Introspecting continuations in order to update active code

Jérémy Buisson, Fabien Dagnat

Introspecting continuations in order to update active code

Jérémy Buisson Fabien Dagnat
Institut TELECOM / TELECOM Bretagne
Université européenne de Bretagne
{Jeremy.Buisson,Fabien.Dagnat}@telecom-bretagne.eu

Abstract

In the case of critical systems and dynamic environments, it is necessary to apply bug fixes and functional enhancements at runtime. Mainly due to technical difficulties, updating active code is usually considered impractical. Most of researches on dynamic software update therefore prevent changing active code. In this paper, we study how to express manipulations of the execution state in terms of operations on continuations, thus enabling update of active code. We explore how language support can help doing so in a type-safe manner thanks to specific operators.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; D.2.9 [*Software Engineering*]: Management

General Terms dynamic update of active code, continuation, introspection of execution states, language construct

Keywords DSU, control operator, continuation, typing

1. Introduction

Software evolution and maintenance is continuously required in order to fix bugs and to add new features. In many cases, despite the need for updates, applications cannot be stopped. Updates must therefore occur during software execution. Beyond critical systems, it is also desirable to update from one alternative implementation to another one when applications adapt to dynamic execution contexts. In any case, preserving the consistency of the code that is effectively executed is one of the main challenges. To this end, researchers work on finding the right timing for updates.

In this area, it is commonly assumed that updates can hardly occur while any of the changed pieces of code is active, i.e., being executed. This is what has led to definitions like the one of *quiescent* states (Kramer and Magee 1990). Basically, the idea consists in defining some state properties that ensure that the updated pieces of code are not active nor get activated during updates. The update mechanism can actively bring the execution in a safe state (Kramer and Magee 1990; Polakovic et al. 2007) for instance stopping components in component-based software. A variant (Hoareau and Mahéo 2008) enriches the semantic of (component) client interfaces in order to detect when server components are unavailable, for instance when they are involved in updates. The status is propagated back to clients, therefore ensuring *quiescence*. Alternatively, the mechanism can passively wait and detect when the execution reaches a safe state (Gilmore et al. 1997; Ensink and Adve 2004; Neamtiu et al. 2006; Stoyle et al. 2007) for instance introducing an `update` instruction explicitly put by developers. In specific cases such as parallel programs, actions such as synchronizations can be taken in order to help safe states emerging from the execution (Buisson et al. 2006).

Building on previous results, it has been observed that requiring all of the updated pieces of code to be inactive is too strong. It indeed results in some updates being delayed, possibly for unbounded time. On one side, Java HotSwap (Dmitriev 2001), Proteus (Stoyle et al. 2007) and Erlang (Ericsson AB 2008) allow several versions of classes, modules and functions to coexist in an execution. Activations that exist at the time of the update continue their execution using the old version of the code, even after update completion. The new code is executed either at any next call (Java) or at so-called external calls, i.e., calls that come from outside the module (Erlang). The old code is garbage collected when it cannot be reached anymore. In both cases, the system implicitly assumes that nesting new versions within old version does not break consistency. On the other side, *tranquility* (Vandewoude et al. 2007) and *transactional version consistency* (Neamtiu et al. 2008) relax constraints on safe states. The overall idea of those two relies on atomic transaction interleaving. The software shall behave as if updates were occurring out of any applicative transaction, instead of actually doing so. Intuitively, an update shall not impact both what has been already executed and what has still to be executed in active transactions, hence ensuring atomicity.

In addition, preventing the update of active code makes it almost infeasible to update certain functions such as the `main` one and interactive loops. As in Ginseng (Neamtiu et al. 2006) for instance, common workarounds consist in slicing those pieces of code into functions, which can therefore be updated independently. Nevertheless, updates have to be anticipated and code slicing results in runtime overhead.

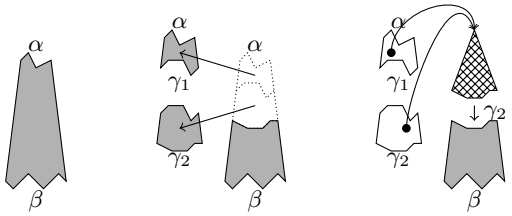
In this paper, we envision the other direction: we consider that active code can be updated consistently. Actually, doing so runs into low-level technical issues such as adjusting instruction pointers, and reshaping and relocating stack frames. Building on previous work on control operators and continuation, we outline how language support could help dealing with those difficulties thanks to continuation manipulation operators. We do not claim that updating active code is made easy. But giving the opportunity of doing so may relax even more the constraints on update timing. It would also allow updates without having sliced the code in anticipation.

The rest of the paper is structured as follows. Section 2 presents an overview of current continuation frameworks. Section 3 depicts the design of operators for the manipulation of continuation objects. Section 4 shows how the outlined operators could be used at the time of updates. Section 5 concludes the paper with a discussion of the proposal.

2. Continuations

At an abstract level, a *continuation* denotes what remains to compute. As such, it captures the state of the execution machine when it is created. Later, this state can be restored (*reinstated*) in order to resume execution. The mechanism generalizes many control operators such as exception dispatching. Concretely, a continuation may be for instance a record containing an instruction pointer, a call stack, local variables and captured environments. While the `call/cc` Scheme function captures the whole continuation, *prompts* (Felleisen 1988) formalize a lower bound that delimits captured continuations, thus giving type and result to reinstating. In the latter case, continuation operators works in triplets for setting a prompt, capturing the delimited continuation and reinstating a continuation. Examples of continuation capturing operators are `shift` (Danvy and Filinski 1990), `cupto` (Gunter et al. 1995) and `withSubCont` (Dybvig et al. 2007). They mainly differ in whether the prompt survives the capture and whether it is saved at the bottom of the continuation.

In most continuation systems, the continuation object is an opaque object. Usually, no accessor or manipulation function is provided except for resuming continuations. Notable exceptions are Smalltalk systems, which provide a programming interface (`ContextPart`, `BlockContext` and `MethodContext` classes) for iterating and modifying stack frames. Nevertheless, Smalltalk has the usual drawbacks of dynamic typing. Strongtalk, a strongly typed Smalltalk system, withdraws such support. The internal representation used by (Dybvig et al. 2007) provides functions to split and append sequences of prompt-delimited stack frames. Yet those facilities are not intended to be used by applications. No introspection facility is provided.



1. capture the continuation | 2. pop stack frames | 3. reload values from popped frames in order to build a replacement

Figure 1. Manipulating stack frames in a type-safe manner.

3. Using continuations for updates

We assume a continuation framework similar to the one of (Dybvig et al. 2007). The `withSubCont` operator captures the continuation up to a prompt and aborts the captured continuation. The `pushSubCont` operator reinstates a continuation on top of the current one with a given expression. Using this framework, the `update` operator is implemented as the following OCaml code.

```
let update v = if update_pending & safe
  then withSubCont root_prompt (fun k ->
         apply_update (); update_tail k v)
  else v
```

In this code, `root_prompt` is a prompt at the root of the execution. It delimits the part of the continuation that can be altered by updates. We assume the prompt is suitably placed.

In normal operations, the `update` operator behaves like the identity function (`else` branch). If an update is pending and the state is safe, the update is applied. The process completes with an *update tail*, which aims at compensating explicitly the update in the flow of execution. The *tail* is specific to each update. In simple cases, it straightforwardly reinstates the continuation, resulting in the regular `update` operator. The *tail* mechanism gives also the opportunity to walk and change the continuation according to the captured state and to the specific update in more complex situations.

In the following, Greek letters denote types. $\alpha \rightarrow \beta$ is the type of functions mapping α parameters to β results. $\beta\ prompt$ is the type of a prompt set at a β expression. $(\alpha, \beta)\ cont$ is the type of a continuation that expects an α value to fill in the captured context, and that produces a β value, i.e., a continuation captured up to a $\beta\ prompt$ prompt.

Walking through the stack frames of a continuation actually means slicing the continuation into subcontinuations. Therefore, a new operator with type $(\alpha, \beta)\ cont \rightarrow (\alpha, \gamma)\ cont \times (\gamma, \beta)\ cont$ shall be introduced in order to

$$\frac{(1)\ \Gamma \vdash e_1 : (\alpha,\beta)\,cont \quad (2)\ \Gamma \vdash \texttt{<label>} : \Gamma_{\texttt{<label>}} \quad (3)\ \Gamma \vdash \Gamma_{\texttt{<label>}}.par <: \alpha \quad (4)\ \Gamma \vdash e_3 : \tau \quad (5)\ \Gamma \cup \Gamma_{\texttt{<label>}}.locals \cup \{i_{hd} : (\Gamma_{\texttt{<label>}}.par, \Gamma_{\texttt{<label>}}.res)\,cont; i_{tl} : (\Gamma_{\texttt{<label>}}.res, \beta)\,cont\} \vdash e_2 : \tau}{\Gamma \vdash \texttt{match\_cont}\ e_1\ \texttt{with (<label> as}\ i_{hd}\texttt{) ::}\ i_{tl}\ \texttt{->}\ e_2\ \texttt{|\ \_ ->}\ e_3 : \tau}$$

Figure 2. One possible typing rule for **match_cont**.

pop stack frames from continuations. If the type γ could be statically determined, then the traversal of continuations could be statically typed. In addition, changing stack frames requires reloading local and environment-captured values from popped frames. Enforcing static typing of such manipulations would help building correct new stack frames as replacements. Figure 1 illustrates those operations. The proposal builds on the following observation.

> If the popped stack frame is one activation of a single function `f`, knowing the call site in `f` that has activated `withSubCont` suffices to determine α, γ and the types of local variables stored in the popped stack frame.

Indeed, α is the return type of the called function; γ is the return type of `f`; local variables are the ones existing at the call site in `f`. In addition, the call site gives an indication of what remains to do in the activation that has been popped, hence an indication of what the update tail has to compute.

For instance, assume the following Fibonacci code where `<L0>` and `<L1>` name the call sites.

```
let rec fib = function
    0 -> 0 | 1 -> 1
  | n -> let fn_1 = <L0> fib (n-1) in
         let fn_2 = <L1> fib (n-2) in
           fn_1+fn_2
```

If a continuation of type $(\alpha,\beta)\,cont$ has `<L1>` as its top level call site, then we can deduce that this continuation assumes an `int` value when reinstated (adding constraint $\alpha = \texttt{int}$); and that the popped frame produces an `int` value (adding constraint $\gamma = \texttt{int}$). We therefore know that the pop operation results in continuations of types $(\texttt{int},\texttt{int})\,cont$ and $(\texttt{int},\beta)\,cont$ for the popped frame and the remainder respectively. In addition, at the `<L1>` call site, the set of existing local and environment variables is $\{\texttt{fib},\texttt{n},\texttt{fn\_1}\}$. This set shapes the stack frame and the types of the variable are known, thus providing sufficient information to reload variables in a type-safe manner. Last, the popped frame captures the **let** `fn_2` instruction and the `fn_1+fn_2` addition. The update tail has still to evaluate these instructions.

The above example outlines the overall idea of the proposed mechanism. It consists in providing a language support in order to match the call site on top of a continuation against the call sites of the code. Hardcoding the mechanism in the language permits to enrich the environment (including the typing environment) of the right-hand sides of match clauses with information saved from the matched call sites. A new kind of expression is introduced in the language for the continuation match and pop operator.

$$\begin{array}{rcl} expr & ::= & \ldots \mid \textbf{match\_cont}\ expr\ \textbf{with}\ clause^* \\ clause & ::= & pattern\ \texttt{->}\ expr \\ pattern & ::= & \_ \mid (label\ \textbf{as}\ ident) :: ident \end{array}$$

In addition, each call site `<label>` is associated with a record $\Gamma_{\texttt{<label>}}$ that holds the α (par field) and γ (res field) type constraints along with the types of variables stored in the stack frame. For example, the following record results from typing `fib`.

$$\Gamma_{\texttt{<L1>}} = \left\{ \begin{array}{l} par = \texttt{int}; res = \texttt{int}; \\ locals = \left[\begin{array}{l} \texttt{fib} : \texttt{int} \to \texttt{int}; \\ \texttt{n} : \texttt{int}; \texttt{fn\_1} : \texttt{int} \end{array} \right] \end{array} \right\}$$

In the right-hand side of each clause, we enforce constraints and type information stored in $\Gamma_{\texttt{<label>}}$ records. Figure 2 shows one possible typing rule for the **match_cont** operator. Premise 1 ensures that the operand is a continuation; premise 2 that patterns are made of call site labels with known $\Gamma_{\texttt{<label>}}$ records; premise 3 that the type of the operand continuation is compatible ($<:$) with the recorded constraints for the call site. The two last premises ensure that the right-hand sides can be typed consistently when the environment of the popped frame (premise 5) is reloaded. The res constraint of the call site is used when binding the sliced continuations in that updated environment.

From the point of view of the implementation, a continuation is a sequence of stack frames. As depicted in Figure 3, each stack frame is made of an instruction pointer upon return (*ipr*) and of the values of variables (*locals*). This structure is in line with existing execution machines such as the ZINC (Leroy 1990) one underlying the OCaml bytecode interpreter. Storage could in addition be added for registers in order to mimic current real processors.

Given that each call site can be identified by a unique return address, the **match_cont** operator can be implemented as comparisons with the *ipr* field on top of the stack frame. Labels are symbolic names used in order to refer conveniently to call site return addresses. They have no existence in the code of the application. When developing updates, the development environment can support developers in identifying relevant call sites, assigning labels and generating update tail skeletons.

Once the matching clause is identified thanks to the *ipr* field, splitting the sequence of stack frames implements continuation slicing. Last, local variables and environment of the popped frame are reloaded from the *locals* field. Holes

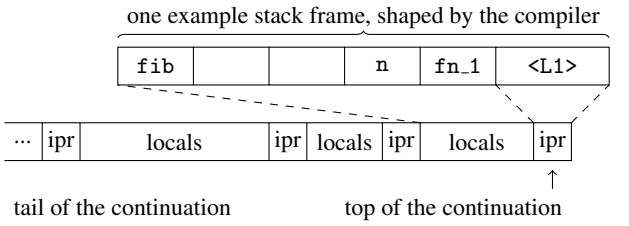


Figure 3. One possible implementation of continuations.

in that field hold temporary variables generated by the compiler. The compiler should therefore save in $\Gamma_{\texttt{<label>}}$ records the positions of values in the stack frame (the shape of the frame) in addition to their types.

4. Using the operator in updates

In the following, we show examples that use the **match_cont** operator in updates. We consider again the above Fibonacci code and assume first that the function has to be updated in order to use arbitrary-precision integers for the result. In the updated code `fib'`, `big_int_of_int` converts OCaml integers into arbitrary-precision ones and `add_big_int` adds two arbitrary-precision integers.

```
let rec fib': int -> big_int = function
    (0 | 1) as n -> big_int_of_int n
  | n -> let fn_1 = fib' (n-1) in
         let fn_2 = fib' (n-2) in
           add_big_int fn_1 fn_2
```

With usual approaches, the application cannot be updated while the `fib` function is active. In the case of Erlang, recursive calls must be internal calls (of the old version). In the context of other approaches, type consistency is violated.

With the approach described in this paper, an update tail gives the opportunity to manipulate the continuation captured at the time of the update. The following listing shows how the update tail can be implemented in a type-safe manner thanks to the **match_cont** operator.

```
let rec update_tail k r =
 match_cont k with
  (<L0> as hd)::tl -> tail_of_L0 tl n r
| (<L1> as hd)::tl -> tail_of_L1 tl fn_1 r
and tail_of_L0 k n r =
  let fn_1 = r in
  let fn_2 = fib' (n-2) in
  let r' = add_big_int
             (big_int_of_int fn1) fn_2
  in match_fib_callers k r'
and tail_of_L1 k fn_1 r =
  let fn_2 = r in
  let r' = add_big_int
               (big_int_of_int fn_1)
               (big_int_of_int fn_2)
  in match_fib_callers k r'
and match_fib_callers k r =
 match_cont k with
  (<L0> as hd)::tl -> tail_of_L0 tl r
| (<L1> as hd)::tl -> tail_of_L1 tl fn_1 r
| _ -> pushSubCont k r
```

In the `update_tail` frontend function, the top stack frame is matched against `<L0>` and `<L1>`. In case `<L1>` is matched, the value of `fn_1` is reloaded from the popped frame; parameter `r` is the result of `fib (n-2)`. Those values are used in order to evaluate `tail_of_L1`, the new tail of the `fib` function starting at `<L1>`. That new tail performs the conversion to arbitrary-precision integers in accordance to the new version of the `fib` function. `tail_of_L0` performs similarly; it uses the new version for the recursive call in the tail. Stack frames are recursively popped by the `match_fib_callers` as long as `<L0>` and `<L1>` are matched, calling back the update tails. In so doing, update tail functions trace back the call graph of the application. Last, when there is evidence that the update has no more impact, the remainder of the continuation is reinstated.

Now assume that the `fib` function is updated to a better linear-time algorithm `fib''`, not changing types. In this case, following options are equally correct: complete activations and do recursive calls with the old version (`tail_continue`); use the new version at any forthcoming call (`tail_immediate`); or cancel activations and restart with the new version (`tail_cancel`). Those update tails are implemented as follows.

```
let tail_continue k r = pushSubCont k r
let tail_immediate k r =
  match_cont k with
    (<L0> as hd)::tl ->
      tail_immediate tl (r+(fib'' (n-2)))
  | (<L1> as hd)::tl ->
      tail_immediate tl (fn_1+r)
  | _ -> pushSubCont k r
let tail_cancel k r =
  match_cont k with
    ((<L0>|<L1>) as hd)::tl ->
      let rec tail_restart k i =
        match_cont k with
            ((<L0>|<L1>) as hd)::tl ->
              tail_restart tl n
          | _ -> pushSubCont k (fib'' i)
      in tail_restart tl n
  | _ -> pushSubCont k r
```

Previous approaches only permits the two first options. In Erlang, that choice is hardwired in the initial application code: `tail_continue` if internal calls are used; `tail_immediate` otherwise. With Java HotSwap, the system imposes `tail_immediate`.

In contrast, the approach described in this paper allows the three options. Furthermore, the choice is made in the update tail, not in the application. Consequently, different choices can be made for different updates. For instance the Java HotSwap semantic can be chosen for the `fib''` update, even if it is incorrect with the `fib'` one.

5. Conclusion

In this paper, we have explored how language support could help updating active code at runtime. Among difficulties, doing so requires to update the execution state, including call stack and instruction pointers. As preliminary results, we describe type-safe manipulation of the execution state that builds on previous work on continuations. In comparison to existing Smalltalk systems, static typing would help detecting incorrect manipulations of execution states.

We have outlined a possible implementation strategy for the proposed **match_cont** operator. The foreseen strategy does not involve any specific code generation scheme. As a result, no anticipation is required by updates. Furthermore, $\Gamma_{\texttt{<label>}}$ records can even be generated afterward, when updates are compiled. Thus high flexibility is provided.

Examples show how the proposed **match_cont** operator can be used in order to update active code. Examples emphasize some of the advantages with regard to other approaches. In addition to allowing updates of active code, the approach avoids requiring anticipating updates in the code of the application e.g., slicing the code into functions. While other approaches usually tangle consistency constraints into the code e.g., explicit external calls of Erlang and transactions in (Neamtiu et al. 2008), our approach extracts this concern in the update itself, in the update tail. As benefit, it allows to adapt consistency constraints to each specific update. The counterpart is that the update tail function implements a traversal of the reversed call graph of the application, leading to high complexity. With traditional approaches, the difficulty merges into the complexity of the application, resulting in apparent ease. Yet, demarcating transactions is not trivial.

Practical implications have to be assessed in order to propose tools, e.g., generation / checking of the reversed call graph traversal. Mutable continuation, i.e., writing values in stack frames, may also be valuable. Continuation implementation has to be adjusted (e.g., cloning stacks or not) according to realistic needs. These ideas are part of our future plans.

Acknowledgments

The work described in this paper has been funded by the French ministry of research through the SPaCIFY consortium (ANR 06 TLOG 27).

References

J. Buisson, F. André, and J-L. Pazat. Afpac: enforcing consistency during the adaptation of a parallel component. *Scalable Computing: Practice and Experience*, 7(3):61–73, September 2006.

O. Danvy and A. Filinski. Abstracting control. In *Conf. on LISP and Functional Programming*, pages 151–160, Nice, France, June 1990. doi: 10.1145/91556.91622.

M. Dmitriev. Safe class and data evolution in large and long-lived java applications. Technical Report TR-2001-98, Sun Microsystems, August 2001.

K. Dybvig, S. Peyton Jones, and A. Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, November 2007. doi: 10.1017/S0956796807006259.

B. Ensink and V. Adve. Coordinating adaptations in distributed systems. In *Int. Conf. on Distributed Computing Systems*, pages 446–455, Tokyo, Japan, March 2004. doi: 10.1109/ICDCS.2004.1281611.

Ericsson AB. *Erlang 5.6.3 Reference manual*, chapter 12. Compilation and code loading. 2008. `http://www.erlang.org/doc/reference_manual/part_frame.html`.

M. Felleisen. The theory and practice of first-class prompts. In *Principles of Programming Languages*, pages 180–190, San Diego, CA, USA, January 1988. doi: 10.1145/73560.73576.

S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-379, University of Edinburgh, December 1997.

C.A. Gunter, D. Rémy, and J.G. Riecke. A generalization of exceptions and control in ML-like languages. In *Int. Conf. on Functional Programming Languages and Computer Architecture*, pages 12–23, La Jolla, CA, USA, June 1995. doi: 10.1145/224164.224173.

D. Hoareau and Y. Mahéo. Middleware support for ubiquitous software components. *Personal and Ubiquitous Computing*, 12 (2):167–178, February 2008. doi: 10.1007/s00779-006-0110-7.

J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990. doi: 10.1109/32.60317.

X. Leroy. The ZINC experiment, an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Programming Language Design and Implementation*, pages 72–83, Ottawa, Ontario, Canada, June 2006. doi: 10.1145/1133981.1133991.

I. Neamtiu, M. Hicks, J. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Principles of Programming Languages*, pages 37–49, San Francisco, CA, USA, January 2008. doi: 10.1145/1328438.1328447.

J. Polakovic, S. Mazare, J-B. Stefani, and P-C. David. Experience with safe dynamic reconfigurations in component-based embedded systems. In *Component-Based Software Engineering*, volume 4608 of *LNCS*, pages 242–257, Medford, MA, USA, July 2007. doi: 10.1007/978-3-540-73551-9_17.

G. Stoyle, M. Hicks, G. Bierman, P. Sewel, and I. Neamtiu. Mutatis Mutandis: safe and flexible dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 29(4), August 2007. doi: 10.1145/1255450.1255455.

Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, December 2007. doi: 10.1109/TSE.2007.70733.