u8, 'T' as u8, 'S' as u8,
]);

#[derive(Debug)]
enum EHAction {
None,
Cleanup(usize),
Catch(usize),
}

pub struct NativeTsException {
pub header: unwinding::abi::UnwindException,
// whether it is being thrown
pub thrown: bool,
// handler count
pub handler_count: usize,
pub next_exception: *mut NativeTsException,
/// native-ts specific memory
pub exception_value: Any,
}

pub type StaticSlice = EndianSlice<'static, NativeEndian>;

pub unsafe fn get_unlimited_slice<'a>(start: *const u8) -> &'a [u8] {
// Create the largest possible slice for this address.
let start = start as usize;
let end = start.saturating_add(isize::MAX as _);
let len = end - start;
unsafe { core::slice::from_raw_parts(start as *const _, len) }
}
```

```
pub unsafe fn deref_pointer(ptr: Pointer) -> usize {
match ptr {
Pointer::Direct(x) => x as _,
Pointer::Indirect(x) => unsafe { *(x as *const _) },
}
}

fn parse_pointer_encoding(input: &mut StaticSlice) ->
gimli::Result<constants::DwEhPe> {
let eh_pe = input.read_u8()?;
let eh_pe = constants::DwEhPe(eh_pe);

if eh_pe.is_valid_encoding() {
Ok(eh_pe)
} else {
Err(gimli::Error::UnknownPointerEncoding)
}
}

fn parse_encoded_pointer(
encoding: constants::DwEhPe,
unwind_ctx: &UnwindContext<'_>,
input: &mut StaticSlice,
) -> gimli::Result<Pointer> {
if encoding == constants::DW_EH_PE_omit {
return Err(Error::CannotParseOmitPointerEncoding);
}

let base = match encoding.application() {
constants::DW_EH_PE_absptr => 0,
constants::DW_EH_PE_pcrel => input.slice().as_ptr() as u64,
constants::DW_EH_PE_textrel =>
unwinding::abi::_Unwind_GetTextRelBase(unwind_ctx) as u64,
constants::DW_EH_PE_datarel =>
unwinding::abi::_Unwind_GetDataRelBase(unwind_ctx) as u64,
constants::DW_EH_PE_funcrel =>
unwinding::abi::_Unwind_GetRegionStart(unwind_ctx) as u64,
constants::DW_EH_PE_aligned => return
Err(Error::UnsupportedPointerEncoding),
_ => unreachable!(),
};

let offset = match encoding.format() {
```

```
constants::DW_EH_PE_absptr =>
input.read_address(core::mem::size_of::<usize>() as _),
constants::DW_EH_PE_uleb128 => input.read_uleb128(),
constants::DW_EH_PE_udata2 => input.read_u16().map(u64::from),
constants::DW_EH_PE_udata4 => input.read_u32().map(u64::from),
constants::DW_EH_PE_udata8 => input.read_u64(),
constants::DW_EH_PE_sleb128 => input.read_sleb128().map(|a| a as u64),
constants::DW_EH_PE_sdata2 => input.read_i16().map(|a| a as u64),
constants::DW_EH_PE_sdata4 => input.read_i32().map(|a| a as u64),
constants::DW_EH_PE_sdata8 => input.read_i64().map(|a| a as u64),
_ => unreachable!(),
}?;

let address = base.wrapping_add(offset);
Ok(if encoding.is_indirect() {
Pointer::Indirect(address)
} else {
Pointer::Direct(address)
})
}

fn find_eh_action(
reader: &mut StaticSlice,
unwind_ctx: &UnwindContext<'_>,
) -> gimli::Result<EHAction> {
let func_start = unwinding::abi::_Unwind_GetRegionStart(unwind_ctx);
let mut ip_before_instr = 0;
let ip = unwinding::abi::_Unwind_GetIPInfo(unwind_ctx, &mut ip_before_instr);
let ip = if ip_before_instr != 0 { ip } else { ip - 1 };

let start_encoding = parse_pointer_encoding(reader)?;
let lpad_base = if !start_encoding.is_absent() {
unsafe { deref_pointer(parse_encoded_pointer(start_encoding, unwind_ctx,
reader)?) }
} else {
func_start
};

let ttype_encoding = parse_pointer_encoding(reader)?;
if !ttype_encoding.is_absent() {
reader.read_uleb128()?;
}

let call_site_encoding = parse_pointer_encoding(reader)?;
```

```
let call_site_table_length = reader.read_uleb128()?;
reader.truncate(call_site_table_length as _)?;

while !reader.is_empty() {
let cs_start = unsafe {
deref_pointer(parse_encoded_pointer(
call_site_encoding,
unwind_ctx,
reader,
)?)
};
let cs_len = unsafe {
deref_pointer(parse_encoded_pointer(
call_site_encoding,
unwind_ctx,
reader,
)?)
};
let cs_lpad = unsafe {
deref_pointer(parse_encoded_pointer(
call_site_encoding,
unwind_ctx,
reader,
)?)
};
let cs_action = reader.read_uleb128()?;
if ip < func_start + cs_start {
break;
}
if ip < func_start + cs_start + cs_len {
if cs_lpad == 0 {
return Ok(EHAction::None);
} else {
let lpad = lpad_base + cs_lpad;
return Ok(match cs_action {
0 => EHAction::Cleanup(lpad),
_ => EHAction::Catch(lpad),
});
}
}
}
Ok(EHAction::None)
}
```

```
#[no_mangle]
pub extern "C" fn __native_ts_eh_personality(
version: c_int,
actions: UnwindAction,
exception_class: u64,
exception: &mut NativeTsException,
context: &mut UnwindContext,
) -> UnwindReasonCode {
// version of the unwind library must be 1
if version != 1 {
return UnwindReasonCode::FATAL_PHASE1_ERROR;
}

// check the exception class
if exception_class != NATIVE_TS_EXCEPTION_CLASS {
// ignore foreign exception
return UnwindReasonCode::CONTINUE_UNWIND;
}

let lsda = unwinding::abi::_Unwind_GetLanguageSpecificData(&context);
if lsda.is_null() {
return UnwindReasonCode::CONTINUE_UNWIND;
}

let mut lsda = EndianSlice::new(
unsafe { get_unlimited_slice(lsda as *const u8) },
NativeEndian,
);
let eh_action = match find_eh_action(&mut lsda, &context) {
Ok(v) => v,
Err(_) => return UnwindReasonCode::FATAL_PHASE1_ERROR,
};

// in the search phase, phase 1
if actions.contains(UnwindAction::SEARCH_PHASE) {
match eh_action {
// no catch is found, continue unwind
EHAction::None | EHAction::Cleanup(_) => return
UnwindReasonCode::CONTINUE_UNWIND,
// a handler is found
EHAction::Catch(_) => return UnwindReasonCode::HANDLER_FOUND,
}
}
```

```
// should be in cleanup phase, phase 2
if !actions.contains(UnwindAction::CLEANUP_PHASE) {
return UnwindReasonCode::FATAL_PHASE2_ERROR;
}

// the catch clause is not allowed to catch the exception
// only procceed to cleanup and resume exception
if actions.contains(UnwindAction::FORCE_UNWIND) {}
// the handler frame
if actions.contains(UnwindAction::HANDLER_FRAME) {}

if actions.contains(UnwindAction::END_OF_STACK) {}

match eh_action {
// no action is required
EHAction::None => return UnwindReasonCode::CONTINUE_UNWIND,
// setup the context and transfer to landingpad
EHAction::Catch(landingpad) | EHAction::Cleanup(landingpad) => {
// set the ip to the landing pad
unwinding::abi::_Unwind_SetIP(context, landingpad);

#[cfg(target_arch = "x86_64")]
let regs = (gimli::X86_64::RAX, gimli::X86_64::RDX);
#[cfg(target_arch = "x86")]
let regs = (gimli::X86::EAX, gimli::X86::EDX);
#[cfg(any(target_arch = "riscv64", target_arch = "riscv32"))]
let regs = (gimli::RiscV::A0, gimli::RiscV::A1);
#[cfg(target_arch = "aarch64")]
let regs = (gimli::AArch64::X0, gimli::AArch64::X1);
#[cfg(not(any(
target_arch = "x86_64",
target_arch = "x86",
target_arch = "riscv64",
target_arch = "riscv32",
target_arch = "aarch64"
)))]
compile_error!("unsupported target");

// forward the exception
unwinding::abi::_Unwind_SetGR(context, regs.0 .0 as _, exception as *mut _ as
usize);
unwinding::abi::_Unwind_SetGR(context, regs.1 .0 as _, 0);

return UnwindReasonCode::INSTALL_CONTEXT;
```

```
}
}
}

#[no_mangle]
pub extern "C" fn __native_ts_allocate_exception(value: Any) -> *mut
NativeTsException {
unsafe {
let exception =
libc::malloc(core::mem::size_of::<NativeTsException>()) as *mut
NativeTsException;
ptr::write(
exception,
NativeTsException {
header: core::mem::zeroed(),
thrown: false,
handler_count: 0,
next_exception: 0 as _,
exception_value: value,
},
);
return exception;
}
}
#[no_mangle]
pub extern "C" fn __native_ts_free_exception(exception: &mut
NativeTsException) {
unsafe {
libc::free(exception as *mut _ as *mut c_void);
}
}

static UNCAUGHT_EXCEPTION: AtomicUsize = AtomicUsize::new(0);
static STACK_TOP: AtomicPtr<NativeTsException> = AtomicPtr::new(0 as _);

#[no_mangle]
pub extern "C" fn __native_ts_throw(exception: &mut NativeTsException) {
UNCAUGHT_EXCEPTION.fetch_add(1, Ordering::SeqCst);
exception.header.exception_class = NATIVE_TS_EXCEPTION_CLASS;
exception.thrown = true;

unsafe {
unwinding::abi::_Unwind_RaiseException(&mut exception.header);
}
```

```
}

#[no_mangle]
pub extern "C" fn __native_ts_begin_catch(exception: &mut NativeTsException)
{
exception.handler_count += 1;
UNCAUGHT_EXCEPTION.fetch_sub(1, Ordering::SeqCst);

let current_top = STACK_TOP.load(Ordering::SeqCst);
exception.next_exception = current_top;

STACK_TOP.store(exception, Ordering::SeqCst);
}

#[no_mangle]
pub extern "C" fn __native_ts_end_catch() {
let top = STACK_TOP.load(Ordering::SeqCst);

if let Some(exception) = unsafe { top.as_mut() } {
// decrement handler count
let _ = exception.handler_count.checked_sub(1);

if exception.handler_count == 0 {
// pop from stack
STACK_TOP.store(exception.next_exception, Ordering::SeqCst);
}
if exception.handler_count == 0 && !exception.thrown {
// destroy exception
}
}
}

#[no_mangle]
pub extern "C" fn __native_ts_rethrow() {
let top = STACK_TOP.load(Ordering::SeqCst);

if let Some(exception) = unsafe { top.as_mut() } {
exception.thrown = true;
}
}

#[no_mangle]
pub extern "C" fn __native_ts_unwind_resume(exception: &mut
NativeTsException) {
```

```
unsafe { unwinding::abi::_Unwind_Resume(&mut exception.header) }
}
```

5.4.4 native-ts-runtime/asynchronous/mod.rs

```
use core::ptr::NonNull;

pub mod executor;
pub mod file;
pub mod task;

pub static GLOBAL_EXECUTOR: executor::Executor =
executor::Executor::new();

#[no_mangle]
pub extern "C" fn __native_ts_async_task_create(
poll_func: extern "C" fn(&mut task::AsyncContext, *mut u8, *mut u8) ->
task::AsyncTaskPoll,
task_size: usize,
result_size: usize,
) -> task::AsyncTask {
let state = unsafe{crate::gc::allocate_raw(task_size)};
let re = unsafe{crate::gc::allocate_raw(result_size)};

return task::AsyncTask {
poll: poll_func,
state: state,
ready: false,
result: re,
};
}

#[no_mangle]
pub extern "C" fn __native_ts_async_spawn(task: task::AsyncTask) ->
executor::TaskHandle<'static>{
GLOBAL_EXECUTOR.spawn(task)
}

#[no_mangle]
pub extern "C" fn __native_ts_async_await(task: executor::TaskHandle) ->
Option<NonNull<u8>>{
match GLOBAL_EXECUTOR.poll_task(task){
Some(p) => Some(NonNull::new(p as *const _ as *mut _)?),
None => None,
```

```
}
}
```

5.4.5 native-ts-runtime/asynchronous/task.rs

```
use core::future::Future;
use core::pin::Pin;
use core::ptr::NonNull;
use core::task::Poll;

use crate::gc;

#[repr(C)]
pub struct AsyncContext<'a> {
raw_context: core::task::Context<'a>,
}

#[repr(u8)]
pub enum AsyncTaskPoll {
Pending = 0,
Ready = 1,
}

#[repr(C)]
#[derive(Clone, Copy)]
pub struct AsyncTask<T = u8> {
/// the poll function
pub(crate) poll:
extern "C" fn(ctx: &mut AsyncContext, state: *mut u8, re: *mut u8) ->
AsyncTaskPoll,
/// state of function
pub(crate) state: NonNull<u8>,
/// if is ready
pub(crate) ready: bool,
/// the result store
pub(crate) result: NonNull<T>,
}

impl AsyncTask {
pub fn from_future<F, T>(future: F) -> AsyncTask
where
F: Future<Output = T> + Unpin + 'static,
{
extern "C" fn poll_future<T, F>(
```

```
ctx: &mut AsyncContext,
state: *mut u8,
re: *mut u8,
) -> AsyncTaskPoll
where
F: Future<Output = T>,
{
unsafe {
let f = &mut *(state as *mut F);
let future = Pin::new_unchecked(f);

match future.poll(&mut ctx.raw_context) {
Poll::Pending => return AsyncTaskPoll::Pending,
Poll::Ready(v) => {
(re as *mut T).write(v);
return AsyncTaskPoll::Ready;
}
};
}
}

unsafe {
let re = gc::allocate::<T>(core::mem::zeroed());
let state = gc::allocate::<F>(future);
return AsyncTask {
poll: poll_future::<T, F>,
state: state.cast(),
ready: false,
result: re.cast(),
};
}
}
}
```

5.4.6 native-ts-runtime/asynchronous/executor.rs

```
use core::marker::PhantomData;

use alloc::vec::Vec;
use lazy_thread_local::ThreadLocal;

use super::task::AsyncTask;

#[repr(C)]
```

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct TaskHandle<'a, T = u8>(usize, PhantomData<&'a T>);

pub struct Executor {
tasks: ThreadLocal<Vec<AsyncTask>>,
}

unsafe impl Sync for Executor {}

impl Executor {
pub const fn new() -> Self {
Self {
tasks: ThreadLocal::const_new(Vec::new()),
}
}

pub fn spawn<T>(&self, task: AsyncTask<T>) -> TaskHandle<T>{
let mut task: AsyncTask = unsafe{core::mem::transmute(task)};

self.run_once(&mut task);
let tasks = self.tasks.get_mut();
let id = tasks.len();
tasks.push(task);

TaskHandle(id, PhantomData)
}

pub fn wait_all(&self) {
loop {
let tasks = self.tasks.get_mut();
let mut all_done = true;

for task in tasks {
all_done &= self.run_once(task);
}

if all_done {
break;
}
}
}

/// poll the task once and return reference to result
```

```
pub fn poll_task<'a, T>(&'a self, handle: TaskHandle<'a, T>) -> Option<&'a
T>{
let tasks = self.tasks.get_mut();
if let Some(task) = tasks.get_mut(handle.0){
if self.run_once(task){
return Some(unsafe{task.result.cast().as_ref()})
}
}
return None
}

fn run_once(&self, task: &mut AsyncTask) -> bool {
if task.ready {
return true;
}

return false;
}
}
```

5.5 garbage collector

5.5.1 native-ts-gc/lib.rs

```
extern crate alloc;

mod heap;
mod cell;
mod thread;

use std::sync::atomic::{AtomicUsize, Ordering, AtomicBool};

use cell::{Flags, CellHeader};
use crossbeam_channel::{Sender, Receiver};
use heap::Heap;

const ALLOCATION_PER_GC: usize = 4096 * 4;

pub struct Allocator{
allocations: AtomicUsize,

gc_running: AtomicBool,
gc_marking_phase: AtomicBool,
gc_grey_scanning_phase: AtomicBool,
```

```
gc_sweeping_phase: AtomicBool,

threads_at_safe_point: AtomicUsize,

num_threads: AtomicUsize,
threads: [Option<()>;32],
gc_thread_handle: Option<std::thread::JoinHandle<()>>,

heap16: Heap<16>,
heap32: Heap<32>,
heap64: Heap<64>,
heap128: Heap<128>,
heap256: Heap<256>,
heap512: Heap<512>,
}

impl Allocator{
pub fn is_gc_marking_phase(&self) -> bool{
false
}
pub fn is_grey_scanning_phase(&self) -> bool{
false
}
}

static mut ALLOCATOR: Allocator = Allocator{
allocations: AtomicUsize::new(0),

gc_running: AtomicBool::new(false),
gc_marking_phase: AtomicBool::new(false),
gc_grey_scanning_phase: AtomicBool::new(false),
gc_sweeping_phase: AtomicBool::new(false),

num_threads: AtomicUsize::new(0),
threads: [None; 32],
threads_at_safe_point: AtomicUsize::new(0),
gc_thread_handle: None,

heap16: Heap::new(),
heap32: Heap::new(),
heap64: Heap::new(),
heap128: Heap::new(),
heap256: Heap::new(),
```

```
heap512: Heap::new()
};

lazy_static::lazy_static!{
static ref WORK:(Sender<Box<dyn Fn() + Send>>, Receiver<Box<dyn Fn() +
Send>>) = crossbeam_channel::unbounded();
}

pub fn safe_point(){
unsafe{
if ALLOCATOR.gc_grey_scanning_phase.load(Ordering::SeqCst) {
ALLOCATOR.threads_at_safe_point.fetch_add(1, Ordering::SeqCst);
while ALLOCATOR.is_grey_scanning_phase(){
if let Ok(work) = WORK.1.try_recv(){
// do the work
work();
}
}
ALLOCATOR.threads_at_safe_point.fetch_sub(1, Ordering::SeqCst);
}
}
}

pub fn write_barrier(ptr: &mut cell::Cell, slot: &mut *mut cell::Cell, value: &mut
cell::Cell){
unsafe{
if ALLOCATOR.gc_running.load(Ordering::Relaxed){
ptr.header.flags.set_grey();
value.header.flags.set_grey();
*slot = value;
// a possible safe point
safe_point();
} else{
*slot = value;
}
}
}

pub fn allocate(size: usize) -> &'static mut cell::Cell{
// a possible safe point
safe_point();

unsafe{
```

```
if ALLOCATOR.allocations.fetch_add(1, Ordering::Relaxed) ==
ALLOCATION_PER_GC{
ALLOCATOR.allocations.store(0, Ordering::SeqCst);
garbage_collect();
}
match size + core::mem::size_of::<CellHeader>(){
16 => ALLOCATOR.heap16.allocate(),
32 => ALLOCATOR.heap32.allocate(),
64 => ALLOCATOR.heap64.allocate(),
128 => ALLOCATOR.heap128.allocate(),
256 => ALLOCATOR.heap256.allocate(),
512 => ALLOCATOR.heap512.allocate(),
_ => {
todo!()
}
}
}
}

pub fn shade(cell: &mut cell::Cell){
cell.header.flags.set_grey();
}

pub unsafe fn garbage_collect(){
// set gc running to true
if ALLOCATOR.gc_running.swap(true, Ordering::SeqCst){
// gc already running
return;
}

static GC_THREAD_LAUNCHED: AtomicBool = AtomicBool::new(false);

if !GC_THREAD_LAUNCHED.swap(true, Ordering::SeqCst){
let handle = std::thread::spawn(||unsafe{
loop{
if ALLOCATOR.gc_running.load(Ordering::SeqCst){
// start marking phase
ALLOCATOR.gc_marking_phase.store(true, Ordering::SeqCst);

// mark roots

// finish marking phase
ALLOCATOR.gc_marking_phase.store(false, Ordering::SeqCst);
```

```
// grey scanning
ALLOCATOR.gc_grey_scanning_phase.store(true, Ordering::SeqCst);

// wait for threads to reach safe point
while ALLOCATOR.threads_at_safe_point.load(Ordering::Relaxed) <
ALLOCATOR.num_threads.load(Ordering::Relaxed){}

// actual grey scanning
ALLOCATOR.heap16.grey_scan();
ALLOCATOR.heap32.grey_scan();
ALLOCATOR.heap64.grey_scan();
ALLOCATOR.heap128.grey_scan();
ALLOCATOR.heap256.grey_scan();
ALLOCATOR.heap512.grey_scan();

// finish scanning
ALLOCATOR.gc_grey_scanning_phase.store(false, Ordering::SeqCst);

// sweeping
ALLOCATOR.gc_sweeping_phase.store(true, Ordering::SeqCst);

ALLOCATOR.heap16.sweep();
ALLOCATOR.heap32.sweep();
ALLOCATOR.heap64.sweep();
ALLOCATOR.heap128.sweep();
ALLOCATOR.heap256.sweep();
ALLOCATOR.heap512.sweep();

// finish sweeping
ALLOCATOR.gc_sweeping_phase.store(false, Ordering::SeqCst);

// finish gc
ALLOCATOR.gc_running.store(false, Ordering::SeqCst);
}
std::thread::park();
};
});
ALLOCATOR.gc_thread_handle = Some(handle);
}

// unpark the gc thread
ALLOCATOR.gc_thread_handle.as_ref().unwrap().thread().unpark();
// return and continue execution
```

```
return
}
```

5.5.2 native-ts-gc/thread.rs

```
use std::marker::PhantomData;

use alloc::boxed::Box;

#[cfg(unix)]
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord)]
pub struct ThreadID(libc::pthread_t);

pub struct JoinHandle<T>{
thread: Thread,
_mark: PhantomData<T>
}

impl<T> JoinHandle<T>{
pub fn thread(&self) -> &Thread{
return &self.thread
}
}

pub struct Thread{
id: ThreadID
}

impl Thread{
pub fn unpark(&self){
}
}

#[cfg(unix)]
pub fn spawn<F: Fn() -> T + Send + 'static, T>(f: F) -> JoinHandle<T>{
extern "C" fn pthread_wrapper<T, F: Fn() -> T + Send + 'static>(value: *mut
libc::c_void) -> *mut libc::c_void{
unsafe{
let value = (value as *mut F).as_mut().unwrap_unchecked();
let re = (value)();
drop(Box::from_raw(value));
return Box::leak(Box::new(re)) as *mut T as *mut libc::c_void
```

```
}
}

unsafe{
let mut id: libc::pthread_t = 0;
let mut attr: libc::pthread_attr_t = core::mem::zeroed();
let wrapped = Box::leak(Box::new(f));
let r = libc::pthread_create(&mut id, &mut attr, pthread_wrapper::<T, F>,
wrapped as *mut F as *mut libc::c_void);

return JoinHandle {
thread: Thread { id: ThreadID(id) },
_mark: PhantomData
}
}
}
```

5.5.3 native-ts-gc/heap.rs

```
use std::sync::atomic::{
Ordering,
AtomicUsize,
AtomicBool
};

use crate::cell::{Cell, Flags};

pub struct Block<const N:usize>{
cursor: AtomicUsize,
data: *mut [u8;4096 * 4]
}

pub struct Heap<const N:usize>{
cursor: AtomicUsize,
growing: AtomicBool,
blocks: Vec<Block<N>>,
}

unsafe impl<const N:usize> Sync for Heap<N>{}
unsafe impl<const N:usize> Send for Heap<N>{}

impl<const N:usize> Heap<N>{
```

```
pub const fn new() -> Self{
Self{
cursor: AtomicUsize::new(0),
growing: AtomicBool::new(false),
blocks: Vec::new(),
}
}
pub fn allocate(&mut self) -> &'static mut Cell{
if self.blocks.len() == 0{
self.grow();
};
// load cursor
let mut cursor = self.cursor.load(Ordering::Relaxed);
loop{
// allocate from block
if let Some(cell) = self.blocks[cursor].allocate(){
return cell
}
// cursor reaches end
if cursor +1 >= self.blocks.len(){
self.grow();
}

// cursor is updated
if let Err(updated) = self.cursor.compare_exchange_weak(cursor, cursor + 1,
Ordering::SeqCst, Ordering::Relaxed){
cursor = updated;
} else{
cursor += 1;
}
}
}
pub fn grow(&mut self){
// heap is already growing
if self.growing.swap(true, Ordering::SeqCst){
while self.growing.load(Ordering::Relaxed){};
return;
}

// allocate new block
self.blocks.push(
Block {
cursor: AtomicUsize::new(0),
data: unsafe{
```

```
alloc::alloc::alloc_zeroed(
alloc::alloc::Layout::new::<[u8;4096 * 4]>()
) as *mut [u8;4096 * 4]
}
}
);

// finish growing
self.growing.store(false, Ordering::SeqCst);
}

pub fn grey_scan(&self){
for b in &self.blocks{
b.grey_scan();
}
}

pub fn sweep(&self){
for b in &self.blocks{
b.sweep();
}
// reset cursor
self.cursor.store(0, Ordering::SeqCst);
}
}

impl<const N:usize> Block<N>{
pub fn allocate(&self) -> Option<&'static mut Cell>{
let mut cursor = self.cursor.load(Ordering::Relaxed);

loop{
if cursor >= 4096 * 4{
return None
}

unsafe{
let ptr = (self.data as *mut u8).add(cursor);
let cell = (ptr as *mut Cell).as_mut().unwrap_unchecked();

// not allocated
if !cell.header.flags.swap_allocated(true){
// set grey
cell.header.flags.set_grey();
// return cell
```

```
return Some(cell)
}
}
if let Err(updated) = self.cursor.compare_exchange_weak(cursor, cursor + N,
Ordering::SeqCst, Ordering::Relaxed){
cursor = updated;
} else{
cursor += N;
}
}
}

pub fn grey_scan(&self){
let ptr = self.data as *mut u8;

for i in 0..(4096 * 4)/N{
unsafe{
let cell = (ptr.add(N * i) as *mut Cell).as_mut().unwrap_unchecked();
// is grey
if cell.header.flags.is_grey(){
cell.trace();
}
}
}
}

pub fn sweep(&self){
let ptr = self.data as *mut u8;

for i in 0..(4096 * 4)/N{
unsafe{
let cell = (ptr.add(N * i) as *mut Cell).as_mut().unwrap_unchecked();
// is white and allocated
if cell.header.flags.set_white() && cell.header.flags.is_allocated(){
if let Some(dtor) = cell.header.dtor{
// remove destructor
cell.header.dtor = None;
// call destructor
dtor(cell.payload.as_mut_ptr());
}
// deallocate
cell.header.flags.clear();
}
}
```

```
}
// reset cursor
self.cursor.store(0, Ordering::SeqCst);
}
}
```

5.5.4 native-ts-gc/cell.rs

```
use std::sync::atomic::{AtomicU8, Ordering};


#[repr(C)]
#[derive(Debug)]
pub struct Flags(AtomicU8);

impl Flags{
pub const EMPTY: u8 = 0;
pub const GREY: u8 = 0b00000001;
pub const BLACK: u8 = 0b00000010;
pub const ALLOCATED: u8 = 0b00000100;

pub fn clear(&self){
self.0.store(0, Ordering::SeqCst);
}
pub fn is_white(&self) -> bool{
self.0.load(Ordering::SeqCst) & (Self::GREY | Self::BLACK) == Self::EMPTY
}
pub fn set_white(&self) -> bool{
let f = self.0.fetch_and(!(Self::BLACK | Self::GREY), Ordering::SeqCst);
return f & (Self::GREY | Self::BLACK) == Self::EMPTY
}
pub fn is_grey(&self) -> bool{
self.0.load(Ordering::SeqCst) & Self::GREY == Self::GREY
}
pub fn set_grey(&self){
self.0.fetch_or(Self::GREY, Ordering::SeqCst);
}
pub fn is_black(&self) -> bool{
self.0.load(Ordering::SeqCst) & Self::BLACK == Self::BLACK
}
pub fn swap_black(&self, b: bool) -> bool{
```

```
if b{
let f = self.0.fetch_or(Self::BLACK, Ordering::SeqCst);
return f & Self::BLACK == Self::BLACK
} else{
let f = self.0.fetch_and(!Self::BLACK, Ordering::SeqCst);
return f & Self::BLACK == Self::BLACK
}
}
pub fn is_allocated(&self) -> bool{
self.0.load(Ordering::SeqCst) & Self::ALLOCATED == Self::ALLOCATED
}
pub fn swap_allocated(&self, b: bool) -> bool{
if b{
let f = self.0.fetch_or(Self::ALLOCATED, Ordering::SeqCst);
return f & Self::ALLOCATED == Self::ALLOCATED
} else{
let f = self.0.fetch_and(!Self::ALLOCATED, Ordering::SeqCst);
return f & Self::ALLOCATED == Self::ALLOCATED
}
}
}

#[repr(C)]
pub struct CellHeader{
pub flags: Flags,
pub dtor: Option<extern "C" fn(*mut u8)>,
pub trace: Option<extern "C" fn(*mut u8)>
}

#[repr(C)]
pub struct Cell{
pub(crate) header: CellHeader,
pub(crate) payload: [u8;0],
}

impl Cell{
pub fn trace(&self){
// set colour to black
if self.header.flags.swap_black(true){
// already traced
return;
}
if let Some(t) = &self.header.trace{
(t)(self.payload.as_ptr() as *mut u8)
```

```
}
}
}
```