Software Engineering for Compilers

Josef Svenningsson

Compiler Construction, Spring 2014

Testing compilers

Trusting the compiler

Bugs

When finding a bug, go to great lengths to find it in our own code.

- Most programmers trust the compiler to generate correct code
- The most important task of the compiler is to generate correct code

Establishing Compiler Correctness

Alternatives

- Proving the correctness of a compiler is prohibitively expensive
  (however, see the CompCert project)
- Testing is the only viable option

Testing compilers

- Most compilers use *unit testing*
- They have a big collection of example programs which are used for testing the compiler
- For each program the expected output is stored in the test suite
- Whenever a new bug is found, a new example program is added to the test suite. This is known as *regression testing*.

Random Testing For Compilers

- Testing compilers using random testing means generating programs in the source language.
- Writing good random test generators for a language is very difficult
- Different parts of the compiler might need different generators:
  - The parser needs random strings, but they need to be skewed towards syntactically correct programs in order to be useful.
  - The type checker needs a generator which can generate type correct programs (with high probablity)
- It can be hard to know what the correct execution of a program is.
  We need another compiler or interpreter to test againts.
- What if the generated program doesn't terminate, or takes a very long time?
- Using random testing for compilers is *a lot* of work.

Random testing

Random testing

- Generating random inputs and check correctness of output
- Used by e.g. QuickCheck

Project

Remember to test your compiler!

- Use the provided test suite!
- Write your own tests!

Compiler Bootstrapping

A self-hosting compiler

If you're designed an awesome programming language you would probably want to program in it.

In particular, you would want to write the compiler in this language.

A real language

Some people say:

A programming language isn't real until it has a self-hosting compiler

The chicken and egg problem

If we want to write a compiler for the language X in the language X, how does the first compiler get written?

Solutions

- Write an interpreter for language X in language Y.
- Write another compiler for language X in language Y.
- Write the compiler in a subset of X which is possible to compiler with an existing compiler.
- Hand-compile the first compiler.

Porting to new architectures

A related problem

How to port a compiler to a new hardware architecture.

Solution: Cross-compilation

Let the compiler emit code for the new architecture while still running on an old architecture.

Writing Makefiles

Make

The utility `make` is very handy for compiling large projects

It can help to track which files have been edited and recompile object files and programs where necessary.

Rules

A Makefile consists of rules which specifies:

- Which target file will be generated
- How these files are generated.

General structure of rules

```
target: dependencies .....
    shell commands specifying how to generate target
```

Concrete example

```
module.o : module.c
    gcc -c module.c -o module.o
```

Caveat

- The space before the shell commands needs to be *a tab stop*!
- If you just use spaces then the commands will not execute.

Using `make`

- Invoking `make` without any arguments will make the first target in the Makefile.
- When giving `make` a target as an argument it will try to build that target and any of its dependencies if needed.

Pattern rules

- When having lots of targets it can be inconvenient to list all of them in the in a Makefile.
- Then pattern rules come in handy

```
%.o : %.c
    gcc -c $< -o $@
```

PHONY rules

- Sometimes it is convenient to have targets which do not produce files.
- A common example is `clean` which removes all generated files.
- These targets should be declared as `PHONY`

```
.PHONY clean

clean:
  rm -f *.o
```

Outlook

- There is a lot more the `make`, but these basic principles will get you very far.
- `make` is not without flaws. But it is very widely available and good to know.

Project

- In the project you automatically get a Makefile from the BNFC tool.
- Don't forget to `make clean` before packaging your solution for submission
- It can be very convenient to have a target which automatically makes a package for submission

Managing state in the compiler

OO vs functional implementation language

- When writing the type checker and code generator, the compiler needs to carry around *symbol tables* with information about e.g. the type of a variable.
- This is handled differently when implementing the compiler in an OO language or a functional language.

OO

In OO languages it is easy to manage state, simply by using a local variable which is updated, or an object field.

Functional

In functional languages it can be tiresome to carry around state. Can be made much more convenient by using a *state monad*.

The state monad

The state monad provides a convenient way to carrying around state in Haskell

```
data CompileState = ....

data CompileMonad a = CM (State CompileState a)
```

State transformer

For debugging purposes it is often convenient to use the state monad *transformer* on top of the *IO monad*.

This allows for easily printing debug-information.

```
data CompileState = ....

data CompileMonad a = CM (StateT CompileState IO a)
```

State monad demo

Live coding

The lens package

The package `lens` provides functions which makes it more convenient to use the state monad.

Suppose we wish to use the following state in our state monad

```
data FState =
  FState { _consts  :: [Int]
         , _subst   :: [(V,V)]
         , _nameGen :: Int
         }

makeLenses ''FState
```

This produces *lenses* named `const`, `subst` and `nameGen`.

Note the underscores in the names!

Requires language extension `TemplateHaskell`.

State monad and lenses: Getting

Getting a field in the state

Without lenses

```
st <- get
let c = const st
```

With lenses

```
c <- using const
```

State monad and lenses: Setting

Setting a field in the state

Without lenses

```
set (st {const = []})
```

With lenses

```
const .= []
```

State monad and lenses: Updating

Updating a field in the state

Without lenses

```
set (st {const = i : const st})
```

With lenses

```
const %= (i:)
```

State monad and lenses

- The `lens` library is a *huge* library with lots of convenient functionality.
- We have only scratched the surface here.
- It is not mandatory to use either the state monad or the `lens` library in the project
- Use the tools you feel are helpful