Ginseng User's Guide

Iulian Neamtiu
neamtiu@cs.umd.edu

January 13, 2008

1 Overview

Ginseng transforms C programs into updateable C programs that can be updated on the fly in a safe manner. The compiler handles two tasks. First, it compiles programs to be dynamically updateable. In particular, programs are compiled so that when dynamically patched with replacement functions, existing code will be able to call the new versions. In addition, when a type is updated, existing values of that type must be transformed to have the new type's representation, in order to be compatible with the new code. Code is compiled to notice when a typed value is out of date, and to then apply the necessary transformation function. Second, the compiler uses a suite of analyses to ensure that updates are always type-safe, even when changes are made to function prototypes or type definitions. The basic idea is to examine the program to discover assumptions made about the types of updateable entities (i.e., functions or data) in the continuation of each program point. These assumptions become constraints on the timing of updates. For a detailed description of the compiler see [2]. **Ginseng** is based on the CIL framework [3], and uses Banshee [1] for constraint solving.

Section 2 lays out the installation procedure for our compiler. Since **Ginseng** works on single-source programs, applications consisting of multiple source files have to first be merged into a single source file. The merging procedure is explained in Section 3. Section 4 describes modifications that have to be made to C programs before handing them over to **Ginseng**. Section 6 covers the safety analysis step of the compiler, along with warning/error messages, how to interpret and how to fix them. Section 7 describes the dynamic patch generator. Section 8 presents a simple yet realistic example of using **Ginseng** to perform on-the-fly software evolution for a linked list.

2 Installation

First, extract the distribution

```
$ tar xzvf ginseng.tar.gz
```

You will need `ocaml 3.08` or later in order to build **Ginseng**. **Ginseng** seems to work with `gcc 3.3`, `gcc 3.4` and `gcc 4.0`, but has been most thoroughly tested with `gcc 3.4`, hence this is the prefered `gcc` version.

It should build out of the box on Linux and Mac. On *BSD, replace `make` with `gmake`. Now you can do the proper build

```
$ cd ginseng/cil

$ ./configure; make
```

3 Merging programs

Ginseng performs *whole-program* analysis and transformation. That is, the program to be analyzed and compiled has to consist of one source file. In most cases applications consist of multiple source files that are compiled indepedently and linked together, so we first need to "merge" all source files into one.

3.1 Prerequisites

Ginseng performs a semantics preserving, source-to-source transformation of C programs, turning them into updateable C programs. Therefore, before compiling with **Ginseng**, we must be sure the original, unmodified application performs as expected. The application must compile using the normal C compiler (usually `gcc`), so the first step in constructing updateable programs is to build the application (usually `make`) and make sure it passes the compiler and linker alright. Ideally, the application comes with regression tests, so make sure it passes the tests as well.

3.2 Merging using CIL

We use the CIL merger `http://manju.cs.berkeley.edu/cil/merger.html` for merging all source files into one single file. Since **Ginseng** is built on top of CIL, the CIL merger is already included in your distribution. The CIL merger can be used as indicated in the guide at the URL above, but we had limited success with that approach. Instead, we use these two simple steps: (1) Invoke

```
$ make CC=<path to DSU compiler>/cil/bin/cilly --trueobj --save-temps --merge
```

on the application; CIL will write preprocessed files into files with the `.o_saved.c` extension. (2) Invoke CIL in merge mode

```
$ CILLY_DONT_COMPILE_AFTER_MERGE=1 <path_to>cil/bin/cilly --merge --keepmerged -o App.o `find
.  -name saved.c`
```

The resulting combined one-source is in `App.o_comb.c`, and it is this file that you will feed to **Ginseng**. The resulting object file is in `App.o_comb.o`; it is used for checking the correctness of merging, see Section 3.3. Note: for projects using `automake`

```
$ make CC='<path_to>/cil/bin/cilly --trueobj --merge --save-temps' CCLD='echo'
```

seems to do a good job at generating the preprocessed `.o_saved.c` files.

If there is an inconsistency between declaration and definition, the CIL merger will fail. For example, if `file1.c` contains a declaration

```
extern int foo;
```

and `file2.c` defines `foo` as

```
long foo;
```

the merger will fail with an error like

```
file2.c:1: Error: Incompatible declaration for foo.
 Previous was at file1.c:1 (different integer types int  and long )}
```

Such inconsistencies must be solved manually; in this case by changing `file1.c` to have a proper declaration for `foo` i.e.,

```
extern long foo;
```

3.3 Checking the merged source

The CIL merger is supposed to produce a single source file equivalent to the original source bundle, but there are no formal guarantees for this. Therefore, before feeding the merged source to **Ginseng**, link the `.o_comb.o` file the same way it would be original sources would be linked, and run the regression tests as indicated in 3.1. This should be fairly trivial, and it normally entails linking the `App.o_comb.o` and running the application test suite on the resulting executable.

4 Preparing sources

After merging, the resulting source file is almost ready to be passed to **Ginseng**. There are a few things the user has to inform **Ginseng** about: where the update points are, which loops ought to be extracted, which functions act like `malloc`. These "notes to the compiler" take the form of `#pragma` directives that the user annotates the source code with, and **Ginseng** in turn will pick them up.

4.1 Specifying update points

We define a *quiescent point* in the program as one at which there are no partially-completed transactions, and all global state is consistent. Dynamic updates are best applied at such quiescent points, and preferably those that are stable throughout a system's lifetime. If the application is structured around an event processing loop, the end of the loop defines a stable quiescent point: there are no pending function calls, little or no data on the stack, and the global state is consistent. Once the user has identified quiescent points in the program flow, an update point can be specified by inserting a call to the update function at that point:

```
...
__DSU_update();
...
```

Establishing an update point in the (infinite) event loop is described in Section 4.2.

4.2 Loop extraction

Ginseng cannot replace code on the stack, but can replace functions (using function indirection). Thus, to replace the body of a loop, we have to ”extract” the loop body in a separate function. This usually comes in handy when the application is structured around an infinite event loop. The user can request loop extraction as indicated in Figure 1; first, the loop to be extracted is labeled, then a `#pragma __DSU_loop("label_name")` is added.

Original program

```
...
  while (1) {
    ...
  }
...
```

Program prepared for loop extraction

```
#pragma __DSU_loop("L")
...
  L:while (1) {
    ...
  }
...
```

Figure 1: Loop extraction.

To specify an update point at the end of the loop, use `#pragma __DSU_loopupd("label")` instead. **Ginseng** will automatically insert a `__DSU_update()` in the extracted loop.

4.3 `Malloc` lookalikes

The safety analysis part of **Ginseng** has to treat `malloc` and other memory allocation functions specially, since these functions are used to construct abstract type values (see Section 4 of [2]). **Ginseng** recognizes `malloc` and `alloca` by default, but sometimes the applications use custom memory allocators, hence the names of allocation functions has to be communicated to the compiler using `#pragma __DSU_malloc("function_name")`. For instance, OpenSSH uses a custom function for memory allocation (`xmalloc`), so the user would have to notify **Ginseng** as follows:

```
#pragma __DSU_malloc("xmalloc")
```

4.4 Overriding the analysis

Since the safety analysis in **Ginseng** is conservative, it might deem types non-updatable, even though the programmer knows the types are used in a safe manner. Classical examples are conversions to and from `void *`. The programmer can choose to override the analysis and “force” types updateable by using the directive:

```
#pragma  __DSU_FORCE_UPDATABLE("type_name")}
```

Conversely, a

```
#pragma  __DSU_FORCE_NONUPDATABLE("type_name")}
```

directive is provided for effectively forcing a type non-updateable; this is useful when the programmer knows the type representation is unlikely to change in future versions, or when type representations are fixed (e.g., hardware-mapped structures). See Section 6 for details.

5 Compiler flags

Flag	Values	Description
Operating modes		

`--do-update`		mandatory option for enabling the **Ginseng**
`--dosemdiff`		compiler operates in *semdiff* mode; takes two `.c` input files and prints out the AST differences
`--dopatchgeneration`		compiler operates in patch generation mode; takes two `.c` input files and, depending on the value of `--type-transformers`, generates auto-patches or type transformer files
`--patchmode`		compiler operates in patch compilation mode; takes a `.patch.c` input files and generates a `.patch.cil.c`
`--testmode`		compiler operates in test mode. Seldom used.
`--printversiondata`		dump the contents of the `.vd` file specified in `--versiondata-in`
Ginsengoptions		
`--update-points`		controls speculative update points insertion
	`full`	insert speculative update points after all statements
	`returnonly`	insert speculative update points only before `return` keywords
	`none`	no speculative update points are inserted
`--effect-printing`		controls inferred effect printing
	`full`	print effects (as source code coments) at each program point
	`functions`	print effects around function signatures only
	`types`	
	`none`	effects are not printed
`--gvar-types`		auto-generate unique named types for each global variable
`--no-static-conabs`		prevent con/abs functions from being `static inline`
`--versiondata-in`		read the version data from this file. Mandatory in patch compilation mode or when `--printversiondata` is pecified
`--versiondata-out`		write the version data to this file. Mandatory in normal and patch compilation modes
Analysis options		
`--assembly`		what to do when abstract types are used in inline assembly
	`warn`	do nothing, just issue a warning
	`ignore`	do nothing, issue no warning
	`kill-shallow`	kill top level types invloved only
	`kill-deep`	kill top level types and their children
	`die`	fail-stop; die at that particular program point
`--from-void`		what to do when `void *` values are cast down to abstract type pointers
	`warn`	do nothing, just issue a warning
	`ignore`	do nothing, issue no warning
	`kill-shallow`	kill top level types invloved only
	`kill-deep`	kill top level types and their children

	`die`	fail-stop; die at that particular program point
Patch generation options		
`--type-transformers`		controls generation of type transformers (applicable in patch mode only)
	`yes`	type transformers are generated along with the auto-generated patch
	`no`	omit type transformer generation
	`exclusive`	emit only type transformers
`--no-patchsplit`		prevent splitting the auto-generated patch file into one-function-files (aka *splinter* files)
`--patchdirectory-out`		directory where the patch file (or splinter files, see `--no-patchsplit` above) will be generated
Debugging options		
`--enable-debug`		turn on printing of debug messages for the specified module. See `updatedebug.ml::bugType` for a list of modules
`--stop-loops`		stop after extracting loops (for debugging loop extraction)
`--no-suspect-alias`		don't perform suspect alias analysis

6 Analysis

Ginseng performs a safety analysis to detect types used in a representation-dependent way that hampers future changes in a type's representation. For example, uses of `sizeof` or unsafe type casts that are legal in the current program version might become illegal in future versions, once the type representation has changed. A type used in an illegal fashion is deemed *non-updateable*; **Ginseng** will not use the type wrapping scheme for such a type, and its representation cannot change in future versions.

The programmer might have to guide **Ginseng**'s safety analysis in certain cases. Since the analysis is monomorphic, it will not detect universal or existential uses of types, rendering certain types non-updateable, although they are used in a type-safe, representation-independent fashion. On the other hand, the analysis might deem a type updateable, but the programmer needs to have a fixed, non-wrapped representation for the type in question.

To override the analysis and *force* a type (non)updateable, **Ginseng** provides two `pragma` primitives — `#pragma __DSU_FORCE_NONUPDATABLE` and `#pragma __DSU_FORCE_UPDATABLE`; their use is detailed in Section 4.4.

Whenever **Ginseng** encounters an "illegal" type use, it prints out an error message in the format

```
(<source file>:<line>) setTypeNonUpdatable(<type name>) (<illegal use>)
```

This points the programmer to the offending source code line; there are cases when changes to the source code eliminate the offending use (e.g. instantiating an existential). When such changes are not effective, the last resort is forcing types (non)updateable.

For example, in updating `sshd` we had to use a `#pragma __DSU_FORCE_UPDATABLE("struct_Channel")` to tell **Ginseng** that an existential use of `struct Channel` is update-safe. Conversely, when updating `vsftpd` we used a `#pragma __DSU_FORCE_NONUPDATABLE("struct_vsf_sysutil_ipv4port")` to prevent **Ginseng** from wrapping `struct vsf_sysutil_ipv4port`.

C lacks support for universal or existential polymorphism, so programmers have to resort to using `void *` for polymorphism. **Ginseng** checks all upcasts to `void *` and downcasts from `void *` to ensure no type "laundering" occurs (see Section 4.3 of [2]). **Ginseng** tracks all upcasts from an abstract type pointer `T *` into `void *` by annotating the `void *` and tracking its subsequent flow. If a `void *` flows to an abstract type pointer `S *`, with $\mathtt{T} \neq \mathtt{S}$, both `S` and `T` are set non-updateable, to avoid representation inconsistencies. Whenever a downcast to `S *` from a `void *` with annotation `T *, U *, V *, etc.` is encountered, the **Ginseng** message has the format:

```
(<source file>:<line>) printVoidConstraints <S> <= <T U V> subset:[true|false]
```

Unless both the left and the right hand side of the subset constraint are identical, singleton sets, all types in the lhs and rhs sets are set non-updateable. Pointers to base types are indicated using the format `__base_type_x`.

7 Dynamic patches

Dynamic patches are generated mostly automatically by **Ginseng**, but (depending on the nature of changes between versions), the programmer might still have to write *type transformers* and *state transformers*. Source code for patches consists of two files: a `.patch.custom.c` file containing state and type transformers, which can be tailored by the programmer, and a `.patch.gen.c` containing definitions of new (or changed) types and functions. **Ginseng** generates both these files automatically, but the programmer is only supposed to alter the former.

7.1 Type transformers

When type representations change, *type transformers* will convert values from the old representation to the new one. **Ginseng** compiler automatically generates type transformer skeletons containing “best guess” conversion functions between representations, but the programmer still has to intervene in order to verify the auto-generated conversions and add initialization code where needed. For instance, if a `struct` type has changed, the stub consists of code to copy the preserved fields over from the old to the new definition, and the programmer will have to initialize newly added fields. Type transformers bear the following signature:

```
void tt_type(type_old *xin, type_new *xout, type *xnew)
{
  ...
}
```

The arguments are pointers to the “raw”, unwrapped, type representations (`xin` and `xout`) and to the wrapped representation (`xnew`); most of the time, `xin` and `xout` are sufficient for writing the conversion function, but when converting linked structures e.g., trees or lists, `xnew` is needed as well. In most cases `type` is a `struct`, and the effort consists of initializing newly added fields. Depending on when/how the new code uses the newly added fields, writing the type transformer can range from trivial (assigning a default value) to impossible (see Section 6.2 of [2]). If no type has changed, the auto-generated `.patch.custom.c` will be empty, meaning there are no type transformers to be filled out. Note however that state transformers (see next section) might still be necessary.

7.2 State transformers

A state transformer is a function run at update time to (1) convert global state in order to establish the invariants the new code expects, and (2) run initialization code the new code depends on, but is not part of the old program’s initialization code.
Since a state transformer function is optional, it is not included by default in the `.patch.custom.c`; the programmer has to add it using the following prototype:

```
void __DSU_state_xform()
{
  ...
}
```

Just like in the type transformer case, state transformer complexity can range from trivial (if at all needed) to impossible (e.g. hardware initialization at boot time). The most complicated cases we have encountered were refactorings of global structures where global state had to be transferred between the old and new storage model.

8 Example

The *linkedlist* example (`ginseng/cil/test/update/linkedlist`) demonstrates dynamic updating with **Ginseng**. To build and run it:

```
$ cd ginseng/cil/test/update/linkedlist; make; ./linkedlist.exe
```

The easisest way to start your own dynamic updating project is to use the *linkedlist* skeleton (`Makefile` and 0/1 directories). Replace `linkedlist-[012].c` with the successive versions of the application to be updated, and adjust the `PROG` variable in the `Makefile` accordingly.

Ginseng supports updating types, functions, global variables, and long-running loops, and the *linkedlist* example illustrates all these.

In `linkedlist-0.c`, a singly-linked list type (`Tlist`) contains elements of type `T`. `main()` runs a loop whose body will be updated twice. `interactive_update()` prints the list and invokes the dynamic updating function `__DSU_update()`; `interactive_update()` will also be updated twice.

In `linkedlist-1.c`, the element type `T` has grown to two fields, `x` and `y`. The transformers for updating $linkedlist_0$ to $linkedlist_1$ are in `0/linkedlist.patch.custom.c`. The `tt_T` type transformer converts values of type `T` from the old (0, $\texttt{T}_{old}$) into the new (1,T_{new}) representation. Global state is transformed in the function `__DSU_state_xform()`. Note that `printT()`, `interactive_update()` and the body of `MAINLOOP` are automatically updated, requiring no user intervention.

In `linkedlist-2.c`, the singly-linked list (`Tlist`) is augmented, making it a doubly-linked list. The transformers for updating $linkedlist_1$ to $linkedlist_2$ are in `1/linkedlist.patch.custom.c`. The `tt_Tlist` type transformer converts values of type `Tlist` from the old (1, $\texttt{Tlist}_{old}$) into the new (2, $\texttt{Tlist}_{new}$) representation. Global state is transformed in the function `__DSU_state_xform()`. Note that the new functions `lastT` and revIterT are automatically rendered accessible to the updated program. As expected, `consT`, `interactive_update()` and the body of `MAINLOOP` are automatically updated.

References

[1] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Proc. SAS*, September 2005. `http://banshee.sourceforge.net/`.

[2] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C (extended version). Technical Report CS-TR-4790, Department of Computer Science, University of Maryland, March 2006. `http://www.cs.umd.edu/projects/dsu/DSU-TR.pdf`.

[3] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science*, 2304:213–228, 2002. `http://manju.cs.berkeley.edu/cil/`.