A Deep Dive into Nomia

Shea Levy

April 21, 2021

> There are only two hard things in Computer Science: cache invalidation and naming things.
>
> —Phil Karlton

Naming and substitution are ubiquitous[1] in computation, and many systems end up dealing with them explicitly. Compilers take module names and substitute in appropriate symbol tables. Browsers take URLs and substitute in appropriate web sites. Package managers take package names and substitute in appropriate changes to your environment. In other cases, the system has no explicit handle on names but the user or programmer fills in: We refer to other pieces of code, or techniques, or other computations, or data sources, or a million other things by name. In implementation, we fill in a special-case substitution of that name that preserves the intended meaning[2].

These domains are, of course, very different. Browsers don't know what a symbol table is, and installing a package is distinct from translating source code to object code. But there are many conceptual commonalities between them, commonalities which in principle could allow for shared implementation and semantics. Unfortunately, most of the time the required functionality is reimplemented from scratch. Like any missed opportunity for reuse, this duplicates work and bugs, leaves many implementations incomplete with respect to functionality or performance, and increases cognitive overhead for users and developers. In this case we also miss opportunities for **cross**-system composition: My package manager may know how to install libpq, and my compiler may know how to resolve libpq-fe.h to a library once it's installed, but there's no general-purpose way to note that the one name links to the other. With the commonalities abstracted into a shared

[1] If you take the Church side of the Church-Turing thesis, name substitution is what computation **is**.

[2] We hope!

component, system implementers can focus on their domain expertise, and users can benefit from correctness, efficiency, and a coherent, easy to use experience across all of their systems.

Nomia is a system designed to provide a shared conceptual model and common implementation for substitution of named resources. This document describes how.

1 A vision of the future

Imagine you're a developer, and this is a typical day at work:

You're about to start working on a service that you haven't worked on before. You point your editor to the source file you need to work on, and after a slight delay for the first-time download, you have it open in front of you. You start modifying the code and, after a minute or two for the library dependencies to be downloaded, syntax checking kicks in, and points out a typo a few lines above. You save your change and jump over to another file that depends on it.

The file opens immediately, and you start making changes. At first, your editor, with a warning that it's not up-to-date, says there's an undefined reference at the location where you mentioned a function you added in the previous file. Then, after a few seconds for the first module to compile, those errors go away. You finish the work and point your browser to the local service URL. You get a page saying that the service is being spun up, with a detailed progress report on what remains to be done. You notice a database dump is being loaded and you know that will take a while, so you switch to another project, a compute pipeline.

When you enter the development environment for the compute pipeline, you notice a huge merge has just happened. You were up to date with master, so you don't have any conflicts, but you're not sure if the merge impacted the parts of the codebase you care about. A colleague recommends you try out Meld to review the diff, so you run a command to launch it. Then, after a minute of downloading, since you've never used Meld before, the window pops open and you confirm the merge was fine. You're testing out an algorithm tweak on the last stage that should only impact a small portion of the parallel work units of the pipeline. Although the CI system hasn't noticed the merge yet, when you open the module you notice the modules it depends on have already been fetched from when the original developer built the branch.

After you finish implementation, you kick off a dry-run of the pipeline and confirm that it will produce the same as that day's production run except at the end, for the subset of the data your change impacts. You kick off the job, and while that's running, you switch back to your browser and see the service is loaded. You check your work and, satisfied, open a PR. Automatically, the QA team is mentioned with a URL to test out. You notice the pipeline run isn't quite done, so you turn off your computer and head out to lunch.

Back from lunch, you see that the QA team has reviewed your fix and it looks good, so you merge your changes to the first project. The pipeline run is done, so you open up a comparison of the real prod run's results with your test run. After confirming the new results seem better, you open a PR with your proposed changes and head home.

To some readers, this may sound like a utopian dream. To others, a secret nightmare in which all of the magic and implicit assumptions will inevitably cause a catastrophic break or, worse, subtle bugs missed until it's too late. A lucky few might have some subset of this available in some form in their domain. No one has it all... yet.

The vision outlined here is not impossible. It's not inherently unreliable, or brittle, or limited to a few special use cases. With appropriate use of names and a common system for substitution, we can dramatically reduce manual work, increase efficiency, and ensure correctness in almost any domain where computers are used. Nomia can make it real.

2 Conceptual model and mechanisms

Nomia provides various mechanisms implementing a model[3] that defines several core concepts and their interrelation[4]. In this section, we'll revisit the developer's dream from the introduction, and progressively build up the model and the mechanisms, showing how Nomia can bring that dream to life.

2.1 Resources

In your hypothetical work day, you exploited a number of resources: The source files you edited, the modules you imported, the web service you tested. A **resource** is any system component viewed as an object to be manipulated

[3]Nomia's model is based off of structures borrowed from category theory. No category theory is needed to understand this section, but footnotes will be included for those with the background or interest.

[4]Many of the concepts come together to form a particular kind of traced monoidal 2-category, possibly with some notion of a "universe" object.

and used. Resources in Nomia can be considered very broadly; in addition to traditional resources like files, you could, say, treat the result of some expensive computation as a resource.

In Nomia, resources are manipulated via abstract references called **handles**. Nomia handles are ultimately based on the basic handles provided by the OS: When your browser loaded the test page, under the hood there was an open TCP connection, and when your code checker needed to read in the dependent module interfaces it required a readable file descriptor. A Nomia handle typically has a limited lifetime and in many cases can be passed from process to process or even system to system, in the manner of object capabilities.

Within Nomia, resource handles expose various affordances: A handle to the results of a pipeline run might have a "read" affordance giving you access to the bytes of the output, a handle to the Meld program will have an "exec" affordance to execute it. An **affordance** is the form in which an attribute or capability of a resource that is accessible by the user of that resource is exposed. Different handles might expose different affordances with different semantics based on the resource in question.

Until the day when our OS and hardware are integrated with Nomia, we must eventually translate Nomia-aware handles and references to resources to something the underlying system knows how to work with, such as file descriptors or open TCP connections. An **anomic** handle is one which functions outside of Nomia[5]. For resources which have a corresponding anomic handle type, an anomic handle can be acquired from a Nomia handle. These handles can then be used by Nomia-agnostic components.

Under the hood, handles within Nomia may add extra layers of indirection or by-need evaluation when manipulating resources, such as when your editor only gives you partial code checking while modules are being compiled. An anomic handle, by contrast, must identify a fully realized resource with respect to the system that will operate on it. A resource is said to be **ready to hand** when it is fully materialized in whatever sense is relevant for proper efficient operation outside of Nomia's confines.

During the lifecycle of a resource, many events of interest may occur, such as the resource becoming ready to hand or its contents being updated. Nomia handles allow the user to subscribe to notifications of these events and respond accordingly.

[5]And thus is "lawless" relative to the guarantees Nomia provides. The caller must ensure through other means (such as keeping a Nomia handle itself open) that the relevant preconditions are met.

2.2 Resource types

Resources can be classified by their resource types: the algorithm you modified was a *pipeline component*, the site the QA team evaluated was a *test web service*. A **resource type**[6] is a conceptual identification of many different resources as being the same kind of thing from a certain perspective. A given resource may have many types: To your compiler, a module file is seen as a readable file, whereas to your editor the same file is seen as a read/writeable one.

Each different resource type corresponds to a different Nomia handle type. The handle acquired to a given resource depends both on the specific resource *and* the type of resource the caller is viewing it as.

Resource types identify affordances common to all resources of that type and their semantics: A read/writeable file can be written to and read from, and, assuming no intervening modifications, a read from a certain location will give back the same contents that you last wrote. The **semantics** of an affordance is the meaning ascribed to it; the same affordance may have different semantics from different perspectives. A Nomia handle type therefore defines the set of affordances available and their semantics.

The semantics of an affordance can often be described in terms of resource state: an immutable readable file has its *contents*, some (finite) sequence of bytes, and sequential reads of the file will yield successive portions of those contents. The **resource state** refers to the attributes of the resource that matter to its identity when seen as being of some specific resource type. The resource state in turn can often be modelled mathematically.

Resource types come with a notion of equivalence relative to that type: The pipeline dry-run determined that most of the results of the test run would be the same compute results as those of the prod run that day. Two resources are **equivalent** as instances of some resource type when they are the same for all intents and purposes relevant to the perspective that motivated defining that resource type. Equivalence can often be identified in terms of conditions on the resource state.

Resource types sometimes exist in a supertype/subtype relation: Any immutable readable Unix file can also be seen as a generic Unix filesystem object. One resource type is said to be a **subtype** of another (the **supertype**) when any resource of the subtype can be seen as also being of the

[6]The (generators of the) 0-cells of the category. Note that we do not in general identify a specific resource with some point of the relevant 0-cell, in part because there is no 1:1 mapping between a resource and its type, and in part for reasons detailed in the next section.

supertype. Note that affordance semantics and equivalence are not necessarily preserved across this reinterpretation: On the one hand, Unix filesystem objects in general have no notion of "contents" (such as a socket) and some can't even be read from. On the other, two immutable files with the same contents (thus the same *as immutable files*) may have different inode numbers and be different as general filesystem objects. Handles of a subtype can be converted into handles of the supertype and used by components only aware of the supertype.

Note that resource types can be very domain-specific, and thus Nomia handle types are an "open universe" extensible by the user. They all depend on the relationship of the specific attributes of the resource in question *and* your specific perspective and purpose in using it. Suppose the compute pipeline is written in C++ and your CI system uses gcc for performance, but you prefer clang locally for the better error messages. The object files produced by the two compilers can be quite different, even viewed as object files, and so naïvely the object files compiled by CI after the big merge wouldn't be equivalent to the object files you'd compile locally. But viewed as "object files exporting the right symbols following the right platform ABI based on the relevant headers", they can be considered the same, as long as that perspective meets the needs of your use case.

2.3 Names

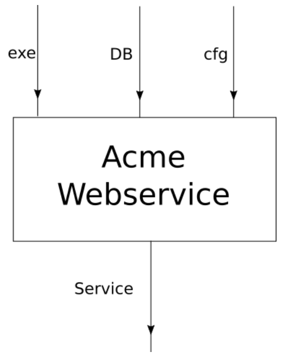
Each of the resources you utilized were first referenced by a name: "Meld" names a particular program, "the test site for PR #XXX" names a particular web service. We might be tempted to think of names as identifying a specific resource, but in general we want to be able to work with names such as "the Acme webservice," which identifies, say, some specific web service *given* some particular executable, a database, and a configuration file. In this broader sense, a **name**[7] is an identifier for some functional relationship between a (possibly empty) sequence[8] of **input** resources by type, and a resulting sequence of **output** resources by type[9]. There is a visual notation

[7]The 1-cells of the category.

[8]Treating the inputs and outputs as a sequence is convenient for understanding the theory, but for practical use the inputs and outputs are named and can have variable multiplicities (e.g. "cat" might be a name with a "single" input that is an arbitrary length list of readable files).

[9]The domain and codomain of the 1-cells. Note that this could in principle be extended to a "dependent category" by allowing the output types to depend on the specific input resources provided. It could similarly be extended to a "codependent category" by allowing the inputs to vary depending on how the outputs are used. There is currently no known

for representing names generally in diagrams, where names are the boxes and inputs/outputs are labelled with their types; "the Acme webservice" described in this paragraph can be visualized as:



Names with an empty list of inputs are therefore called **named resources**[10], since they correspond directly to the resources produced when the name is run.

Nomia provides a syntactic representation of resource names as plain data. It also allows for **resolving** names. If a resolved name has no inputs, handles to its outputs can be acquired from the resolved name. Resource events can be subscribed to from a resolved name. Resolved names can also be passed between components, allowing them to be used without each component needing to be aware of the whole name's structure.

The relationships identified by names must be **deterministic**: input resources which are equivalent will result in equivalent outputs. This may seem to make them too strict to be useful. Recall, however, that equivalence is relative to the resource type, a domain-specific notion; depending on how high-level the notions of equivalence are, there may be quite a bit of leeway in exactly how the desired resources are instantiated.

Sometimes, we still want to use names which identify a specific resource only in the specific context of a user of that name, such as "the standard input stream" (which is a different input stream for different processes) or "today's prod pipeline run" (which is different depending on the day). This isn't feasible with determinism alone. For this case, we also allow **contextual** names, ones whose outputs depend on some aspect of the caller's context,

practical use case for those extensions.

[10] *These* are the points of the relevant 0-cell. Not every resource has a name that fits the requirements of names generally, at least not obviously so, so while every named resource corresponds to some resource the converse isn't true.

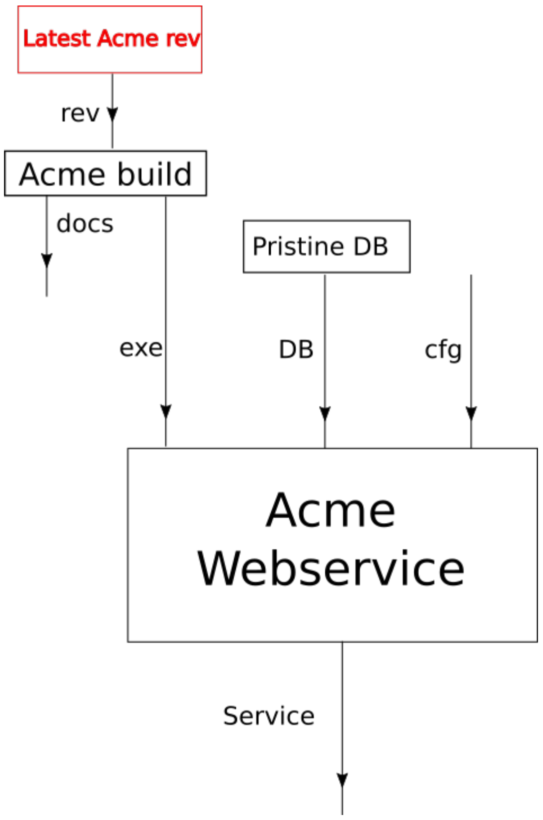
which we model by the name taking a special "context" resource type at the input. "Resources" of this type can be roughly thought of as "the state of the world from some particular perspective"; they are always ultimately instantiated with a (unique) "resource" by the caller from *outside* of the system by passing the relevant resources in at name resolution time. So "the standard input stream" takes in an instantiation of "the state of the world from the perspective of this process" and outputs a readable file stream, and when you resolve that name you must pass in your process's file descriptor table so it can be accessed. Because each top-level instantiation is unique, contextual names are essentially unrestricted with respect to determinism, so long as the lack of determinism can be captured in the context.

Much like with handles, we have anomic *names* that non-Nomia-aware components can use to reference resources, such as normal file paths to file system object resources. These anomic names can be acquired from a resolved name.

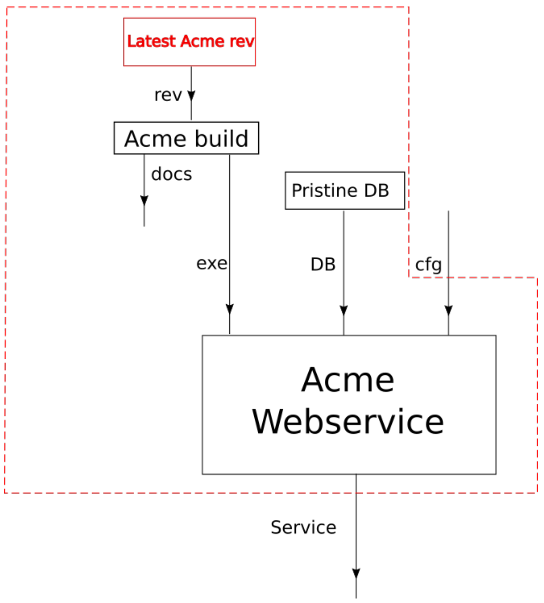
2.4 Substitution

Once we've generalized names to refer to relationships between resources, we may want to substitute the outputs of one name for the inputs of another. **Substitution**[11] is the creation of a new name that relates the inputs of some names to the outputs of others by pairing the outputs of the first with the inputs of the second. We might have a contextual name for "the latest Acme revision", a name to build the Acme source and produce its docs and binaries, and a (non-contextual) name for a pristine Acme database, and compose them all with the "Acme webservice" name to get a name like "the Acme webservice using the executable compiled from the latest code, the pristine test db, and some provided config file". In the visual notation, this would look like:

[11] This is (unbiased) composition of the 1-cells, including tensoring/composing along 0-cells (i.e. projections).



Which as a whole can be seen as new contextual name taking a config file as an input:



Substitutions are exposed via the ability to compose the representations of names together to produce new names, which can themselves be resolved. A name that is not the result of composition is called an **atomic name**.

Names are **referentially transparent**[12] in that we can replace a substitution by "inlining" the result resource rather than referencing it by name, and get the same output (this follows from determinism).

Resource subtyping can be captured in **coercions** (or **upcasts**)—names that map a single input to a single output and are operationally noops. The server compilation process coerced the writable file your editor was using to a readable stream to generate an updated server executable. These coercions can be omitted from the name representation if they can be inferred.

Because of determinism, using names forces us to say exactly what we mean. Domain-specificity of resource types and contextuality *allow* us to say exactly what we mean, and no stricter, especially if the contextual inputs are fine-grained. Together, this gives us **an expressive specification that lets us rely on names and know what to expect** with the resulting resources, **across domains**, modulo implementation bugs. Within one system, we can effectively identify something as broad as "my browser" and something as specific as "Firefox of such-and-such version compiled with this compiler and these configuration flags" and get what we asked for.

Determinism also allows for efficient resource instantiation: If we can

[12]This is "cut elimination" of the underlying multicategory.

cheaply determine that the inputs are all equivalent to some previous instantiation (here or elsewhere), we can safely **reuse the previous result**. And, to the extent that contextuality doesn't tie us to a specific machine, we can safely **distribute the work** to other systems. C programmers may be familiar with ccache, which caches compilation of individual C translation units, and distcc, which allows for distributed computation of C programs; with deterministic names we can get the equivalent for any resource we care to specify! For named resources in particular, since the inputs are always vacuously equivalent, we can aggressively cache and distribute them.

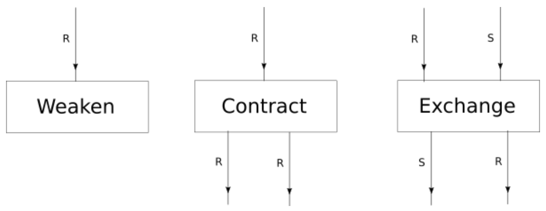
Many names can themselves be cheaply compared for equality by being associated with relatively small byte strings, called their **spellings** — with the semantics that any two names which are spelled the same are the same name. This allows for composed names to be subject to caching without necessarily running intermediate names or even instantiating their results from a cache. If we know that the top-level inputs are equivalent, and each name in the chain is equivalent, then we know the outputs will be equivalent. Spellings typically fall into two categories:

- **Canonical** spellings are short, descriptive character strings. For example, we might have the string `$HOME` spell out a contextual name yielding the caller's home directory.
- **Hashed** spellings are a cryptographic hash of a serialization of (some function of) the data needed to actually run the name. If we substitute some file spelled `foo` into some name that compiles C programs, we might spell the resulting name `sha256("compile-C C11 ${foo}")`. Hashed spellings can omit or transform some of the data from the input to the hash, so long as the name can be considered the same invariantly under that transformation.

One spelling of a non-contextual name can always be determined via a hash of its plain data representation, and other spellings can be accessed from a resolved name.

2.4.1 Technical note: Structural rules

The rules for names given so far technically imply very strict resource management: Every resource must be used, exactly once, in order. There are some cases where this is necessary for correctness. Consider the case where a name depends on three input streams that get instantiated with three pipes, each filled sequentially by the same process. The first pipe must be completely read from in order for the process to start filling the second one, so the process instantiating the name must consume it first, and the data streams can be arbitrarily long so they cannot, in general, be duplicated. In most cases, however, we can relax this through any combination of the following three schemes for **structural names**:



Weakening, which can also be visualized by failing to extend a wire to the output, lets you ignore some resource: the name doesn't do anything with its input. **Contracting**, which can also be visualized by a fork in a wire, lets you duplicate some input: the name copies[13] the resource it's instantiated with and sends one copy over each output. **Exchanging**, which can also be visualized by crossing wires, lets you reorder inputs: the left input wire is forwarded on to the right output wire and vice versa.

These structural names can usually be inferred and thus do not need to be explicitly represented in the name representation.

By default, all inputs and outputs are eligible for all three schemes. On a case-by-case basis, we can conceptually annotate given inputs or outputs with **substructural restrictions**. Marking an output as **relevant** indicates that the result must be used and thus can't be weakened; marking an input as relevant indicates that the name does in fact use that input (e.g. internally it doesn't weaken it anywhere). Marking an output as **affine** indicates that the result can't be copied and thus can't be contracted; marking an input as affine indicates that the name does not duplicate that input. Marking an output as **ordered** indicates that nothing before it can be used once it's used (if ever) and it can't be used once something after has been used and thus can't be exchanged; marking an input as ordered indicates that the name does not reorder resources around that input[14].

In addition to ensuring correctness in rare cases, these annotations can also be used for optimization. If an input is marked relevant, the caller (or general substitution mechanism) might eagerly prepare the resource for

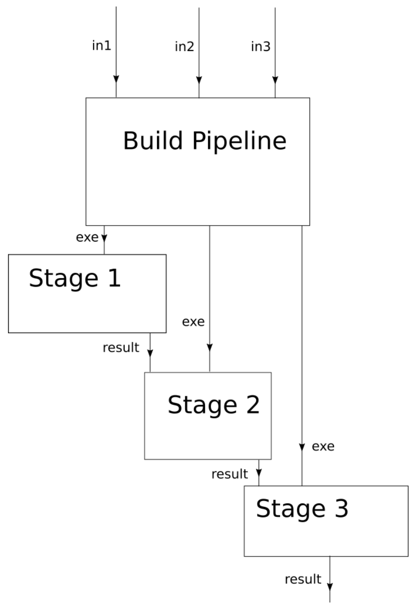
[13]Often by reference!

[14]In principle, we could restrict exchange in only one direction, resulting in a one-way "barrier" to reorders.

consumption (e.g. starting a socket-activated service) rather than waiting for it to be used, since it will be eventually. If an input is marked affine, the caller might garbage collect the resource once it's used. If it's marked ordered, all resources before the input in question can be discarded/preparations stopped once the input is used, and the input itself discarded once something after it is.

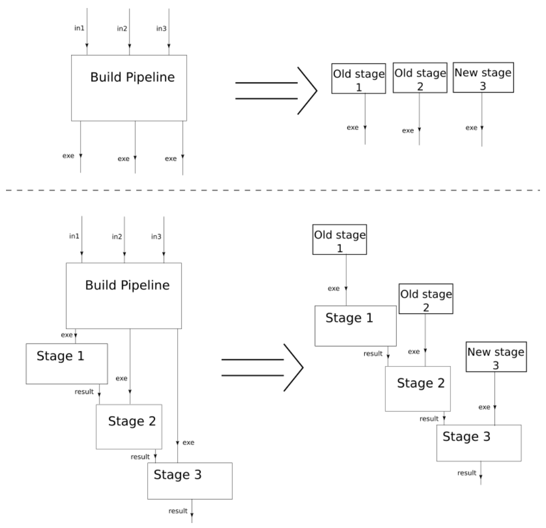
2.5 Reductions

We've already seen how the properties of names allow for efficient resource instantiation and combination. Unfortunately, the efficiency ultimately relies on identifying equivalent inputs, which is not always cheap and is sometimes impossible. Consider the compute pipeline. A "run of the pipeline" might depend on the entire pipeline package and then project out the executable for each stage:



Since you've changed one module in the pipeline, the whole package has changed. If your change only impacts, say, the last stage of the pipeline, the individual stages might be able to recognize that their executables are unchanged. But after the first stage, this recognition wouldn't result in reuse: the first stage may have output cached results, but other stages may not be able to cheaply detect that the output is the same and so would have to rerun.

For these cases, we have **reductions**[15], relationships between *names* in which the reduced-to name is more refined and can stand in for the reduced name to yield an equivalent resource. We might, for example, learn that "build the pipeline" with the modified inputs reduces to a particular set of 3 named executable resources, and then be able to1 infer a reduction of the whole pipeline run:



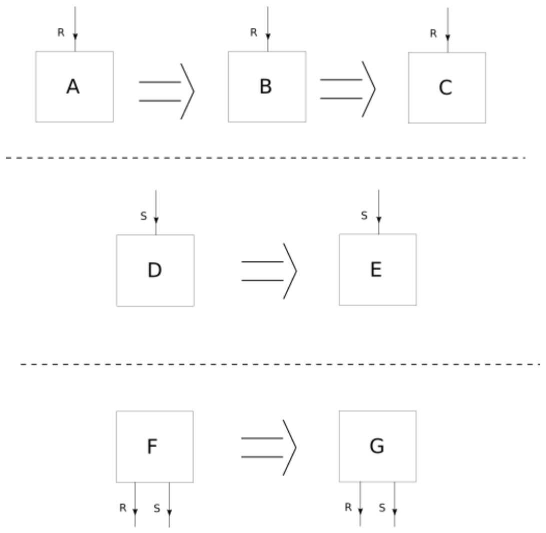
On the right hand side of the inferred reduction, we now know *statically* that the first input into Stage 3 is the same as it was in the previous build! So our caching logic can kick in without running the first 2 stages again.

Reduction events can be subscribed to via a resolved name, allowing the caller to determine if the reduced name is more amenable to its purposes.

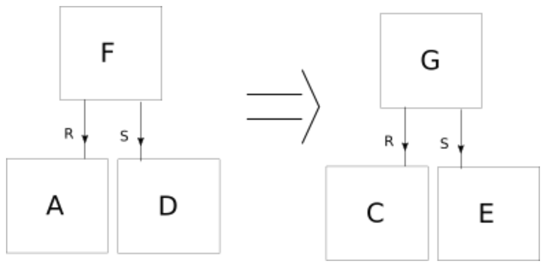
Reductions compose with each other, including across substitutions and projections[16]; they can be thought of as substitutions at the name level. For example, if we have:

[15]The 2-cells. Note that each hom-category is thin for our purposes, i.e., the only relevant 2-dimensional data is whether a reduction exists in a given direction or not.

[16](Unbiased) composition of 2-cells, including vertical, horizontal, and tensoring.



Then we get a composite reduction:



Reductions must preserve determinism: if the original name and the reduced name are passed equivalent inputs, they must yield equivalent outputs. Most reductions are domain-specific, letting you specify how your names relate to other names. Some trivial reductions come automatically:

- Nested substitutions, where a substituted name is substituted into another name, reduces to a substitution where everything is simultaneous[17]
- If a contraction is followed by weakening, it cancels out to a no-op, removing both names.
- A sequence of exchanges that leaves you back where you started cancels out to a no-op.

[17]Thus our 1-composition is lax, not even weak.

Reductions can be determined a priori, just based on the name, or can be identified while the process implementing the named relationship is being run: The process implementing a compilation name might first compile the binary, find the hash of the result, and identify a reduction from the original name to a content-addressed name for the binary. This would allow a case like our pipeline example above.

Reductions can effectively change the input requirements; we can drop, duplicate, or rearrange wires (so long as we respect substructural restrictions[18]). Reductions can also **downcast** an output type into a more specific type, if we know that the resulting resources in the specific cases we've isolated will actually be the right type. Together, these capabilities allow us to flexibly and generically build names that reuse other names for their work, and make that reuse visible to the system as a whole. For example, we could build a TTL cache combinator that takes some name and produces a new name that takes all the same inputs plus the current time and cache state. This either reduces to some cached named resources (ignoring the remaining inputs) if we've run this name recently enough, or reduces to the underlying name with the remaining inputs if we haven't (and captures the result for next time)[19]. Or, all of our names that deal with files could delegate the actual file storage to some content-based names and downcast the results to an appropriate specific kind of file. This allows us to identify two different names that result in a file with the same contents as being the same operationally.

2.6 Namespaces

Implicit in the whole discussion so far is that we are describing an open system: you can freely add new resource types, new names, new reductions, so long as they meet the requirements. Unfortunately, proving or enforcing those requirements is in general infeasible. Therefore, for safety purposes, the system as a whole is conceptually partitioned into multiple **namespaces**, each of which has control over only the names and reductions within it. If one namespace does violate the rules, other namespaces (or users) are only impacted in contexts where they use names from that namespace.

[18]In particular, we can't drop a relevant wire unless we already used the resource before identifying/following the reduction. We can't retain an affine wire unless we haven't used it before identifying/following the reduction, and the evident but verbose rules for ordered wires apply as well.

[19]Note that this could be arbitrarily complex; we could, for instance, have some ML-based "fuzzy matching" on the inputs and an extra model state input, if we have some learned notion of when results are going to be "close enough" based on the input closeness.

Name resolution therefore happens in the scope of some namespaces. A resolved name includes an implicit handle to the namespace that its atomic names have been resolved within, so passing resolved names to other processes can let them access the relevant resources even if they don't otherwise have access to the owning namespace. Thus, namespaces also provide object capability-style access control to resources.

Namespaces are also the locus of caching, including distributed caching and reductions, and spelling. Namespaces can keep previous results in a **store** or **forward** results from another namespace (e.g. on another machine). A namespace can also identify reductions for any of its atomic names at any point in resolving, acquiring handles, or using handles for the resources.

Given a resolved name, a user (or a namespace implementing some other name) may have permissions to **persist** the underlying resource. This persistence can be absolute, lasting until explicitly removed, or it can be conditional on some other resource still existing. This mechanism allows Nomia to efficiently garbage collect unused resources while letting users and resource implementations keep the resources they depend on alive.

In order to have caching/reduction for composite names whose substitutions cross namespace boundaries, we need some way to determine which namespace gets to provide the results or identify the substitutions. When operating on some name, we reduce the name to a fully flattened normal form and work backwards from the final outputs, letting the relevant namespace determine if it knows of a reduction or has a cached result for the whole input graph up to that point at each step.[20] Each namespace is responsible for defining how its atomic names are resolved and how handles are acquired, receiving from Nomia resolved names corresponding to its inputs at the appropriate times.

If one namespace is to use the spelling of names connected to its inputs as part of a caching scheme, that namespace needs some way to get a spelling for a given name that it can trust even if it comes from a different untrusted namespace. We can address this by having namespaces as a resource type and a **namespace of namespaces**. A namespace need not trust all of its peers so long as it trusts some root namespace namespace to give a unique *name* to its peers that it can include in its caching. This can also be used for overlaying optimization or instrumentation; we might have a namespace of namespaces that says "for any name in the namespaces I expose, I'm first

[20]Technically we could safely allow namespaces to reduce based on what comes *after* as well. But until a use case arises, this allows for a much more straightforward and efficient execution algorithm.

going to check this reduction cache I trust to see if it reduces, and only forward on to the underlying namespace if not", which would, among other things, allow different users on the same machine to have their own trusted 3rd party caches without requiring mutual trust. This can also be used to bootstrap the system; much like filenames are usually relative to some ambient root or current directory, most names will be relative to some ambient namespace of namespaces that provides the default set of namespaces for the user or the system.

Namespace resources are implemented as computational components (in-process modules of code or separate processes communicating via inter-process or inter-system communication protocols) that implement all of the relevant mechanisms for the atomic names they control. A handle to a namespace resource is thus a connection to the relevant component, and a namespace of namespaces provides general mechanisms for describing how those components should run.

2.7 Summary of Nomia mechanisms

2.7.1 Names

- Names can be represented as plain data, whether by direct representation of atomic names or by composing existing names into substitutions. Compositions can include resolved names.
- Names can be resolved to resolved names. At resolution time, any contextual inputs must be satisfied. Name resolution happens relative to some root namespace of namespaces.
- The static spelling of a non-contextual name can be determined from the name.

Name syntax The plain-text syntax for names is specified in a formal grammar. Additional domain-specific structured syntaxes, such as a JSON syntax, are expected to be defined as needed.

Informally, most names are of the form:

```
namespace-id?param1=value1&param2="value 2":name-id?param3=value3(
  input: namespace-id-2:name-id-2.output
)
```

Breaking this down in order, we have:

- `namespace-id` is an identifier to locate the namespace the name is in. It is either an identifier string (e.g. `foo`), in which case the location of the namespace is hard-coded or configured into the root namespace of namespaces that the name is resolved in, or it is another name with an output specified in parentheses (e.g. `(foo:bar.namespace)`), in which case the namespace is found by resolving the name and acquiring a handle to the specified output. The `namespace-id` (and the separating `:`) can be omitted, in which case the context in which the name is resolved must specify a default namespace to find names.
- `name-id` is an identifier string whose meaning is supplied by the identified (or default) namespace
- Parameter values are optional, providing more structured data to resolve the specific namespace or name.
- The input specification is optional, identifying any substitutions *into* the identified name. Each input specification can name the input being specified or operate positionally, and references a particular output of some other name.

So this name means: Connect the `output` output of `namespace-id-2:name-id-2` into the `input` input of `namespace-id?param1=value1&param2="value 2":name-id?param3=value3`.

Anywhere outputs are specified, they can instead be omitted, with the calling context selecting a default output name.

To allow for fully arbitrary order-sensitive directed substitution graphs, including cycles[21], there is a more general form of the name syntax:

```
compose
  namespace-id?param1=value1&param2="value 2":name-id?param3=value3(
    input: $name_1.output
  );
  name_1 = namespace-id-2:name-id-2
```

In this form, the name is a sequence of declarations of names with their inputs specified, possibly bound to variable names. These variable names can then be referenced anywhere else within that declaration sequence, including

[21]For example, `glibc` needs to reference a POSIX shell to implement the `system(3)` function, and `bash` needs to reference a C library1. The ideal way to resolve this is to have `glibc` depend on `bash` built against that *same* `glibc`, in a cycle, with infinite recursion avoided because `bash` doesn't need to call `system` in order to load its needed symbols from `glibc`.

beforehand or even within the body of the declaration of the variable in question. This particular example is semantically equivalent to the first, but in general names in this form need not be representable as trees.

As syntax sugar for the case where every name in the substitution except one has outputs feeding, directly or indirectly, into the last one, and the last one has none of its outputs connected anywhere, we have the syntax:

```
let
  name_1 = namespace-id-2:name-id-2
in
  namespace-id:name-id(input: $name_1.output)
```

which is equivalent to:

```
compose
  name_1 = namespace-id-2:name-id-2;
  namespace-id:name-id(input: $name_1.output)
```

Finally, a name can reference a resolved name via a resolved name variable, of the form `@foo`, anywhere a name would otherwise be used. In this case, when the new name is resolved, all resolved name variables must be assigned to a specific resolved name.

2.7.2 Resolved names

- Listeners for events, including name reductions and resource lifecycle events, can be attached to a resolved name.
- Resolved names can be passed to other processes or systems without exposing the underlying namespaces in full.
- Nomia handles to output resources can be acquired from a resolved name with no inputs.
- Anomic names for resources can be acquired from resolved names where relevant. The resources will be ready-to-hand when the anomic name is made available.
- Any spellings of a name can be determined from a resolved name.
- Resolved names can be made persistent, either unconditionally or conditional on the liveness of some other resource.

2.7.3 Handles

- Handles expose affordances for inspecting and manipulating the pointed-to resource.
- Handles can be passed from process to process or system to system where the underlying resource permits it.
- Listeners for resource events can be attached to a handle.

3 Applications

In this section, we'll survey a non-exhaustive list of possible applications of Nomia. Keep in mind that a key feature is that names and substitution can operate across domains, so we should expect synergy between these when multiple domains are implemented!

3.1 Content-addressed storage

Any time we have some resource type defined by its contents and those contents are cheap enough to enumerate, we can build a content-addressed namespace around it. The typical example is immutable files: given any file, we can build a named resource whose contents match that file's at one read-through and whose spelling is a direct hash of the contents. We can also build contextual resources based on handles to the resource in question, e.g. we may have a name `stdin` that takes file descriptor 0 from the context, starts reading through it and saving the file to the store, and when it's done emits a reduction to the named resource corresponding to the file just saved.

There are many many systems implementing content-addressed storage for files, including git's object store and the IPFS distributed file system. These could be reimplemented as Nomia namespaces, or in cases like IPFS, Nomia may reuse its protocols for effective distribution and storage. These systems almost always require you to fully load some resource into the storage before you can fully use it, while with Nomia we can treat as-yet unloaded files the same as already cached ones.

It is expected that many namespaces will have their names reduce to some content-addressed named resource when it's feasible to do so, as this allows sharing of the underlying storage mechanisms and enables reuse when two potentially very different processes result in the same outcome.

3.2 Package management

Fully reproducible efficiently-shared package environments are a core use case of Nomia. The seed of Nomia's design comes from Nix, a system that provides many of the benefits of Nomia specific to the package management domain:

- Nix has content-addressed storage, extended from regular files to the subset of directories that is needed to represent full packages.
- Nix does substitution of compile-time and runtime dependencies by reference, with appropriate reference tracking for resource liveness.
- Nix has a mechanism for serializing package build scripts that captures package dependencies as well as the commands to run, which Nix then hashes to get an identifier for the resulting package.

Together with an isolation mechanism to ensure that nothing unlisted is used, this allows for a package's identifier to correspond exactly to the steps required to produce it from a base set of content-addressed files. Nomia can extend this by having:

- Higher level notions of "package", e.g. a resource type for a "cross-compiled package" that treats as equivalent two packages that use otherwise identical inputs but one is cross-compiled and one is native.
- Multiple namespaces allowing different naming rules and instantiation processes; Nix's are appropriately strict given the need to capture arbitrary package build scripts and ensure determinism, but are overkill and inefficient for many use cases.
- A representation for unsubstituted names with inputs that can be reused in different combinations, allowing for operations like "build that package but with a different compiler version" to be available at the store level.
- Fine-grained contextuality, for cases where full purity is not appropriate.
- Reductions[22], including the so-called "intensional store" and recursive Nix.

[22]Arguably Nix already has reductions in the single case of fixed-output derivations; they (statically) reduce to the fixed output file with the appropriate hash. This allows, for example, for nix-prefetch-url to work without running a derivation.

- Optimizations by Nomia-aware components, such as early use of partially-instantiated packages and more efficient runtime dependency identification.

- Package environments that are themselves first-class resources, enabling higher level operations like "install a package into my user env" to be directly represented in the system.

- Secret files that exist in appropriately restricted namespaces, when building system configurations.

3.3 Unison

Unison is an in-development programming language whose core features can be seen as special cases of Nomia. Unison has immutable content-addressed *expressions*, based on hashing of the language's AST (up to alpha equivalence). This allows for:

- Implicit incremental compilation/evaluation. When Unison needs to evaluate some expression, it can very cheaply determine if it already has, or if it has evaluated some subexpression, and only needs to compile and compute what has changed.

- Exact dependency management within the Unison universe. Any definitions you depend on from some other project are fully content-addressed, with no room for naming conflicts. (Of course, if two parts of your code base use two "versions" of the same type, they won't automatically interconvert.)

- Native distributed computation. Code and computation can be straightforwardly distributed based on the desired compute graph, since we can easily determine if some of the code already exists on a given node or some subset of the computation has already been evaluated. The purity of the language ensures it's safe to combine the results from any node.

- Cheap correct renaming. Human-visible names are simple mappings to the actual underlying content-addressed name that can be easily updated, and in fact different users can have different names for the same expressions without issue.

Nomia can extend this by:

- Combining the language functionalities with package management to give Unison an FFI (foreign function interface) that has the same easy transparent dependency management and preserves Unison's properties.
- Enabling some form of this functionality for arbitrary languages. Without significant work this would have to be restricted to the module level, but it would still allow the implicit recompilation and code distribution for any language.
  - In any context where we can guarantee evaluation is pure (e.g. safe Haskell, or some component we trust promises), we can cache evaluation as well.
- Allowing alternate equivalence classes of expressions. If you update some function to make it more efficient but can prove (or, if trusted, assert) that it has the same behavior, the evaluation cache could use results from either version and older code could be automatically upgraded.

3.4 Service orchestration

By treating services as resources, Nomia can provide an immutable infrastructure-style approach toward service orchestration. Inter-service dependencies can be modelled as substituted inputs to the name representing the final service, which is implemented at resolution by giving one service an open connection to another. If we depend on a service that is the same as one already deployed, we don't need to deploy it again. This shares some properties with Nelson, an orchestration tool that leverages semantic versioning and explicitly configured dependencies to achieve the same outcome in a container-based environment.

3.5 Compute pipelines

By modelling computation results as resources, individual stages as primitive names, and compute graphs as composed names, we can automatically orchestrate arbitrarily complex compute pipelines with safe caching and reuse. The same computation definition can be easily transformed to run locally threaded in-process or across hundreds of machines. We can capture batch processes or system state in contextual inputs that then reduce to non-contextual ones once accessed, thus automatically sharing work without an a priori notion of what has or hasn't changed.

3.6 Continuous integration

A specification for continuous integration can be a name that composes all of the relevant projects together. By combining contextuality and reduction we can capture notions like "the latest version of each dependency" without doing unnecessary new work. Test results can be seen as their own resource and potentially named independently of build products, with parallel computation possible if applicable.

4 Engineering standards

As an aspiring foundational component of nearly every system, it is vital that Nomia be engineered to very high standards. Specific principles include:

- Specification. The system must have clear precise semantics, library interfaces must be fully documented, formats and protocols spelled out in detail. It should be possible based on specifications alone to reimplement any part of the system compatibly, or even the whole.
- Composability. The system must be made up of composable primitives that serve a single semantic purpose and can be combined in arbitrary ways so long as the semantics are respected. Wherever possible this applies even across versions; we do not assume everything running was compiled against the same master codebase. Users should be able to build arbitrary domain-specific systems on top of the core that can all interact. Nomia may include some opinionated "best practice" combinations of components, but cannot assume that those components are always used in that configuration. Nomia provides mechanism, not policy. Nomia provides code for reuse wherever possible.
- Observability. Nomia's users and developers need to be able to understand the behavior and state of the running system, without reinstrumentation or rebuilding. Nomia components can build up and emit rich domain-specific structured event information at every step, which can be sampled and correlated across components to aid in debugging, understand user behavior, identify optimization opportunities, etc.
- Verification. Leveraging as appropriate peer review, testing, fuzzing, formal specification and model checking, formal implementation validation, runtime observation, etc., we want to continually iterate toward ensuring the system is sensibly specified and properly implemented.

- Security. Nomia has security built in from the beginning, with clear boundaries between systems, a model assuming mutually untrusted implementations and users, and applying least privilege throughout. Wherever possible based on the underlying system primitives, Nomia uses object capability-style access control, and where not possible, it is emulated if not prohibitive. In addition to eliminating whole classes of privilege escalation bugs, this makes for a much cleaner programming model when coordinating between many systems.
- Compatibility. Nomia is designed for future enhancements wherever possible, and adheres to strict protocol and API versioning to ensure any backwards incompatibilities that must happen are caught early.
- Portability. The core components should work on most platforms, and cross-platform interaction should work smoothly.

5 Near-term use cases

The long-term vision has Nomia sinking into the background for the user, with all relevant tools having functionality to make them natively Nomia-aware. Your editor, compiler, shell, browser, and application launcher all just understand Nomia names and combine them appropriately. But that's a fairly far-off endpoint. This section lays out three near-term products we're building on top of Nomia at Scarf. Beyond the initial prototype phase, all of these Scarf tools will be built on top of a Scarf-agnostic core Nomia implementation, so alternative uses for and interfaces to Nomia can be built as that core progresses.

5.1 The Scarf Environment Manager

The Scarf Environment Manager is a tool for distributing software and managing per-user and per-project development environments. It will include standard capabilities for adding and removing packages from environments by name (using the full flexibility of Nomia to name packages and environments), declarative environment specifications that can be shared at the appropriate level of specificity to ensure the same environments on multiple machines, and, where possible, automatic integration with domain-specific environment specifications such as `cabal` files or `requirements.txt`.

Package specifications will integrate in with Nix and nixpkgs to leverage the enormous amount of work that has gone into that package set. Eventually we would like to work with the Nix community to have Nix itself built on top of Nomia, with automatic integration in with anything else Nomia-aware, including Scarf's tools.

Package specifications will also likely integrate with the Scarf Gateway, allowing maintainers to host their own package definitions and binaries on third party systems while retaining control over the user access point.

5.2 The Scarf build tool

The Scarf build tool will enable developers to build their projects as Nomia resources, enabling cached builds, distributed builds, and sharing across teams and build modes where relevant. Where possible, the tool will be a drop-in replacement for the relevant build tools that already exist, translating existing CLIs into commands operating on Nomia names, and thereby leveraging the system transparently. As a starting point, we will likely choose a single language to support based on user requirements; Haskell and Rust seem likely candidates.

5.3 The Scarf service manager

The Scarf service manager will be a way to manage services and their interdependencies. While eventually we will likely aim for general deployment management, as a starting point we will focus on local development deployments on a single machine, similar to how Docker Compose is often used. Developers will be able to describe local services for their projects, including any service→service dependencies like databases and service→package dependencies like "postgres depends on the psql binary". The service manager will instantiate them, with any needed builds performed automatically.