Parameterized Packages

Sarthak Shah

April 3, 2023

About me

Contact Details

Name: Sarthak Shah
IRC Nick: cel7t
Github: cel7t
Email: shahsarthakw@gmail.com
Phone: +91 87670 59061
University: BITS Pilani, Hyderabad Campus
Country of Residence: India
Time zone: IST (UTC + 5:30)
Primary Language: English

Background

I am a second year dual degree student studying Mathematics and Electronics & Communications Engineering at BITS Pilani. My programming expertise mainly lies with Scheme, Common Lisp, Clojure and C/C++.

In addition to that, I am working as a system administrator of my university's HPC system.
I am also the president of my college's equivalent of a GNU/Linux advocacy group, through which I have organized and facilitated GNU/Linux installation workshops to introduce my classmates and juniors to free software. I am currently in the process of organizing a workshop on functional programming using Racket.
Thanks to these experiences I have also built a good understanding of build systems and shell scripting, which I believe will be beneficial to this project.

Why GNU Guix?

I've been a Guix user for a few years now, and I believe that its greatest advantage over other functional package managers like Nix is that all its packages are meticulously defined in such a way that they can be built from source wherever needed. Often while running programs in situations such as High-Performance Computing, performance is critical and it is necessary to build packages with certain options to make them run faster and more efficiently. So far Gentoo is the only GNU/Linux distribution offering global build system configuration options (in the form of USE flags); however, if Guix were to get the same ability in the form of parameteric packages it would make it a very good option for any situation where performance is paramount.
I also love using Scheme and the opportunity to work on a large-scale project using Scheme would be very interesting.

Current contributions

I have contributed the following to Guix:

- Two packages: xfishtank and xpenguins
- Bugfix: Replace pumpa origin
- Feature: –with-configure-flag

Project

Abstract

Parameterized packages will provide users a simple way to configure many aspects of packages, à la Gentoo USE flags.
GNU Guix is capable of building all its packages from source, and one great advantage provided by this feature is the ability to specify inclusion or exclusion of optional features in certain packages which will result in greater flexibility and control for the end user. For example, a user running a package on a web server might not require X11 support for it, or a user running Pipewire might want to compile a package with support for it. This project will introduce optional 'package parameters' for all packages in Guix.
Potential Mentors: Ludovic Courtès

Parameterized Packages for Guix

Project Goals

- Providing optional 'parameters' and 'parameter-transforms' values in the package record
- Writing logic to calculate the combinatorial variations created by parameters
- Writing a macro 'let-parameters' to make it easier to write simple parameteric packages
- Writing a package transform for the same in the same vein as –with-patch or –with-source
- Writing documentation and tests for the aforementioned
- Resolving complexities that arise from the addition of these

Project Overview

This project will introduce two extra values in the package record called 'parameters' and 'parameter-transforms'
These will be **completely optional** and if 'parameters' have not been specified for a given package, it will be assumed to depend on the default (unparameteric) version of its dependencies. It will also introduce a macro called 'let-parameters'

How it will work To see how the suggested UI works, see below the example of a hypothetical package 'gui-music-player':

```
(define-public gui-music-player
  (package
    (parameters (and
                 (xor wayland x11^)
                 (xor pulseaudio^ jack* alsa)))
    (parameter-transforms
      ((jack)
       (changes-to-be-made-to-package)))))
```

Alternatively,

```
(define-public gui-music-player
  (package
    (let-parameters (and
                     (xor wayland x11^)
                     (xor pulseaudio^ jack* alsa))
      (p-if x11
        (some-code-for-x11-version))
      (p-case
        (pulseaudio (do-this-for-pulseaudio))
        (jack (do-this-for-jack))
        (alsa (do-this-for-alsa))))))
```

Here, the parameters entry is a list that contains a boolean expression of pre-defined *parameter symbols* like x11, jack etc.
Symbols such as ^, * or ! may be appended to these to indicate the following:

- ^ - the given parameter is *default*; i.e the default version of the package chooses this parameter
  if this is not specified, the first option is assumed to be the default

- * - the given parameter has a non-standard transform, which will be specified in parameter-transforms

- ! - negation

It is **necessary** for any valid list of parameter symbols to evaluate to #t.
The number of combinations for a given parameter list will be computed with the help of an algorithm based on either the Quine-McCluskey Algorithm or Petrick's Method which will be used to simplify the parameter list's boolean expressions.
This will help us avoid combinatorial explosion.
The parameter-transforms will contain procedures for creating package variants based on the flags specified; note that it might be necessary to specify cases such as (and pulseaudio wayland) if the methods for pulseaudio and wayland do not compose.

What if a package record does not contain the parameter value? The package will then be used in its default state by all the packages depending on it. *In general*, parameters propagate to dependencies if they are able to create a valid configuration with the given parameters, and if they do not have the given parameters they are generated in their default state.
This will help in the gradual adoption of parameters, as not every package will have to specify parameters and at the same time the packages specifying parameters will be able to take advantage of them.

One great advantage of this method is that a user could have two packages with conflicting parameters, but they would both work on the system thanks to Guix building both versions of dependencies. This would not work on imperative distributions providing similar functionality.
For example, say gui-music-player (built with pulseaudio) and cli-music-player (built with jack) both depend on mpd, Guix will generate two versions of mpd, one made with the pulseaudio parameter and one made with the jack parameter allowing both to coexist.

Parameter symbols *Fairly generic* options such as x11, gcc or en_US (locale) will be accepted as parameters;
all of them with the exception of locales will be *lowercase* and they, along with their *converses* will have pre-defined transforms for every build system they are valid for.
The vast majority of time in this project is expected to go into writing the main logic and defining a bunch of parameter symbols.
The target would be adding at least 10-20 parameter symbols.

Per-user parameter symbol list An arguably convenient feature would be adding a file ~/.config/guix/parameters.scm that contains a list of parameter symbols for the user that all packages are built with.

Stretch Goals

Adding parameters to a number of packages These will serve as examples for package maintainers and also convenient test-cases for parameter symbols.

Reasons for picking this project

This project will give GNU Guix a competitive edge over not only other functional package managers but also imperative package managers and also give a wide variety of users greater freedom over how they choose to build their packages; a server user might choose not to build packages with x11 or wayland, while an HPC user might choose to build packages with stronger optimization parameters. They would also aid in reducing the size of Guix's packages which are incredibly large compared to the same packages on imperative distributions such as Debian or Artix. Most of all, it would provide potential users one more reason to try Guix!

Timeline

Community-Bonding Period (May 4th - 28th)

Please note that I will have my final exams during this period due to which I have had to reduce the scope of the Goals.

Goals

- Creating a more detailed outline of the project
- Doing some groundwork on package transforms that parameters will build on top of
- Writing a draft of the algorithm that will be used to resolve parameters
- Understanding what kind of parameter symbols Guix users need by interacting with the community

Official Coding Period

Week one (May 29th - June 4th)

I will begin work on package parameters by implementing the parameter list parser and the underlying algorithm for finding valid combinations.

Week two (June 5th - June 11th)

I expect the algorithm writing part of week one's task to continue to week two, and two or three days will be needed to iron out any bugs in the same.

Week three (June 12th - June 18th)

I will work on modifying the package record to include the two optional parameters.

Week four (June 19th - June 25th)

I will work on the let-parameters macro.

Week five (June 26th - July 2nd)

Buffer period, as the tasks in week one and two or unexpected bugs might take more time.

Week six (July 3rd - July 9th)

I will work on writing documentation and tests for all the work done so far.

Midterm evaluations: July 10th

Expected Results:

- A functioning parameter list resolver
- The two additional options added to the package record
- The let-parameters macro
- Documentation and tests for all of the above

Week seven (July 10th - July 16th)

I will work on 'plugging in' transforms; I will add logic that guix will use to transform packages based on a set of given parameters. This includes some 'default transforms' for parameters like x11.

Week eight (July 17th - July 23rd)

Adding more parameter symbols, so that there are at least four or five.

Week nine (July 24th - July 30th)

Writing the first package that supports parameters, possibly Emacs. Ironing out any resulting bugs.

Week ten (July 31st - August 6th)

Writing the –with-parameters transform for package transforms.

Week eleven (August 7th - August 13th)

Writing documentation and tests for all of the work done so far.

Week twelve (August 14th - August 20th)

Writing more parameter symbols and possibly more packages.

Final week (August 21st - August 28th)

Buffer period for any work left or any hard bugs.

Final evaluation

Expected Results (in addition to midterm evaluation results):

- At least ten working parameter symbols with accompanying transforms
- A few packages with parameters defined
- The –with-parameters transform
- Documentation and tests for all of the work done