Crafting Extensible Incremental Parallel Embedded Build Systems

Lucas Escot
l.f.b.escot@tudelft.nl
TU Delft
Delft, Netherlands

ABSTRACT

Despite their ubiquitous use in the daily lives of programmers, build systems have only recently started being the subject of formal investigation. Upon closer inspection, they span accross a spectrum of flavours, possibly satisfying a wild range of properties: Backwards, forward, error-recovering, parallelizable, etc. In this paper, we present ongoing work in designing an extensible embedded domain-specific language for writing forward build tasks. Rather than advertise a new shiny tool that is going to rule them all, we focus our attention to a small set of abstractions that simplify the treatment of incrementality, caching and parallelism. We purposefully move the focus away from file dependencies and side effects to give attention to compositionality of tasks, structural caching of host programs and tracking of value change. We showcase an hybrid approach between incremental computing and build systems, where logical constructs in the host language can too be cached and made incremental.

ACM Reference Format:
Lucas Escot. 2023. Crafting Extensible Incremental Parallel Embedded Build Systems. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As Mokhov et al. [13] put it, "build systems automate the execution of repeatable tasks for individual users and large organisations". Examples that come to mind — like Make [8] — are commonplace in software compilation pipelines. But build systems need not produce only software. Configurable static site generators, that give users a way to specify how to construct their websites from a collection of input files, are also build systems. Excel itself can be construed as some kind of build system [13].

Build systems do not simply execute tasks sequentially and systematically. Their most desired feature is to avoid unnecessary work by only executing tasks that *need to be rebuilt.* For this purpose, build systems typically process *build descriptions* written in a domain-specific language, that specificy the different tasks that compose the build process. These descriptions must also make explicit the dependencies between the tasks, so as to know in which order to execute them, and whether an output of a task being updated shall impact some other task.

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

As software projects grow, so do their compilation pipelines, and what once were simple build scripts can turn into piles of unmaintainable technical debt. Out of the necessity for build scripts to be reliable and somewhat easier to maintain arose *embedded* build systems — like Shake [14]. Those systems rip benefit from an existing general-purpose language (here, Haskell), its strong type system and ecosystem. Still, a significant part of the complexity of defining *correct* build rules resides in accurately stating the dependencies between tasks. Some systems such as Rattle [15] go a step further by deriving these dependencies automatically. Both of these systems — and build systems generally — run tasks by invoking external third-party programs — compilers, linkers, etc. If the domain-specific language (DSL) provided by the build system gives control logic primitives, it usually falls outside of the task layer and is *always* executed on every restart of the build process. In this paper, we propose to move the focus away from external commands to *host language computations.* What if tasks were constructed from programs written in the same language we embed our descriptions in? How would we go about making those incremental?

Contributions. We put forward the following contributions:

- We motivate and describe a new point in the design space of build systems, lying somewhere in between embedded forward build systems and incremental programming.
- We describe a new set of abstractions that facilitate reasoning about task dependency and lack thereof. These abstractions ease somewhat the difficulty of providing a user-friendly syntax, by taking inspiration from the categorical setting.
- We showcase a work in progress *simple* prototype implementation of such a build system, as an embedded domain-specific language in Haskell.

2 A PRIMER ON BUILD SYSTEMS

2.1 Build systems

A build system is a tool whose sole purpose is to produce artifacts according to a user-defined specification. These artifacts are commonly software executables, stemming from the compilation of multiple files of code. As discussed by [13], they tend to share the same following components:

- They provide a domain-specific *language* or interface to specify *build rules* or *tasks.*
- They maintain a *store of values.* For traditional build systems we can conflate values with files on disk. We can distinguish *input*, *intermediate* and *output* values.
- They maintain persistent build information — a *cache* — that they rely on during consecutive runs to avoid repeating work.

- Rules explain how to update a specific value, and have *dependencies* on other values, dependencies that can be either static or dynamic.

Additionally, build systems are expected to satisfy a handful of properties:

- The bare minimum is to be **correct**, that is to consistently produce values that are up to date, according to the specified build rules.
- Another goal is **minimality**, that is to only run tasks that are strictly necessary for correctness to be satisfied, and values be brought up to date since the last run.
- Build systems that have full knowledge of the dependency graph may attempt running independent tasks in **parallel**.
- **Self-tracking** build systems are also minimal with respect to an updated task description. In particular, having an updated task description shall not produce incorrect builds, nor necessarily require a *full* rebuild for the store of values to be refreshed.
- In some situations, it may be desirable to have *error-recovery*, and not fail as soon as a task fail but still execute all tasks that can be executed to completion.
- Most build system also strive at *extensibility*, in the sense that the task language ought to be expressive enough to allow users to express all the tasks transformations they desire.

2.2 A fresh look

While build systems typically focus on external commands to do *the actual work*, we strive to make tasks out of regular Haskell programs. Indeed, the Haskell ecosystem is full of convenient libraries, compilers and other non-trivial programs, and it would be a shame not to be able to use them directly in an embedded build system. As some of those programs may be computationally intensive, there should be a way to consistently cache them just as external commands. We detail below some design decisions that are instanciated and developed in section 4.

Oblivious to side effects. With this new perspective, side effects of tasks — in the Haskell sense, interactions with the outside world — become less interesting. In existing build systems, it is important to know that a task produces a certain file before this file is consumed by another task. In our setting, this file would contain intermediate information. But if both tasks are Haskell programs, this information need not be stored in a file, it could just be passed — as a Haskell value — between both tasks. Therefore, interactions with the file system are not so interesting anymore, to the extent that we just need to know whether this side effect depended on something that *may* have changed since the last execution. For this reason, we will allow *tasks* to have any kind of side effects, so long as they return which external dependencies would require running them again.

Tasks always run. Rather than have the build system be an orchestrator that decides which tasks should be executed again to reach correctness considering changed inputs, we leave this decision to task themselves. In other words, the build system will always run all[1] the tasks, and it becomes the responsibility of a task itself to decide whether it has to repeat all the work again to produce the expected value. The role of the build system amounts to communicating to tasks whether the input they receive has changed since the last execution. For tasks to be able to *not* repeat work needlessly, they need a way to store persistent information between executions, hence:

Structural caching. Because tasks are no longer identified by their target, we propose to have their *identity* be their syntactic location in the build descrition. And because we want such tasks to have access to a local cache they can use as they wish, persistent between builds, we need a way to manage the cache of every tasks. Our solution is to have a cache whose structure directly maps to the syntactic structure of the build description. By ensuring that the representation of the build description is *static*, we make sure that cache fragments will always be distributed to the same tasks.

Variable binding. The traditional mapping between *output targets* and tasks no longer makes sense, as tasks now produce Haskell *values*. The process of specifying targets amounts to simply *naming* tasks and specifying their *type*, in other words: binding a variable. Likewise, a task depending on another means it relies on the *value* computed by the latter, precisely because we've argued it should *not* rely on external effects from the task. Therefore, task dependencies map precisely to variable use.

Extensibility. We intend the build system to be extensible in the sense that new task operations ought to be created by external users. We distinguish three layers of interaction with the build system:

(1) At the first layer is a core set of primitives provided by the build system itself, introduced in section 4.
(2) On the second layer, library designers or experienced users can extend the build system language with new *domain-specific* build transformations, that they have to make sure are *correct* and *well-behaved*.
(3) On the last layer, end users rely on the language provided by the build system and the primitives from external libraries. They do not have to think about dependencies as this matter will have been solved at the abstraction level in the two layers beneath. The syntax manipulated by end users is shown in section 3.

3 AN EXTENSIBLE BUILD SYSTEM

We showcase here a toy extensible build system embedded as a Haskell library [2], developed using the ideas we introduce in section 2 and 4. This EDSL is not meant to be a generic build system, but rather specifically tailored for static website generation; an area where other DSLs have shined in the past [18].

3.1 A quick tour

Task abstraction. The only abstraction exposed to end users from our core library is the `Task` abstraction. An object of type `Task m a` models an *incremental* effectful computation that, when executed, will produce a value of type `a` while running side effects in some

[1]Unless `match`, `cached` and brancing combinators come into play
[2]Publicly available over at https://github.com/flupe/achille

monad m. We provide instances for Functor and Applicative, which grants us plenty of operations on tasks: any host value can be lifted into a (constant, pure) task; any pure function can be lifted into a function over tasks, etc.

```
pure   :: Monad m => a -> Task m a
fmap   :: Monad m => (a -> b) -> Task m a -> Task m b
liftA2 :: Monad m => (a -> b -> c)
       -> Task m a -> Task m b -> Task m c
```

When a function is lifted to tasks using fmap, the most conservative assumption our library makes is to consider the given function *injective*. If the input task has an updated output since the last execution, then the lifted function applied to this task will also be considered to have a different output.

Task sequencing and assignment. Tasks can be freely sequenced and bound to variables using Haskell's *do* notation:

```
doThing :: Task m Int
doStuff :: Task m String -> Task m Int -> Task m ()

build = A.do thing <- doThing
             doStuff "hello" thing
```

This notation shouldn't be confused with Haskell's usual monadic *do* notation. In the example above, variable thing still has type Task m Int. We do not provide nor make it possible to define the monadic bind operation, which is precisely how our system is able to keep track of the flow of values through task descriptions.

Effectful tasks. We provide additional tasks to interact with the filesystem:

```
readText :: Task IO Path -> Task IO Text
readBS   :: Task IO Path -> Task IO ByteString
write    :: Writeable a
         => Task IO Path -> Task IO a -> Task IO a
```

Most of the *computationally expensive work* in traditional build systems is typically executed by *external* programs or commands. In contrast, in our language we don't put such an emphasis on commands, and create tasks out of regular Haskell programs. Even if we can invoke external programs, we discourage the use of intermediate files as a means of communication between programs, and rather encourage library designers to provide interfaces that reflect dependencies between tasks as explicit value dependencies in the host language.

Incrementality and caching. To avoid recomputing things needlessly on every run, we provide a *caching* combinator that stores the intermediate result in a *cache*.

```
cached :: Binary a => Task m a -> Task m a
```

When executed, cached t will only *recompute* t if its dynamic or static dependencies have changed. In a similar vein, the match combinator can prove very useful to incrementally process files in the exact same way:

```
match :: (Binary a, Eq a)
      => Pattern -> (Task m Path -> Task m a)
      -> Task m [a]
```

Pattern is the type we give to *glob* patterns such as "posts/*.md" that will match all files with the .md extension in the posts/ folder.

match pat t, when executed for the first time, will invoke t on every path matching the pattern pat, cache the result, and collect all outputs into a list. When it is executed again later, it only triggers evaluation of t on paths whose underlying file has been updated since the last time, *or* if t depends on values that have changed, thus making match properly incremental.

Extensibility. This overview is by no means exhaustive, and we provide many more usual Haskell functions that have been lifted over tasks and made properly incremental. Additionally, our language is extensible in such a way that other library designers can introduce new incremental operations on tasks. We've already designed such prototype extensions for parsing (and caching) *YAML* files into Haskell types, parsing (and caching) Mustache [2] templates[3], and processing Markdown files and their associated front-matter using the Pandoc [3] library. All three are shown in figure 1.

3.2 Toy example

```
import Achille as A
import Achille.Stache (loadTemplate, applyTemplate)
import Achille.Pandoc (processPandocMeta)

main :: IO ()
main = achille A.do
  -- load mustache template for blog posts
  tPost <- loadTemplate "templates/page.mustache"

  -- load mustache template for index page
  tIndex <- loadTemplate "templates/index.mustache"

  posts :: Task IO [Path] <-
    match "posts/*.md" \src -> A.do
      -- parse markdown and frontmatter of an article
      content <- processPandoc src

      -- apply mustache template and write HTML
      write (src -<.> "html")
        (applyTemplate tPost content)

  -- write index page with list of articles
  write "index.html" (applyTemplate tIndex posts)
```

Figure 1: Build description written using our EDSL

Figure 1 shows a simplistic task description, that already showcases many components of the library. This task description states the following build steps:

(1) Load templates for blog posts and the index page.
(2) For every Markdown file located in the posts/ folder of the source directory:
    (a) Read the file and convert it to HTML using the *Pandoc* library.
    (b) Apply the *mustache* template to the article and write to the appropriate output file.

[3]A format used to specify how to turn data into raw text — in our case HTML.

(3) After retrieving all the URLs of the generated blog posts, render the index page using and write it to disk.

Once compiled, this Haskell code becomes a custom static website generator. When first executed, the program is run as one would expect. Of note is the fact that all articles are generated in parallel due to the `match` combinator. A source directory with the following structure:

```
├ templates/
│  ├── page.mustache
│  └── index.mustache
└ posts/
   ├── one.md
   └── two.md
```

will produce an output directory with the following files once the program is executed:

```
├ index.html
└ posts/
   ├── one.html
   └── two.html
```

Incrementality. The generated program is incremental, which can be observed in the following situations:

- If we run the program immediately without changing anything, neither the templates nor the Markdown file will be read from. No file is written to. Indeed, nothing has changed.
- If we modify `posts/one.md` and run again, only this file will be processed again, and the index be updated. The templates, stored in the cache, will not be processed.
- If we modify `template/post.mustache` and run again, both markdown files will be rendered once again, as the inner task of the `match` combinator depends on the template. The index will not be written to, since the result of the `match ...` expression remains unchanged.

4 ABSTRACTIONS

In this section, we describe some of the abstractions that lie behind the prototype showcased in section 3. We explain how we model the change of *values* (4.1) during execution. We introduce *recipes* (4.6) as transformations between values, equipped with a local *cache* (4.4) and executed in some *context* (4.3). These transformations communicate which dynamic dependencies (4.2) they relied on during execution. *Valid* transformations form a *category* which makes reasonning about *composition* and parallelism straightforward. Finally, we provide a *pointful* syntax for constructing tasks.

4.1 Values

Library designers — that will come to write the domain-specific building blocks of the build system — need to write programs and computations that are incremental by nature, that is, computations that explicitely tell, given *changes* of inputs, whether the result of the computation *changes*. Before we specify in detail what me mean by these computations, we first need to make precise what we mean by values.

A `Value a` is a *wrapper* over a value of type `a`, that additionally stores information about whether this value has changed *since the last run*.

```haskell
data Value a = Value
  { theVal     :: a
  , hasChanged :: Bool
  , changeInfo :: Maybe (ChangeInfo a)
  }

value :: Bool -> a -> Value a
value c x = Value x c Nothing
```

The field `hasChanged` is responsible for indicating whether the value changed since the last run. Yet there is one extra field called `changeInfo`. Its reason to be is simple: for more elaborate types — say `(a, b)` — it's not enough to know if a value of such type changed, but additionally *how* it changed, or rather, *where*. In the case of `(a, b)`, a value may only have changed if either its left component or right one changed. We introduce the `Diffable` class to refine this information:

```haskell
class Diffable a where
  type ChangeInfo a = r | r -> a
  splitValue :: Value a -> ChangeInfo a
  joinValue  :: ChangeInfo a -> Value a

instance Diffable (a, b) where
  type ChangeInfo (a, b) = (Value a, Value b)
  splitValue (Value _ _ (Just vs)) = vs
  splitValue (Value (x, y) c Nothing) =
    (value c x, value c y)

  joinValue vs@(vx, vy) =
    Value (theVal vx, theVal vy)
          (hasChanged vx || hasChanged vy)
          (Just vs)
```

In effect, we've explained how to separate a `Value (a, b)` into both projections `Value a` and `Value b`, and the reverse operation. We require that `splitValue . joinValue` be the identity, so that a value of a *composite* types constructed out of smaller components retain information about change of the subvalues. Without going into detail, it's possible to define many more `Diffable` instances for usual Haskell types, like `[a]`, `Map k v`, `Either a b`, etc.

We don't simply make `Value` itself be a type family part of the `Diffable` class for practicality. If we did so, we would have to require that all — possibly user-defined — types used in our DSL provide such an instance.

4.2 Dynamic dependencies

Dynamic dependencies are an abstraction over external resources that cannot be modeled by mere *values*. File is the most common example, and is the only one we define here. Consider the task that reads the contents of a file given a task producing a *path*:

```haskell
readText :: Task IO Path -> Task IO Text
```

`readText t` has an *explicit* dependency over its input *t*. If the path changes, then `readText t` ought to run again. Additionally, once executed, it has a *dynamic* dependency over the file itself.

If the path computed by `t` doesn not change, but the file itself is updated, then `readText t` must run again and consider its output changed.

```
data DynDeps = Deps
  { fileDeps :: Set Path
  , globDeps :: [Pattern]
  }
```

In our language, we can either depend on a specific file — say, when doing `readText "file.txt"` — or on an entire *glob* pattern, when using the `match "posts/*.md"` combinator. The first kind of dependency gets *dirty* when the file itself is *modified*, while the latter becomes *dirty* when files suddenly *match* the pattern by being created, or no longer — they've been deleted. We make sure that `DynDeps` is both a `Semigroup` and a `Monoid`, so that we can merge group of dynamic dependencies when combining tasks later on:

```
mempty :: DynDeps
(<>)   :: DynDeps -> DynDeps -> DynDeps
```

When a full build is executed, we collect all the dynamic dependencies and store them in the *cache* (4.4). When a build is executed again, we retrieve the dynamic dependencies from the cache, check whether some of them have become *dirty*, and store this information in the *context* (4.3).

For prototyping purposes, we use *modification time* as a measure of file difference. It is known that even though this mechanism is widely used in existing build systems, it is not without fault, and more robust mechanism like relying on hashes would be preferable. We leave this for future work, even though it should just be an implementation detail of the `DynDeps` abstraction.

4.3 Context

Every computation of our system will run in a given `Context`. Information contained in the latter will certainly be domain-specific — eg. which input folder we're currently in — but it should at least contain:

- the current time ;
- the last time the build was executed ;
- a set of dynamic dependencies that are *dirty*.

4.4 Cache

Our `Cache` abstraction is a wrapper aroung Haskell's *lazy* `Bytestring`. For future-proofing, we use a `newtype` and only expose the following two operations:

```
newtype Cache
emptyCache :: Cache
toCache    :: Binary a => a -> Cache
fromCache  :: Binary a => Cache -> Maybe a
```

Any binary-encodable value can be transformed into a cache, and *possibly* retrieved from a cache. Given these two operations, we can define the following constructions for glueing two caches into one, and split them back:

```
joinCache  :: Cache -> Cache -> Cache
splitCache :: Cache -> (Cache, Cache)
```

We expect `splitCache (joinCache ca cb) = (ca, cb)`.

4.5 Primitive tasks

A primitive task `PrimTask m a` is a *computation* in some *monad* `m`, in some given `Context`, with access to a local `Cache`, *possibly* producing a value of the required type `a`, along with an *updated cache* and a list of its *dynamic dependencies*. [4]

```
type PrimTask m a
  = Context
  -> Cache
  -> m (Maybe a, Cache, DynDeps)

forward :: Monad m => Maybe a -> PrimTask m a
```

This abstraction is without surprise a *monad* (assuming `m` is too), with the property that two primitive tasks in sequence get their dynamic dependencies *merged*, and the second one is executed only if the first one succeeds.

4.6 Recipes

Finally, we introduce perhaps the most important abstraction: *recipes*. Because we emphasize that caching should be a *local issue*, and that the responsability of deciding whether to run again hence falls upon task themselves, they need to have some awareness about how their input evolves between runs. Knowing whether the input has changed at all is a start, but in some situations knowing how it has changed is the thing we want. Because build rules can be viewed as a set of *transformations* applied to inputs, said transformations need communicate *how their output changes* in relation to change of inputs. A recipe from `a` to `b` is thus to be understood as a transformation from *values* of type `a` to values of type `b`.

```
type PrimRecipe m a b = Value a -> PrimTask m (Value b)
```

Because it uses our previous definition `PrimTask`, a recipe is innately equipped with *its very own cache* and ran in some *context*. We consider a recipe to be *valid* if it is *deterministic* with regards to its inputs and its dynamic dependencies. If neither the inputs nor any of the dynamic dependencies have changed since the last run, a recipe should *always* compute the same value. Additionally, we impose that dependencies between transformations should be reflected at the type-level, in the sense that one should depend on the *output* of the other. Implicit dependencies — such as a transformation writing to a file that will be read by another — are forbidden unless this dependency manifests itself as a value that gets passed between the transformations.

Our recipe construction looks like a glorified Kleisli arrow, and it's only natural to expect it to be compositional. Additionally, because dependency between recipes should always be stated as a dependency over values, two recipes that don't rely on either's output can always be considered independent and thus run in parallel. Rather than define composition and other combinators on primitive recipes directly, we instead introduce `Recipe m` as the *free cartesian category* over primitive recipes, with a GADT. The motivation for doing so will become clear in a moment.

```
data Recipe m a b where
  Embed :: PrimRecipe m a b -> Recipe m a b
```

[4]The actual definition uses a long stack of monad *transformers* to benefit from as much free constructions as possible and avoid boilerplate. Rather than spend some time introducing each of those transformers to the reader, it suffices to understand that `PrimTask m a` is equivalent to the type written

```
  Id   :: Recipe m a a
  Comp :: Recipe m b c -> Recipe m a b -> Recipe m a c
  Exl  :: Recipe m (a, b) a
  Exr  :: Recipe m (a, b) b
  Void :: Recipe m a ()

  (:***:) :: Recipe m a b -> Recipe m c d
          -> Recipe m (a, c) (b, d)
  (:&&&:) :: Recipe m a b -> Recipe m a c
          -> Recipe m a (b, c)
```

Saying that `Recipe m` forms a cartesian category simply states:

- For any input type `a`, we have the identity recipe `Id` that does nothing.
- Two recipes `f :: Recipe m a b` and `g :: Recipe m b c` can be sequenced into a third transformation `Comp g f`
- We can put two independent transformations in parallel (`:***:`), and have many operations to operate over `(a, b)`.

We can now give an instance for `Category (Recipe m)`. Note the definition of composition. Rather than use the constructor for sequential composition directly, we do some transformations. the first two cases remove unnecessery composition with the identity. the following case forces the same systematic ordering of sequences. This is what makes

```
instance Category (Recipe m) where
  id = Id

  Id . f        = f
  f  . Id       = f
  f  . Comp g h = (f . g) . h

  (p :***: q) . (f :***: g) = (p . f) :***: (q . g)
  (p :***: q) . (f :&&&: g) = (p . f) :&&&: (q . g)
  g           . f           = Comp g f
```

These optimisations stem directly from laws every cartesian category must abide by. The ones involving (`:***:`) and (`:&&&:`) move the parallelism combinators outside of function composition to avoid unnecessery synchronisation. One thing to note is that we don't apply *all* reductions allowed by category laws. An example is the following law stating that composing the left projecton with two morphisms put in parallel is equal to the first.

```
Exl . (f :&&&: g) = f
```

While this may be true for in the abstract setting, this definition does not account for *side effects* that may be produced by `g`.

Now that we have a syntax for recipes that we can inspect and manipulate, we define an evaluator to run these recipes. This evaluator makes it explicit how we make sure to construct and propagate cache chunks that closely *follow the structure* of the syntax:

```
runRecipe :: Monad m => Recipe m a b -> PrimRecipe m a b
runRecipe r vx = case r of
  -- ...
  f :***: g -> do
    let (va, vb) = splitValue vx
    (cf, cg)  <- splitCache
    (vc, cf') <- withCache cf (runRecipe f va)
    (vd, cg') <- withCache cf (runRecipe f va)
    joinCache cf' cg'
    forward (joinValue <$> ((,) vc <*> vd))
```

In the snippet above illustrating how we evaluate two independent recipes receiving the same input, we can note that:

- The incoming cache is split in two separate caches, for either recipes `f` and `g`.
- They are executed *independently* with the appropriate smaller caches.
- The updated smaller caches are joined together and become the new cache of the outer recipe (`f :***: g`).
- The output of the recipe succeeds if both `f` and `g` succeed, and keeps track of local change information thanks to `splitValue`.

5 A POINTFUL SYNTAX

While we just settled on an abstraction enabling us to model parallelism and explicit static dependencies over host values that flow between tasks, it should become clear that nobody in their right mind would be willing to construct build descriptions using these primitives. The heart of the problem is the lack of *points*. Morphisms in our category can only be constructed in a *point-free* style using the operations of cartesian categories. Manageable as it is for small definitions, this process quickly gets tedious. Taking a look at the example below,

```
build :: Recipe IO () ()
build =
  loadTemplates "templates" >>>
  match_ "posts/*.md"
    ((processMarkdown *** (lookup <<< (id *** "index")))
    >>> swap >>> applyTemplate >>> write ...)
```

Looking past the precise definition of the primitives used, it should become clear that this kind of code is hard to decipher and maintain. If we had an hypothetical syntax with binders, we could write an equivalent snippet:

```
build :: Task IO ()
build = A.do
  templates <- loadTemplates "templates"
  match "posts/*.md" \src -> do
    content <- processMarkdown src
    applyTemplate (templates ! "index") content
      & write (src -<.> "html")
```

The only notable difference is the ability to define intermediate variables. The categorical structure of our `Recipe` abstraction suggest that we can provide a pointful syntax for recipes [7][5]. However, we found existing solutions to be lacking, either because they enforce relying on experimental compiler plugins [7] or necessitate using Linear Haskell in a context where linearity is not needed [5]. Instead, we implement a variant of HOAS à la Atkey et al. [4].

5.1 A language for points

We introduce a separate language specifically for constructing points in our category, that is, *programs* that do not have inputs.

```
data Program m a where
  Var  :: Int -> Program e m a
  Let  :: Program m a -> Program m b -> Program m b

  App  :: Recipe m a b -> Program m a -> Program m b

  Seq  :: Program m a -> Program m b -> Program m b
  Pair :: Program m a -> Program m b -> Program m (a, b)
  Match :: Binary a => Program m Pattern -> IntSet
        -> Program m a -> Program m [a]
  Cached :: Binary a => IntSet
         -> Program m a -> Program m a
  Prim :: PrimTask m (Value a) -> Program m a
  Ite  :: Program m Bool
       -> Program m a -> Program m a
       -> Program m a
```

Constructor `Let x t` binds program `x` to a fresh variable when executing program `t`. Constructor `Var k` represents a program returning the variable with de Bruijn level k in the environment. The `App r t` constructor applies any recipe `r` to a program `t`. This separation into programs — that carry an environment — and recipes — that do not have a notion of variables — makes it easier for library designer to introduce new transformations without having to reason about bindings.

Then, we provide more program constructions such as conditional branching (`Ite`), parallel sequencing (`Seq`) and glob pattern-matching (`Match`). In the case of `Match`, we store the output of the inner task executed on matching paths in the cache. If a file hasn't changed since the last run, we can thus avoid duplicating work. However, if *any* of the variables used in the inner program have changed, we force execution, which is why we keep track of the free variables of the inner task in the `Match` constructor, as an `IntSet`. Likewise, we keep the dynamic dependencies of all branches in the cache of the `Match` program, so that if any of those become dirty we also ignore the cache and recompute the output. This guarantees that the output of `Match` is always sound with respect to host value and dynamic dependencies, in spite of caching.

We again can evaluate programs in a context and with an associated cache:

```
runProgramIn
  :: (Monad m, MonadFail m, AchilleIO m)
  => Env -> Program m a -> PrimTask m (Value a)

runProgramIn env t = case t of
  -- ...
  Seq x y -> do
    (cx, cy) <- splitCache
    (_,  cx) <- withCache cx $ runProgramIn env x
    (vy, cy) <- withCache cy $ runProgramIn env y
    joinCache cx cy
    forward vy
```

Similarly to parallel conjunction of recipes, not only can programs that are sequenced be run in parallel, but failure of the first currently does not prevent the second one to run to completion.

5.2 Higher-order syntax

To provide a user syntax that desugars into the first-order representation of programs introduced earlier, we follow the guidelines from Atkey et al. [4]. We define `Task m a` as an *opaque* wrapper around a function producing a `Program m a` for a given number of bound variables in scope.

```
newtype Task m a
  = T { unTask :: Int -> (Program m a, IntSet) }
```

This is the only piece of abstraction that end users have to deal with. In order to execute a task, we retrieve the underlying program in an empty environment, and execute it.

```
runTask :: (Monad m, MonadFail m, AchilleIO m)
        => Task m a -> Context -> Cache
        -> m (Maybe (Value a), Cache, DynDeps)
runTask (T p) = runProgramIn emptyEnv (fst (p 0))
```

Such definition of `Task m a` enables us to provide a convenient higher-order `match` combinator, which takes a function as argument and yet can be converted into a first-order representation seamlessly:

```
match :: (Binary b, Eq b)
      => Pattern -> (Task m Path -> Task m b)
      -> Task m [b]
match pat t = T \n ->
  -- we remove locally-bound variables
  let (t', IntSet.filter (< n) -> vst)
        = unTask (t $ T $ const (Var n, IntSet.empty))
                 (n + 1)
  in (Match pat t' vst, vst)
```

To provide a convenient user-syntax for variable bindings, we rely on the `QualifiedDo` Haskell language extension [1]. We define combinators `(>>)` and `(>>=)` that desugar into respectively `Seq` and `Let`:

```
(>>)  :: Task m a -> Task m b -> Task m b
(>>=) :: Task m a -> (Task m a -> Task m b) -> Task m b
```

Thanks to the `QualifiedDo` extension, users can reuse the standard do notation in Haskell to sequence tasks and assign them to variables. The following code:

```
build = A.do
  x <- doSomething
  doSomethingElse x
```

gets desugared into the following:

```
build = doSomething A.>>= doSomethingElse
```

which when evaluated reduces to the following syntax tree:

```
Let doSomething (doSomethingElse (Var 0))
```

Using such overloading of monadic operators is to our knowledge the simplest way to obtain a syntax with explicit binders. We certainly cannot prevent users from relying on the Haskell `let` construct, which causes task duplication. There are known techniques [11] to overcome this and recover implicit sharing in EDSLs, but they require significant development effort, so we leave this for future work to assess whether they are desirable.

6 RELATED WORK

6.1 Build systems

Given the experimental nature of our system and the departure from traditional build systems, there is little point giving a detailed comparision between our language and a specific reference point. We only highlight here a few high-level differences and similarities. Shake [14] and more recently Rattle [15] are two build systems embedded in the Haskell language. This choice is motivated by the practicality of reusing an existing language's type system and ecosystem to ease the definition of rules. Shake fits in the formal framework given by Mokhov et al. [13], in the sense that it structures build descriptions as dictionaries mapping keys (files) to tasks that compute values for said keys. In this regard, we are much closer to Rattle, a forward build system that gets away with clearly identifying tasks by target and instead putting the emphasis on composition and combination of tasks through a layer of control logic. Still, Shake like Rattle are used to schedule and invoke external programs and commands. Though Rattle uses system tracing — observing filesystem accesses during execution — to fully and automatically infer dynamic dependencies, the control logic layer is always executed and commands are the only point at which incrementalization happens. With our system, we investigate whether this logical layer in the host language can too be made incremental. Rattle exposes a monadic abstraction for constructing build rules, which is precisely what we avoid in order to gain some level of introspection over the structure of tasks. Even though their choice for doing so is motivated, we found that a non-monadic abstraction can actually be made practical and intuitive. We note that the approach taken by Rattle — inferring dynamic dependencies with system tracing and speculating parallelism — and by us — stating dynamic dependencies explicitely at the abstraction layer — are not incompatible. We could imagine providing primitives for calling commands that would discover dependencies just like Rattle does, and not be as optimistic about parallelism and implement a similar speculative scheduler.

6.2 Incremental programming

Our approach to build systems turned out to be quite similar to the notion of incremental programming. In such a setting, given a base program, the question is how to efficiently compute the necessary change of outputs to reflect a change of inputs. This is typically achieved by computing the differential of the initial program. In the work of Giarrusso [9], any incremental lambda calculus program can thus be made incremental, and the calculus itself is designed to be extensible with plugins, in the same way recipes can be introduced in our DSL. Incremental programming as a discipline is typically only considering pure programs operating on values of types that we know how to differentiate somewhat efficiently. A build system, in contrast, must know when to trigger side-effects after inputs have changed. Because of this, we're not so keen on knowing precisely how inputs have changed, merely that they have since the last time we executed the current task. Our cache closely mimicking the structure of programs turned out to be quite similar to the Cache-Transfer-Style transformation of Giarrusso et al. [10], whereby each syntactic construct is equipped with a cache storing the value of the previous run. In practice, storing every single intermediate value is quite the overhead. It forces users to make every they manipulate serializable, something which may not even be possible if said value is higher-order. That's why we rather introduce an explicit caching combinator.

6.3 Haskell EDSLs

Once we settled on the `Recipe` abstraction for representing incremental, cached transformations, the hard part remained how to provide a more legible syntax to end users. It is well-known that cartesian categories model precisely "box-and-wire" flow diagrams describing computation graphs, so we already knew that our abstraction ought to be somewhat expressive enough to describe build tasks. The GHC plugin from Elliott [7] seemed to be the perfect solution as it transforms regular Haskell definitions — with named variables — into their categorical counterpart, for a category of our choice, free of variables. Sadly, this plugin proved to be very hard to work with, hard to integrate and extend. Additionally, the naive translation from Haskell terms to category morphisms reifies the entire initial context into an input of the resulting morphism, even if some variables in the context are not used, which inhibits minimality of incrementality. The solution from Bernardy and Spiwack [5] served as a basis for our syntax, as it provides the same functionality as the former without having to rely on compiler plugins, through a small library. However, implementing the higher-order `match` combinator with such an encoding of ports proved to be difficult if not straight impossible. McDonell et al. [12] showcase how to enable pattern-matching for embedded languages in Haskell, a feature that we have yet to implement. Currently all our syntax transformations happen at runtime, which so far hasn't been a bottleneck. There is much inspiration to draw from Willis et al. [19], who show how to transform higher-order combinator into efficient parsers at compile-time. They also showcase a way to recover not only implicit sharing but also recursion — which we so far have ignored — into the internal syntax of their DSL.

6.4 Static-site generation

Our prototype started as a language for incremental static website generation, until we realized the abstrations we defined were applicable to build systems in general. Still, our prototype itself is very much intended to be used for the specific domain of static site generation, and was directly inspired by Hakyll [18], another language embedded in Haskell. Hakyll is much more restrictive in the kind of tasks it supports — called `Rules` — and enforces that every such rule produces exactly one output. Reusing intermediate computations between rules require users to relies on a global store indexed with glob patterns and coercions. This motivated our design choice to have the build system be oblivious to side effects — in that a file can be processed into as many output files as one wishes — and rely on regular Haskell variables to preserve and depend on intermediate results from the build process. However, our current language is still far from feature-complete, and many design patterns supported by Hakyll do not have a ready-made equivalent yet in our library. In no particular order: handling of tags, pagination and global state are left as future work.

7 FUTURE WORK

In this paper, we mainly focused on motivating such a point in the design space of build systems and introducing a set of abstractions that ease reasonning about task ordering, parallelism and dynamic dependencies. However, much is left to be investigated.

7.1 Performance analysis

We only implemented a small prototype so far, and because of our limited ability to write efficient Haskell programs, we are certain that our current implementation could be drastically improved. Our implementation of concurrent tasks, albeit quite easy to implement, is rather naive. We haven't included nor performed benchmarks that evaluate the performance benefits of our tool. This can in part be explained by the fact that we are not trying to promote our own tool, but rather put forward simple and convenient abstractions that could lead to more principled, convenient and efficient build systems. Still, because we argue for making more of the control logic incremental, supporting our claims with actual benchmarks would be good.

7.2 Formal verification

In parallel to our prototype implementation, we provide a small test suite that ensures the primitives of our library are correct and well-behaved with regards to incrementality. An interesting direction would be to give a formal description of our abstractions and algorithms, so as to formally state and certify the correctness properties of our language constructs. Such formalization has been recently carried out by Spall et al. [16] for Rattle [15], using Agda [17] as a proof language. Using a tool such as Agda2HS [6] could even allow us to only work with the formalization in Agda and automatically retrieve a readable, verified implementation in Haskell. However, the latter doesn't seem mature enough to easily verify the kind of Haskell code we write.

7.3 Self-tracking

A core property often overlooked by build system designers is self-tracking: the ability to detect changes to build tasks themselves and preserve correctness. In most build systems, changing the build task description requires running the build system from scratch. In some systems, where the control logic layer is never incrementalized and always executed — like Rattle — self-tracking is obtained for *free*. In our setting, we currently do not detect whether the task description has changed, thus there can be a mismatch between the structure of the old cache and the updated task description. Throughout our presentation however, we put the emphasis on enforcing that tasks are internally represented as a *static*, first-order tree structure. By only having to deal with a first-order static reprensentation of build descriptions, we recover some form of introspection typically lost by embedded build systems. This *should* enable us to handle self-tracking. Indeed, it should be possible to construct a tree such that every node in the syntax tree of a task gets attributed a *hash*. Not only would identifying hash mismatches enable our system to detect and react on program changes, it could also allow us to implement cache reconciliation heuristics so as to not throw away the previous cache entirely.

REFERENCES

[1] [n. d.]. *6.2.5. Qualified Do-Notation — Glasgow Haskell Compiler 9.9.20230802 User's Guide.* https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/qualified_do.html

[2] [n. d.]. *{{ Mustache }}.* https://mustache.github.io/

[3] [n. d.]. *Pandoc Documentation.* https://pandoc.org/

[4] Robert Atkey, Sam Lindley, and Jeremy Yallop. [n. d.]. Unembedding Domain-Specific Languages. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell* (Edinburgh Scotland, 2009-09-03). ACM, 37–48. https://doi.org/10.1145/1596638.1596644

[5] Jean-Philippe Bernardy and Arnaud Spiwack. [n. d.]. *Evaluating Linear Functions to Symmetric Monoidal Categories.* https://doi.org/10.48550/arXiv.2103.06195 arXiv:2103.06195 [cs]

[6] Jesper Cockx, Orestis Melkonian, Lucas Escot, James Chapman, and Ulf Norell. [n. d.]. Reasonable Agda Is Correct Haskell: Writing Verified Haskell Using Agda2hs. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium* (Ljubljana Slovenia, 2022-09-06). ACM, 108–122. https://doi.org/10.1145/3546189.3549920

[7] Conal Elliott. [n. d.]. Compiling to Categories. 1 ([n. d.]), 1–27. Issue ICFP. https://doi.org/10.1145/3110271

[8] Stuart I. Feldman. 1979. Make — a program for maintaining computer programs. *Software: Practice and Experience* 9, 4 (1979), 255–265. https://doi.org/10.1002/spe.4380090402 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380090402

[9] Paolo G Giarrusso. [n. d.]. Optimizing and Incrementalizing Higher-order Collection Queries by AST Transformation. ([n. d.]).

[10] Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. 2019. Incremental λ-Calculus in Cache-Transfer Style. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 553–580.

[11] Andy Gill. [n. d.]. Type-Safe Observable Sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell* (Edinburgh Scotland, 2009-09-03). ACM, 117–128. https://doi.org/10.1145/1596638.1596653

[12] Trevor L. McDonell, Joshua D. Meredith, and Gabriele Keller. [n. d.]. Embedded Pattern Matching. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium* (2022-09-06). 123–136. https://doi.org/10.1145/3546189.3549917 arXiv:2108.13114 [cs]

[13] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. [n. d.]. Build Systems à La Carte. 2 ([n. d.]), 1–29. Issue ICFP. https://doi.org/10.1145/3236774

[14] Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. 2016. Non-Recursive Make Considered Harmful: Build Systems at Scale. *SIGPLAN Not.* 51, 12 (sep 2016), 170–181. https://doi.org/10.1145/3241625.2976011

[15] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. 2020. Build scripts with perfect dependencies. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (nov 2020), 1–28. https://doi.org/10.1145/3428237

[16] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. 2022. Forward build systems, formally. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM. https://doi.org/10.1145/3497775.3503687

[17] Agda Development Team. [n. d.]. *Agda 2.6.3 Documentation.* https://agda.readthedocs.io/en/v2.6.3/

[18] Jasper Van der Jeugt. [n. d.]. *Hakyll.* https://jaspervdj.be/hakyll/

[19] Jamie Willis, Nicolas Wu, and Matthew Pickering. [n. d.]. Staged Selective Parser Combinators. 4 ([n. d.]), 1–30. Issue ICFP. https://doi.org/10.1145/3409002