Code Staging in GNU Guix

Ludovic Courtès

Code Staging in GNU Guix

Ludovic Courtès
Inria
Bordeaux, France

Abstract

GNU Guix is a "functional" package manager that builds upon earlier work on Nix. Guix implements high-level abstractions such as packages and operating system services as domain-specific languages (DSLs) embedded in Scheme. It also implements build actions and operating system orchestration in Scheme. This leads to a multi-tier programming environment where embedded code snippets are staged for eventual execution.

This paper presents *G-expressions* or "*gexps*", the staging mechanism we devised for Guix. We explain our journey from traditional Lisp S-expressions to G-expressions, which augment the former with contextual information and ensure hygienic code staging. We discuss the implementation of gexps and report on our experience using them in a variety of operating system use cases—from package build processes to system services. Gexps provide a novel way to cover many aspects of OS configuration in a single, multi-tier language, while facilitating code reuse and code sharing.

CCS Concepts • **Software and its engineering → Source code generation**; *Functional languages*; *System administration*;

Keywords Code staging, Scheme, Software deployment

ACM Reference Format:
Ludovic Courtès. 2017. Code Staging in GNU Guix. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3136040.3136045

1 Introduction

Users of free operating systems such as GNU/Linux are used to *package managers* like Debian's `apt-get`, which allow them to install, upgrade, and remove software from a large collection of free software packages. GNU Guix[1] is a *functional* package manager that builds upon the ideas developed for Nix by Dolstra *et al.* [5]. The term "functional" means that software build processes are considered as pure functions: given a set of inputs (compiler, libraries, build scripts, and so on), a package's build function is assumed to always produce the same result. Build results are stored in an immutable persistent data structure, the *store*, implemented as a single directory, `/gnu/store`. Each entry in `/gnu/store` has a file name composed of the hash of all the build inputs used to produce it, followed by a symbolic name. For example, `/gnu/store/yr9rk90jf...-gcc-7.1.0` identifies a specific build of GCC 7.1. A variant of GCC 7.1, for instance one using different build options or different dependencies, would get a different hash. Thus, each store file name uniquely identifies build results, and build processes are *referentially transparent*. This simplifies reasoning on complex package compositions, and also has nice properties such as supporting transactional upgrades and rollback "for free." The Guix System Distribution (or GuixSD) and NixOS extend the functional paradigm to whole operating system deployments [6].

Guix implements this functional deployment paradigm pioneered by Nix but, as explained in previous work, its implementation departs from Nix in interesting ways [3]. First, while Nix relies on a custom domain-specific language (DSL), the Nix language, Guix instead implements a set of DSLs and data structures embedded in the general-purpose language Scheme. This simplifies the development of user interfaces and tools, and allows users to benefit from everything a general-purpose language brings: compiler, debugger, REPL, editor support, libraries, and so on.

```
(define hello
  (package
    (name "hello")
    (version "2.8")
    (source (origin
              (method url-fetch)
              (uri (string-append "mirror://gnu/hello/hello-"
                                  version ".tar.gz"))
              (sha256 (base32 "0wqd8..."))))
    (build-system gnu-build-system)
    (arguments
      '(#:configure-flags
        `("--disable-color"
          ,(string-append "--with-gawk="
                          (assoc-ref %build-inputs "gawk")))))
    (inputs `(("gawk" ,gawk)))
    (synopsis "GNU Hello")
    (description "Example of a GNU package.")
    (home-page "https://gnu.org/software/hello/")
    (license gpl3+)))
```

Figure 1. A package definition using the high-level interface.

[1] `https://gnu.org/software/guix`

In Guix, high-level package definitions like the one shown in Figure 1 are compiled to *derivations*, the low-level representation of build actions inherited from Nix. A derivation specifies: a command to run to perform the build (the *build program*), environment variables to be defined, and derivations whose build result it depends on. Derivations are sent to a privileged daemon, which is responsible for building them on behalf of clients. The build daemon creates isolated environments (*containers* in a chroot) in which it spawns the build program; isolated build environments ensure that build programs do not depend on undeclared inputs.

The second way in which Guix departs from Nix is by using the same language, Scheme, for all its functionality. While package definitions in Nix can embed Bash or Perl snippets to refine build steps, Guix package definitions instead embed Scheme code. Consequently, we have two strata of Scheme code: the *host code*, which provides the package definition, and the *build code*, which is staged for later execution by the build daemon.

This paper focuses on code staging in Guix. Our contribution is twofold: we present G-expressions (or "gexps"), a new code staging mechanism implemented through mere syntactic extensions of the Scheme language; we show the use of gexps in several areas of the "orchestration" programs of the operating system. Section 2 discusses the early attempt at code staging in Guix, as mentioned in [3], and its shortcomings. Section 3 presents the design and implementation of gexps. Section 4 reports on our experience using gexps in a variety of areas in Guix and GuixSD. Section 5 discusses limitations and future work. Finally Section 6 compares gexps to related work and Section 7 concludes.

2 Early Attempt

Scheme is a dialect of Lisp, and Lisp is famous for its its direct representation of code as a data structure using the same syntax. "S-expressions" or "sexps", Lisp's parenthecal expressions, thus look like they lend themselves to code staging. In this section we show how our early experience made it clear that we needed an *augmented* version of sexps.

2.1 Staging Build Expressions

In previous work [3], we presented our first attempt at writing build expressions in Scheme, which relied solely on Lisp quotation [2]. Figure 2 shows an example that creates a derivation that, when built, converts the input image to JPEG, using the `convert` program from the ImageMagick package—this is equivalent to a three-line makefile rule, but referentially transparent. In this example, variable `store` represents the connection to the build daemon. The `package-derivation` function takes the `imagemagick` package object and computes its corresponding derivation, while the `add-to-store` remote procedure call (RPC) instructs the daemon to add

```
(let* ((store (open-connection))
       (drv   (package-derivation store imagemagick))
       (image (add-to-store store "image.png" #t "sha256"
                            "./GuixSD.png"))
       (build
        '(let ((imagemagick (assoc-ref %build-inputs
                                       "imagemagick"))
               (image (assoc-ref %build-inputs "image")))
           (mkdir %output)
           (system* (string-append imagemagick "/bin/convert")
                    "-quality" "75%"
                    image
                    (string-append %output "/image.jpg")))))
  (build-expression->derivation store "example" build
                                #:inputs `(("imagemagick" ,drv)
                                           ("image" ,image))))
```

Figure 2. First attempt: build expressions as sexps.

the file `GuixSD.png` to `/gnu/store`. The variable `build` contains our build program as an sexp (the apostrophe is equivalent to `quote`; it introduces unevaluated code). Finally, `build-expression->derivation` takes the build program and computes the corresponding derivation without building it. The user can then make an RPC to the daemon asking it to build this derivation; only then will the daemon create an isolated environment and run our build program.

`build-expression->derivation` arranges so that the build program, `build`, is evaluated in a context where two extra variables are defined: `%build-inputs` contains a list that maps labels, such as `"imagemagick"`, to file names, such as `/gnu/store/...-imagemagick-6.9`, and `%output` contains the file name for the result of this derivation. Thus, the `assoc-ref` calls in `build` allow it to retrieve the file name of ImageMagick.

2.2 S-Expressions Are Not Enough

Needless to say, this interface was extremely verbose and inconvenient—fortunately, users usually only had to deal with the more pleasant `package` interface shown in Figure 1. All these steps are necessary to define the derivation and its dependencies, but where does the verbosity come from? First, we have to explicitly call `package-derivation` for each package the expression refers to. Second, we have to specify the inputs with labels at the call site. Third, the build code has to use this `assoc-ref` call just to retrieve the `/gnu/store` file name of its inputs. It is error-prone: if we omit the `#:inputs` parameter, of if we mispell an input label, we will only find out when we build the derivation.

Another limitation not visible on a toy example but that became clear as we developed GuixSD is the cost of carrying this `#:inputs` argument down to the call site. It forces programmers to carry not only the build expression, `build`, but also the corresponding `inputs` argument, and makes it very hard to compose build expressions.

While `quote` allowed us to easily represent code, it clearly lacked some of the machinery that would make staging in Guix more convenient. It boils down to two things: it lacks *context*—the set of inputs associated with the expression—and

it lacks the ability to serialize high-level objects—to replace a reference to a package object with its `/gnu/store` file name.

3 G-Expressions

We devised "G-expressions" to address these shortcomings. This section describes the design and implementation of G-expressions, as well as extensions we added to address new use cases.

3.1 Design Principle

```
(let* ((image (local-file "./GuixSD.png" "image.png"))
       (build #~(begin
                  (mkdir #$output)
                  (system* (string-append #$imagemagick
                                          "/bin/convert")
                           "-quality" "75%"
                           #$image
                           (string-append #$output "/image.jpg")))))
  (gexp->derivation "example" build))
```

Figure 3. Creating a derivation from a gexp.

G-expressions *bind software deployment to staging*: when a gexp is staged, the software and artifacts it refers to are guaranteed to be deployed as well. A gexp bundles an sexp and its inputs and outputs, and it can be serialized with `/gnu/store` file names substituted as needed. We first define two operators:

- `gexp`, abbreviated `#~`, is the counterpart of Scheme's `quasiquote`: it allows users to describe unevaluated code.
- `ungexp`, abbreviated `#$`, is the counterpart of Scheme's `unquote`: it allows quoted code to refer to values in the host program. These values can be of any of Scheme's primitive data types, but we are specifically interested in values such as package objects that can be "compiled" to elements in the store.
- `ungexp-splicing`, abbreviated `#$@`, allows a list of elements to be "spliced" in the surrounding list, similar to Scheme's `unquote-splicing`.

The example in Figure 2, rewritten as a gexp, is shown in Figure 3. We have all the properties we were looking for: the gexp carries information about its inputs that does not need to be passed at the `gexp->derivation` call site, and the reference to `imagemagick`, which is bound to a package object, is automatically rewritten to the corresponding store file name by `gexp->derivation`[2]. `local-file` returns a new record that denotes a file from the local file system to be added to the store.

[2]Compared to Figure 2, the `store` argument has disappeared. This is because we implemented `gexp->derivation` as a monadic function in the *state monad*, where the state threaded through monadic function calls is that store parameter. The use of a monadic interface is completely orthogonal to the gexp design though, so we will not insist on it.

Under the hood, `gexp->derivation` converts the gexp to an sexp, the residual build program, and stores it under `/gnu/store`. In doing that, it replaces the `ungexp` forms `#$imagemagick` and `#$image` with their corresponding `/gnu/store` file names. The special `#$output` form, which is used to refer to the output file name of the derivation, is itself replaced by a `getenv` call to retrieve its value at run time (a necessity since the output file name is not known at the time the gexp is serialized to disk.)

The `gexp->derivation` function has an optional `#:system` argument that can be used to specify the system on which the derivation is to be built. For instance, passing `#:system "i686-linux"` forces a 32-bit build on a GNU system running the kernel Linux. The gexp itself is system-independent; it refers to the `imagemagick` package object, which is also system-independent. Since the store file name of derivation outputs is a function of the system type, `gexp->derivation` must make sure to use the file name of ImageMagick corresponding to its `#:system` argument. Therefore, this substitution must happen when `gexp->derivation` is invoked, and *not* when the gexp is created.

G-expressions are "hygienic": *they preserve lexical scope across stages* [9, 13, 15]. Figure 4 illustrates two well-known

```
(let ((gen-body (lambda (x)
                  #~(let ((x 40))
                      (+ x #$x)))))
  #~(let ((x 2))
      #$(gen-body #~x))

⇝ (let ((x-1bd8-0 2))
    (let ((x-4f05-0 40)) (+ x-4f05-0 x-1bd8-0)))
```

Figure 4. Lexical scope preservation across stages (⇝ denotes code generation).

properties of hygienic multi-stage programs: first, binding `x` in one stage (outside the gexp) is distinguished from binding `x` in another stage (inside the gexp); second, binding `x` introduced inside `gen-body` does not shadow binding `x` in the outer gexp thanks to the renaming of these variables in the residual program.

3.2 Implementation

As can be seen from the examples above, gexps are first-class Scheme values: a variable can be bound to a gexp, and gexps can be passed around like any other value. The implementation consists of two parts: a syntactic layer that turns `#~` forms into code that instantiates gexp records, and run-time support functions to serialize gexps and to *lower* their inputs.

Scheme is extensible through macros, and `gexp` is a `syntax-case` macro [7]; `#~` and `#$` are *reader macros* that expand to `gexp` or `ungexp` sexps. This is implemented as a library for GNU Guile, an R5RS/R6RS Scheme implementation, *without any modification to its compiler*. Figure 5 shows what

```
#~(list (string-append #$imagemagick "/bin/convert")
        (string-append #$emacs "/bin/emacs"))

⇒ (gexp (list (string-append (ungexp imagemagick)
                             "/bin/convert")
              (string-append (ungexp emacs)
                             "/bin/emacs")))

⇒ (let ((references
         (list (gexp-input imagemagick)
               (gexp-input emacs)))
        (proc (lambda (a b)
                (list 'list
                      (list 'string-append a
                            "/bin/convert")
                      (list 'string-append b
                            "/bin/emacs")))))
    (make-gexp references proc))

⇝ (list (string-append "/gnu/store/65qrc...-imagemagick-6.9"
                       "/bin/convert")
        (string-append "/gnu/store/825n3...-emacs-25.2"
                       "/bin/emacs"))
```

Figure 5. Macro expansion (⇒) of a G-expression and code generation (⇝).

our `gexp` macro expands to. In the expanded code, `gexp-input` returns a record representing a dependency, while `make-gexp` returns a record representing the whole gexp. The expanded code defines a function of two arguments, `proc`, that returns an sexp; the sexp is the body of the gexp with these two arguments inserted at the point where the original `ungexp` forms appeared. Internally, `gexp->sexp`, the function that converts gexps to sexps, calls this two-argument procedure passing it the store file names of ImageMagick and Emacs. This strategy gives us constant-time substitutions.

The internal `gexp-inputs` function returns, for a given gexp, store, and system type, the derivations that the gexp depends on. In this example, it returns the derivations for ImageMagick and Emacs, as computed by the `package-derivation` function seen earlier. Gexps can be nested, as in `#~#$#~(string-append #$emacs "/bin/emacs")`. The input list returned by `gexp-inputs` for the outermost gexp is the sum of the inputs of the outermost gexp and the inputs nested gexps. Likewise, `gexp-outputs` returns the outputs declared in a gexp and in nested gexps.

The `gexp` macro performs several passes on its body:

1. The first pass *α-renames lexical bindings* introduced by the gexp in order to preserve lexical scope, as illustrated by Figure 4. The implementation is similar to MetaScheme [15] and to that described by Rhiger [13], with caveats discussed in Section 5.
2. The second pass *collects the escape forms* (`ungexp` variants) in the input source. The list of escape forms is needed to construct the list of inputs stored in the gexp record, and to construct the formal argument list of the gexp's code generation function shown in Figure 5.
3. The third pass *substitutes escape forms* with references to the corresponding formal arguments of the code generation function. This leads to the sexp-construction expression shown in Figure 5.

Unlike the examples usually given in the literature, our renaming pass must generate identifiers in a *deterministic* fashion: if they were not, we would produce different derivations at each run, which in turn would trigger full rebuilds of the package graph. Thus, instead of relying on `gensym` and `generate-temporaries`, we generate identifiers as a function of the hash of the input expression and of the lexical nesting level of the identifier—these are the two components we can see in the generated identifiers of Figure 4.

```
(define-gexp-compiler (package-compiler (package <package>)
                                        system target)
  ;; Compile PACKAGE to a derivation for SYSTEM, optionally
  ;; cross-compiled for TARGET.
  (if target
      (package->cross-derivation package target system)
      (package->derivation package system)))

(define-gexp-compiler (local-file-compiler (file <local-file>)
                                           system target)
  ;; "Compile" FILE by adding it to the store.
  (match file
    (($ <local-file> file name)
     (interned-file file name))))
```

Figure 6. The gexp compilers for package objects and for `local-file` objects.

We have seen that gexps often refer to package objects, but we want users to be able to refer to other types of high-level objects, such as the `local-file` object that appears in Figure 5. Therefore, the gexp mechanism can be extended by defining *gexp compilers*. Gexp compilers define, for a given data type, how objects of that type can be lowered to a derivation or to a literal file in the store. Figure 6 shows the compilers for objects of the package and local-file types. All these compilers do is call the (monadic) function that computes the corresponding derivation, in the case of packages, or that simply adds the file to the store in the case of `local-file`. Internally, these compilers are invoked by `gexp->sexp` when it encounters instances of the relevant type in a gexp that is being processed.

Gexp compilers can also have an associated *expander*, which specifies how objects should be "rendered" in the residual sexp. The default expander simply produces the store file name of the derivation output. For example, assuming the variable `emacs` is bound to a package object, `#~(string-append #$emacs "/bin/emacs")` expands to `(string-append "/gnu/store/...-emacs-25.2" "/bin/emacs")`, as we have seen earlier. We defined a `file-append` function that returns objects with a custom expander: one that performs string concatenation when generating the sexp. We can now write gexps like:

```
#~(execl #$(file-append emacs "/bin/emacs"))

⇝ (execl "/gnu/store/...-emacs-25.2/bin/emacs")
```

This is convenient in situations where we do not want or cannot impose a `string-append` call in staged code.

3.3 Extensions

```
(with-imported-modules (source-module-closure
                        '((guix build utils)))
  #~(begin
      ;; Import the module in scope.
      (use-modules (guix build utils))

      ;; Use a function from (guix build utils).
      (mkdir-p #$output)))
```

Figure 7. Specifying imported modules in a gexp.

Modules. One of the reasons for using the same language uniformly is the ability to reuse Guile modules in several contexts. Since builds are performed in an isolated environment, Scheme modules that are needed must be explicitly *imported* in that environment; in other words, the modules must be added as inputs to the derivation that needs them. To that end, gexp objects embed information about the modules they need; the `with-imported-modules` syntactic form allows users to specify modules to import in the gexps that appear in its body. The example in Figure 7 creates a gexp that requires the `(guix build utils)` module and the modules it depends on in its execution environment. The source of these modules is taken from the user's search path and added to the store when `gexp->derivation` is called.

Note that, to actually bring the module in scope, we still need to use Guile's `use-modules` form. We choose to not conflate the notion of modules-in-scope and that of imported-modules because some of the modules come with Guile itself; importing those from the user's environment would make derivations sensitive to the version of Guile used on the "host side".

Cross-compilation. Guix supports cross-compilation to other hardware architectures or operating systems, where packages are built by a *cross-compiler* that generates code for the target platform. To support this, we must be able to distinguish between *native* dependencies—dependencies that run on the build system—and *target* dependencies—dependencies that can only run on the target system. At the level of package definitions, the distinction is made by having two fields: `inputs` and `native-inputs`. To allow gexps to express this distinction, we introduced a new "unquote" form, `ungexp-native`, abbreviated as `#+`.

In a native build context, `#+` is strictly equivalent to `#$`. However, when cross-compiling, elements introduced with `#+` are lowered to their *native* derivation, while elements introduced with `#$` are lowered to a derivation that *cross-compiles* them for the target. To illustrate this, consider the gexp used to convert the bootloader's background image to a suitable format, which resembles that of Figure 3. In a cross-compilation context, the expression that converts the image should use the *native* ImageMagick, not the target ImageMagick, which it would not be able to run anyway. Thus, we write `#+imagemagick` rather than `#$imagemagick`. "Nativeness" propagates to all the values beneath `#+`.

4 Experience

Guix and GuixSD are used in production by individuals and organizations to deploy software on laptops, servers, and clusters. Deploying GuixSD involves staging hundreds of gexps. Introducing a new core mechanism in such a project can be both fruitful and challenging. This section reports on our experience using gexps in Guix.

4.1 Package Build Procedures

As explained earlier, gexps appeared quite recently in the history of Guix. Package definitions like that of Figure 1 rely on the previous ad-hoc staging mechanism, as can be seen in the use of labels in the `inputs` field of definitions. Guix today includes more than 6,000 packages, which still use this old, pre-gexp style. We are considering a migration to a new style but given the size of the repository, this is a challenging task and we must make sure every use case is correctly addressed in the new model.

In theory, labels are no longer needed with gexps since one can now use a `#$` escape to refer to the absolute file name of an input in `arguments`. The indirection that labels introduced had one benefit though: one could create a package variant with a different `inputs` field, and `(assoc-ref %build-inputs ...)` calls in build-side code would automatically resolve to the new dependencies. If we instead allow for direct use of `#$` in package `arguments`, those will be unaffected by changes in `inputs`. It remains to be seen how we can allow `#$` forms while not sacrificing this flexibility.

4.2 System Services

GuixSD, the Guix-based GNU/Linux distribution, was one of the main motivations behind gexps. The principle behind GuixSD is that, given a high-level `operating-system` declaration that specifies user accounts, system services, and other settings, the complete system is instantiated to the store. To achieve that, a lot of files must be generated: start/stop script for all the system services (daemons, operations such as file system mounts, etc.), configuration files, and an initial RAM disk (or *initrd*) for the kernel Linux.

The initrd is a small file system image that the kernel Linux mounts as its initial root file system. It then runs the `/-init` program therein; this program is responsible for mounting the real root file system and for loading any drivers needed to achieve that. If the file system is encrypted, this is also the place where a "mapped device" is set up to decrypt it. On GuixSD, this init program is a Scheme program that

```
(expression->initrd
 (with-imported-modules (source-module-closure
                         '((gnu build linux-boot)))
   #~(begin
       (use-modules (gnu build linux-boot))

       (boot-system #:mounts '#$file-systems
                    #:linux-modules '#$linux-modules
                    #:linux-module-directory '#$kodir)))
```

Figure 8. Creating a derivation that builds an initrd.

we generate based on the OS configuration, using gexps. Figure 8 illustrates the creation of an initrd. Here `expression->initrd` returns a derivation that builds an initrd containing the given gexp as the `/init` program. The staged program in this example calls the `boot-system` function from the `(gnu build linux-boot)` module. The initrd is automatically populated with Guile and its dependencies, the closure of the `(gnu build linux-boot)` module, and the `kodir` store item which contains kernel modules (drivers). That all the relevant store items referred to by the gexp, directly or indirectly, are "pulled" in the initrd comes for free.

Once the root file system is mounted, the initrd passes control to the Shepherd, our daemon-managing daemon[3]. The Shepherd is responsible for starting system services—from the email or SSH daemon to the X graphical display server—and for doing other initialization operations such as mounting additional file systems. The Shepherd is written in Scheme; its "configuration file" is a small Scheme program that instantiates service objects, each of which has a `start` and a `stop` method. In GuixSD, service definitions take the form of a host-side structure with a `start` and `stop` field, both of which are gexps. Those gexps are eventually *spliced* into the Shepherd configuration file.

Since this is all Scheme, and since Guix has a `(gnu build linux-container)` module to create Linux *containers* (isolated execution environments), we were able to reuse this container module within the Shepherd [4]. The only thing we had to do to achieve this was to (1) wrap our `start` gexp in `with-imported-modules` so that it has access to the container functionality, and (2) use our start-process-in-container function lieu of the Shepherd's own start-process function. This is a good example of cross-stage code sharing, where the second stage in this case is the operating system's run-time environment.

4.3 System Tests

GuixSD comes with a set of *whole-system tests*. Each of them takes an `operating-system` definition, which defines the OS configuration, instantiates it in a virtual machine (VM), and verifies that the system running in the VM matches some of the settings. The guest OS is instrumented with a Scheme interpreter that evaluates expressions sent by the host OS—we call it "marionette".

[3] `https://gnu.org/software/shepherd/`

```
#~(begin
    (use-modules (gnu build marionette)
                 (srfi srfi-64) (ice-9 match))

    ;; Spawn the VM that runs the declared OS.
    (define marionette (make-marionette (list #$vm)))

    (test-begin "basic")
    (test-assert "uname"
      (match (marionette-eval '(uname) marionette)
        (#("Linux" host-name version _ architecture)
         (and (string=? host-name
                        #$(operating-system-host-name os))
              (string-prefix? #$(package-version
                                 (operating-system-kernel os))
                              version)
              (string-prefix? architecture %host-type)))))
    (test-end)
    (exit (= (test-runner-fail-count (test-runner-current)) 0)))
```

Figure 9. Core of a whole-system test.

Whole-system tests are derivations whose build programs are gexps like that of Figure 9. The build program passes `vm`, the script to spawn the VM, to the instrumentation tool. The test then uses `marionette-eval` to call the `uname` function in the guest: an *additional code stage* is introduced here, this time using `quote` since gexps are currently limited to contexts with a connection to the build daemon. The test matches the return value of `uname` against the expected vector, and makes sure the information corresponds to the various bits declared in `os`, our OS definition.

5 Limitations

Hygiene. Our implementation of hygiene, discussed in Section 3.2, follows the well-documented approach to the problem [13, 15]. Rhiger's implementation handles a single binding construct (`lambda`) and MetaScheme handles a couple more constructs, but ours has to deal with more binding constructs: R6RS defines around ten binding constructs, and Guile adds a couple more.

Hygiene in multi-stage programs relies on identifying binding constructs. This turns out to be hard to achieve in Scheme because macros can define *new* bindings constructs. Our α-renaming pass is oblivious to those so it will not properly rename bindings introduced by user-defined macros. The macro expander, of course, does this and more already, so it would be tempting to reuse it rather than duplicate part of its work. However, we do not want to macro-expand staged code; instead, macro expansion should be performed "the normal way", by the Guile program that compiles or evaluates the staged code. Again, this ensures reproducibility across Guix installations since we control precisely the Guile variant used in derivations whereas we do not control the Guile variant used to evaluate "host-side" code. How we could hook into Guile's macro expander, based on `psyntax` [7], is still an open question. To our knowledge, this problem of hygienic staging of a language with macros has not been

addressed in literature outside of work on macro expanders [7].

On top of that, `gexp` must track the *quotation level* of several types of quotation: `gexp`, `quote`, `quasiquote`, and `syntax` (though our implementation currently leaves out `syntax` handling). For each quotation type, α-renaming must be skipped when the quotation level is greater than zero. For example, in `#~(lambda (x) '(x ,x))`, the first `x` must *not* be renamed, while the second one must be renamed. Needless to say, the resulting implementation lacks the conciseness of those found in the literature. This is another area that could use help from the macro expander.

Modules in scope. The `with-imported-modules` form allows to specify which modules a gexp expects in its execution environment, but we currently lack a way to specify *which modules should be in scope*, which could be useful in some situations. Part of the reason is that in Guile `use-modules` clauses must appear at the top level, and thus they cannot be used in a gexp that ends up being inserted in a non-top-level position. Macro expanders know the modules in scope at macro-definition points so they can replace free variables in residual code with fully-qualified references to variables inside the modules in scope at the macro definition point. How to achieve something similar with gexp, which lack the big picture that a macro expander has, remains an open question.

Cross-stage debugging. `gexp->derivation` emits build programs as sexps in a file in `/gnu/store`. When an error occurs during the execution of these programs, Guile prints a backtrace that refers to source code locations *inside the generated code*. What we would like, instead, is for the backtrace to refer to the location *of the gexp itself*. C has `#line` directives, which code generators insert in generated code to *map* generated code to its source. If a similar feature was available in Scheme, it would be unsuitable: moving the source code where a gexp appears would lead to a different derivation, in turn triggering a rebuild of everything that depends on it. Instead we would need a way to pass source code mapping information *out-of-band*, in a way that does not affect the derivation that is produced. We are investigating ways to achieve that.

6 Related Work

Like Guix, Nix must be able to include references to store items (derivation results) in generated code while keeping track of derivations this generated code depends on. However, Nix is a single-stage language: the "build side" is left to other languages such as Bash or Perl. Users can splice Nix expressions in strings using *string interpolation* [6]; the interpreter records this dependency in the string context and substitutes the reference with the output file name of the derivation.

Nix views staged code as mere strings and thus does not provide any guarantee on the generated code. The string interpolation syntax (`${...}` sequences) often clashes with the target's language syntax (e.g., Bash uses dollar-brace syntax to reference variables), which can lead to subtle errors. In addition, comments and whitespace in those strings are preserved, and changing those triggers a rebuild of the derivation, which is inconvenient.

GuixSD and MirageOS both aim to unify configuration and deployment into a single high-level language framework [14]. MirageOS uses code staging through MetaOCaml, though that is limited to the implementation of its data storage layer.

Code staging is often studied in the context of optimized code generation [1, 10, 11], or that of hygienic macros [7, 8, 9]. Gexps appear to be the first use of staging in the context of software deployment. Apart from LMS, which relies on types [11], most approaches to staging rely on syntactic annotations similar to `bracket` or `gexp`. Scheme's *hygienic macros*, now part of the R5RS and R6RS standards, as well as MacroML [8] support user-defined binding constructs; the macro expander recognizes those bindings constructs, which allows it to track bindings and preserve hygiene, notably by α-renaming introduced bindings.

MetaScheme is a translation of MetaOCaml's staging primitives, `bracket`, `escape`, and `lift` [15] implemented as a macro that expands to an sexp. It considers only a few core binding constructs and does not address hygiene in the presence of user-defined binding constructs introduced by macros. Rhiger's work [13] follows a similar approach but redefines Scheme's quasiquotation instead of introducing new constructs.

Staged Scheme, or S^2, provides bracket, escape, and lift forms separate from `quasiquote` and `unquote` [10]. As with `syntax-case` [7] and `gexp`, staged code has a disjoint type, as opposed to being a list. S^2's focus is on programs with possibly more than two stages, whereas gexps are, in practice, used for two-stage programs. The article discusses *code regeneration* at run time; gexps have a similar requirement here: at run time a given gexp may be instantiated for different systems, for instance `x86_64-linux` and `i686-linux`.

Hop performs *heterogenous staging*: the source language is Scheme, but the generated code is JavaScript [12]. In Hop `~` introduces staged client-side expressions and `$` escapes to unstaged server-side code. Unlike gexps, support for `~` forms is built in the Hop compiler, and `~` forms are not first-class objects. Hop comes with useful multi-stage debugging facilities not found in Guix, such as the ability to display cross-stage stack traces with correct source location information. It also has a way to express modules in scope for staged code.

7 Conclusion

G-expressions are a novel application of hygienic code staging techniques from the literature to functional software deployment. They extend common staging constructs (`bracket` and `escape`) with additional tooling: cross-compilation-aware escapes, and imported-module annotations. Gexps are used in production to express package build procedures in Guix as well as all the assembly of operating system components in GuixSD. Using a single-language framework with staging has proved to enable new ways of code reuse and composition.

8 References

[1] Baris Aktemur, Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shonan Challenge for Generative Programming: Short Position Paper. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, pp. 147–154, ACM, 2013.

[2] Alan Bawden. Quasiquotation in Lisp. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1999)*, pp. 4–12, 1999.

[3] Ludovic Courtès. Functional Package Management with Guix. In *European Lisp Symposium*, June 2013.

[4] Ludovic Courtès. Running system services in containers. April 2017. *https://gnu.org/s/guix/news/running-system-services-in-containers.html*.

[5] Eelco Dolstra, Merijn de Jonge, Eelco Visser. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of the 18th Large Installation System Administration Conference (LISA '04)*, pp. 79–92, USENIX, November 2004.

[6] Eelco Dolstra, Andres Löh, Nicolas Pierron. NixOS: A Purely Functional Linux Distribution. In *Journal of Functional Programming*, (5-6) , New York, NY, USA, November 2010, pp. 577–615.

[7] R. Kent Dybvig. Writing Hygienic Macros in Scheme with Syntax-Case. Technical Report 356, Indiana Computer Science Department, June 1992.

[8] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros As Multi-stage Computations: Type-safe, Generative, Binding Macros in MacroML. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pp. 74–85, ACM, 2001.

[9] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pp. 151–161, ACM, 1986.

[10] Zhenghao Wang and Richard R. Muntz. Managing Dynamic Changes in Multi-stage Program Generation Systems. In *Proceedings of Generative Programming and Component Engineering (GPCE)*, Springer Berlin Heidelberg, Batory, Don and Consel, Charles and Taha, Walid (editor), pp. 316–334, October 2002.

[11] Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Commun. ACM*, 55(6) , New York, NY, USA, June 2012, pp. 121–130.

[12] Manuel Serrano and Christian Queinnec. A Multi-tier Semantics for Hop. In *Higher Order Symbol. Comput.*,

23(4) , Hingham, MA, USA, November 2010, pp. 409–431.

[13] Morten Rhiger. Hygienic Quasiquotation in Scheme. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, Scheme '12, pp. 58–64, ACM, 2012.

[14] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the Virtual Library Operating System. In *Queue*, 11(11) , New York, NY, USA, December 2013, pp. 30:30–30:44.

[15] Oleg Kiselyov and Chung-chieh Shan. MetaScheme, or untyped MetaOCaml. August 2008. *http://okmij.org/ftp/meta-programming/#meta-scheme.*