

Incremental Haskell Builds with Nix

Using old build results as a starting point to speed up new builds.

Together with [Harry Garrood](#) I have been looking into speeding up Haskell builds with Nix. He recently wrote a [blog post](#) about changes in GHC 9.4 that made this task a little bit easier.

With this patch, source file modification times play no part in recompilation checking.

So what is happening instead?

... the hash of the contents of the source file [will] be stored within the corresponding `.hi` file and GHC [will] determine whether the source file had been changed by comparing this [old] hash to the source file's current hash ...

But how would we make use of that with Nix?

The Nix Wrapper

Haskell packages, that are built with `cabal`, can easily be turned into a derivation.

Regular Haskell package in a Nix Flake output

```
{
  packages.x86_64-linux.default =
    with import nixpkgs { system = "x86_64-linux"; };
    haskellPackages.callCabal2nix "example-package" ./.. {};
  # ...
}
```

All the hard work is being done by `cabal2nix`. Now we will add a wrapper around `packages.x86_64-linux.default` to make use of GHCs cleverness.

Wrapped Haskell package to allow incremental builds

```
{
  # ...
  packages.x86_64-linux.incremental =
    with import nixpkgs { system = "x86_64-linux"; };
    with import ./nix/haskell/lib.nix {
      lib = pkgs.lib;
      haskellLib = pkgs.haskell.lib;
    };
    buildIncrementally {
```

```

    regularPackage = self.packages.x86_64-linux.default;
    previousIncrement = incremental.packages.x86_64-linux.incremental
.incrementalBase;
  };
  # ...
}

```

What is of interest here?

buildIncrementally

A wrapper function, that turns a regular derivation into an incremental one.

regularPackage

The derivation, that we want to speed up.

previousIncrement

The derivation, that you previously built using **buildIncrementally**. You can set this to **null** or remove the attribute, if you don't have any previous result yet.

Implementation of **buildIncrementally**

We add an additional output "**incremental**" to the regular package , so that it becomes a multi-output derivation. This new output contains a tarball with basically all the files generated by GHC (**.o**, **.hi**, **.dyn_o**, **.dyn_hi**, **.p_o**, **.p_hi**).

Now we pass all of this information to the next incremental build by extracting the tarball during **preBuild**.

postInstall and **preFixup** take care of generating the tarball for the next incremental build.

Implementation of wrapper function

```

let buildIncrementally =
  { regularPackage, previousIncrement ? null }:
  (haskellLib.overrideCabal regularPackage
    (drv: {
      preBuild = lib.optionalString (previousIncrement != null) ''
        mkdir -p dist/build
        tar xzf ${previousIncrement.incremental}/dist.tar.gz -C dist/build
      '';
      postInstall = ''
        mkdir $incremental
        tar czf $incremental/dist.tar.gz -C dist/build \
          --mtime='1970-01-01T00:00:00Z' .
      '';
      preFixup = ''
        # Don't try to strip incremental build outputs
        outputs=${"\\" + "\${"}outputs[@]/incremental}
      '';
    })

```

```
    ).overrideAttrs (finalAttrs: previousAttrs: {
      outputs = previousAttrs.outputs ++ ["incremental"];
    });
in ...
```

Using the Wrapper with Nix Flakes

It's a bit tricky to get this to work with flakes.

Our flake requires two inputs (`nixpkgs` and `incremental`).

The input `incremental` is pointing to the upstream of the current git repository. We need this to access a prior version of this flake.

Flake inputs

```
{
  inputs = {
    nixpkgs = {
      type = "github";
      owner = "NixOS";
      repo = "nixpkgs";
      ref = "nixpkgs-unstable";
    };
    incremental = {
      type = "github";
      owner = "example-user";
      repo = "example-package";
      ref = "master";
      inputs.nixpkgs.follows = "nixpkgs";
      inputs.incremental.follows = "incremental";
    };
  };
  # ...
}
```

The packages `default` and `incremental` have already been explained.

Flake outputs

```
{
  # ...
  outputs = { self, nixpkgs, incremental }: {

    packages.x86_64-linux.default = ...; # same as above
    packages.x86_64-linux.incremental = ...; # same as above

    packages.x86_64-linux.incrementalBase =
      with import nixpkgs { system = "x86_64-linux"; };
  };
}
```

```
with import ./nix/haskell/lib.nix {
  lib = pkgs.lib;
  haskellLib = pkgs.haskell.lib;
};
buildIncrementally {
  regularPackage = self.packages.x86_64-linux.default;
};

};
}
```

We also added `incrementalBase`, which produces the same result as `incremental`, but doesn't depend on earlier versions. We are using it to set `previousIncrement` in the `incremental` package.

NOTE In the definition of `incrementalBase` we don't pass any old derivation into `buildIncrementally`. We need `incrementalBase` to avoid an infinite recursion.

Conclusion

Overall the wrapper function `buildIncrementally` can speed up compilation a whole lot. The more modules you have, the more time will be saved. With hundreds of modules you can likely reduce your CI time by 90% with the `incremental` output.

This can probably be built into `callCabal2nix`, which would make the interface a bit more comfortable.

The integration with flakes feels a bit awkward, but I guess it works.

Examples

- I played around with this approach [here](#). With flakes enabled you can easily try it out by building an output.

```
nix build --print-build-logs \
  'github:jumper149/consuming-parser/incremental#incremental'
```

You will notice that you are building `consuming-parser` twice. First you will build it regularly via `incrementalBase` and then you will also build `incremental`, which doesn't have to compile any modules, because they are already provided by `incrementalBase`.

- Harry has an even smaller [example](#), that doesn't use nix flakes.