ReCaml: execution state as the cornerstone of reconfigurations

Jérémy Buisson, Fabien Dagnat

ReCaml: Execution State as the Cornerstone of Reconfigurations

Jérémy Buisson
Université Européenne de Bretagne
Écoles de St-Cyr Coëtquidan / VALORIA
Guer, France
jeremy.buisson@st-cyr.terre-net.defense.gouv.fr

Fabien Dagnat
Université Européenne de Bretagne
Institut Télécom / Télécom Bretagne
Plouzané, France
Fabien.Dagnat@telecom-bretagne.eu

Abstract

To fix bugs or to enhance a software system without service disruption, one has to update it dynamically during execution. Most prior dynamic software updating techniques require that the code to be changed is not running at the time of the update. However, this restriction precludes any change to the outermost loops of servers, OS scheduling loops and recursive functions. Permitting a dynamic update to more generally manipulate the program's execution state, including the runtime stack, alleviates this restriction but increases the likelihood of type errors. In this paper we present ReCaml, a language for writing dynamic updates to running programs that views execution state as a delimited continuation. ReCaml includes a novel feature for introspecting continuations called match_cont which is sufficiently powerful to implement a variety of updating policies. We have formalized the core of ReCaml and proved it sound (using the Coq proof assistant), thus ensuring that state-manipulating updates preserve type-safe execution of the updated program. We have implemented ReCaml as an extension to the Caml bytecode interpreter and used it for several examples.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; D.3.4 [*Programming Languages*]: Processors—Compilers; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

General Terms Languages

Keywords dynamic software updating, continuation, functional language, execution state introspection, static typing, Caml

1. Introduction

Stopping a critical and long-running system may not be possible or more simply not acceptable as it would incur an excessive financial or human cost. Dynamic software updating technology addresses this challenge by enabling updates to running software, including bugfixes, feature additions, or even temporary instrumentation for diagnosis or performance tuning [3, 28]. One of the main issues when updating a running software is to ensure safety. After an update, the modified software must remain consistent and continue to achieve its goals. Final results must not be compromised even if intermediate results are reused in a different context. The Gmail outage in February 2009 [11] has shown possible consequences of unsafe updates: an update of the data placement service inconsistent with the redundancy strategy has caused a global denial of service.

Much prior work on dynamic software updating has observed that forms of safety (such as type safety) can be ensured by restricting updates to active system components [1, 9, 19, 22, 36]. For example, if an update attempts to fix a bug in function `foo`, then the update may be rejected if `foo` happens to be on the call stack. Baumann et al. [7] and Arnold and Kaashoek [4] report that for an OS kernel, up to 80% to 90% of the security fixes are supported by this approach. However, it happens that a function that never becomes passive, potentially in critical parts of the software system, needs to be updated. Not being able to update actively running functions prevents for instance updating the outermost loop of a server. Extracting loop bodies into separate functions [28] makes the code briefly inactive between each iteration. However, this technique does not solve any of the following additional cases. The primary Linux scheduler function is never passive as it is on the stack of all threads [4]. Baumann et al. [7] also mention exception handlers in a kernel which may need update at runtime [26]. The use of some compilers that squash software structure makes the situation even worse. For example, synchronous languages, used to program embedded systems, rely on compilers [2] that interleave instructions coming from many components and depending on the same input data into a single block of code. The compiled software structure thus causes what were once independent source-code units to be considered active when any one of them is.

In order to support more updates, Hofmeister and Purtilo [20] have proposed to focus on the execution state rather than the program structure. Upon update, the runtime stack is captured, adjusted then restored. Because the stack is appropriately handled, it does not matter if some of the updated functions are actively running. However, this approach has currently no formal semantics and provides no guarantee that update developers will not produce type-incorrect states.

This paper places the execution state approach [10, 20, 25] on safer ground by defining ReCaml, a functional language designed for manipulating execution states in a safe manner. We have defined ReCaml formally and proved it sound. Viewing the execution state as a delimited continuation [15], updating a computation consists in capturing, modifying and reinstating a continuation. To support the modification of a continuation, we define a new "match_cont" pattern-matching operator. It matches a continuation with call sites

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

to decompose it in stack frames peforming specific update actions on each of them. Depending on the execution state, the update programmer specifies the action to apply, e.g., discarding a frame, modifying a frame or keeping it unmodified. Combining such actions, the approach is flexible enough to support many policies, such as completing the computation at the old version, combining old results with subsequent new computation, or discarding old results for recomputing entirely at the new version. Attaching types to call sites allows us to check that the "match_cont" operator is well typed, and therefore that stack introspection is correct. The main contributions of our work are:

- *Explicit execution state management.* Updates are expressed as manipulations of the execution states. The work of update developers focuses mainly on this aspect, which we call *compensation*. In doing so, a developer can implement resulting deterministic behaviors by explicitly controlling the operations executed by the update depending on its timing.
- *Optimistic update.* As a consequence of the previous point, updates can occur at any time. A compensation ensures consistency afterwards, according to the execution state at the time of the update. Therefore, no preventive action (such as waiting for elements of the software to become inactive) is required. In addition, even if updates might not be effective immediately, they are executed with no delay.
- *DSU as manipulation of delimited continuations.* While continuations are common when studying languages and modelling exceptions and coroutines, they have not before been used for dynamic software updating. Relying on continuations, ReCaml does not require any source code transformation or any specific compilation scheme. DSU as manipulation of continuations fits nicely within a functional framework.
- *Formal semantics and static type system.* ReCaml comes with operators for capturing, modifying and reinstating continuations. It is equipped with a formal operational semantics. Although it is aimed at manipulating execution states, which are dynamic structures, ReCaml, and especially the continuation manipulation, is statically typed. The type system is proved to be sound using the Coq theorem prover.
- *Working prototype.* We have developed a prototype of ReCaml, which we have used to implement a few examples.

In Section 2 we first present concrete strategies based on our approach. Section 3 outlines our approach. Section 4 describes in details ReCaml, the formal language underlying our approach. Section 5 discusses implementation issues.

2. Update Complexity vs Application Simplicity

In this section, our aim is to convince the reader that updates can be so complex that the search for sophisticated solutions is justified. We are aware that supporting tools will be required in order to ease the proposed solution. We leave this problem to subsequent work, beyond the scope of this paper.

Our argumentation relies on a program computing a Fibonacci number. This very simple toy example is just aimed as a proof of concept to illustrate the difficulties when updating a program which is repeatedly active at the time of the update. If updates are already complex for such a simple program, then it should be worse for real applications. The initial version of our example is:

```
let rec fib n =
  if n < 2 then n
  else (fib (n−1)) + (fib (n−2))
in fib 12345
```

2.1 Initial Remarks and Overall Approach

There is no point in splitting this code in finer structural elements[1]. This program is built around a single recursive function, whose outermost execution completes only when the whole program terminates. Hence trying to passivate the `fib` function makes no sense. If old and new versions can be mixed, dynamic rebinding [12, 14] obviously solves the problem: active calls complete with the old version while new calls can be directed to the new version. Usually, this assumption implies that the type of the rebound function does not change. If the type of the `fib` function is changed, then rebinding it breaks consistency.

An update has therefore to deal with the current execution state. It corresponds to the stack of calls already started with their arguments. Such ongoing calls are called *activations* in the rest of the paper. Updating a function requires to specify the action to handle each activation. Such specifications are called *compensations*[2]. For example, updating a function f of type $\tau_1 \rightarrow \tau_2$ while changing its type to $\tau_1' \rightarrow \tau_2'$ may require to convert its argument to its new type (τ_1') or its result to be used by code expecting values of the old type (τ_2). More generally, a compensation can:

- yield to the activation, hence executing the old version until the completion of the activation. The result may need to be converted to conform to the new type of its calling activation if it has changed. Note that this is the semantics of Erlang [14], Java HotSwap [12] and more generally of dynamic rebinding, where result conversion is the identity function.
- cancel the activation, hence starting over the call with the new version. Call parameters shall be converted according to the new version. The result shall also be converted according to how the compensation handles its calling activation.
- extract intermediate results from the activation in order to feed some custom code. Depending on how the calling activation is compensated, this custom code computes the new result in place of the canceled activation.

The relative worth of each strategy depends on the time at which the update occurs. For example, if the considered activation is close to its completion, then it may be worthwhile to let it complete its execution. If the activation has started recently, then it may be better to abort and start over. If the update occurs in the middle of the execution period, then the third option could be more appropriate.

In the third option, the amount of reusable intermediate results varies depending on the old and new versions. The extreme case where no intermediate result can be reused matches the second option, i.e., aborting the activation and starting over the call. The quantity of reusable results gives an additional hint in order to choose the most advantageous option.

2.2 Replacing the Type of Integers

We first emphasize problems arising when modifying a type. As the computed Fibonacci number becomes high, using fixed-size integers will result in an overflow. Instead, it is safer to use arbitrary precision integers. The new version of the program is[3]:

[1] Except possibly abstracting arithmetic operations in the integer data type. Here, the abstract data type is implicit as Haskell's `Num` type class.

[2] Makris and Bazzi [25] use the name *stack/continuation transformer* and Gupta et al. [18] use *state mapping*. Being functional, ReCaml does not allow in place modification of a continuation but favors the construction of a new future. Hence, we prefer a new name to avoid misunderstanding.

[3] In Caml libraries, `num_of_int` is the function that converts an integer to arbitrary precision; `+/` is the addition over arbitrary precision integers.

```
let rec fib n =
  if n < 2 then num_of_int n
  else (fib (n−1)) +/ (fib (n−2))
```

Obviously, using dynamic rebinding forbids this update as the type of `fib` is changed and there is at least one active call. Assuming that the integer data type has been well abstracted, one possible strategy could consist in updating this data type, like Gilmore et al. [16] and Neamtiu et al. [28] do. This approach has two major drawbacks. First, it updates all the uses of integers, while we want that only the result of the `fib` function has the overhead of arbitrary precision integers. Second, at the time of the update, some of the executions of the `fib` function might have already produced overflowed integers. A systematic update of all integers has no chance to distinguish the overflowed values that must be recomputed.

One possible update is as follows. Given an activation, if none of the recursive calls has been evaluated, then the activation can start over with the new version of the function. Otherwise, the compensation checks intermediate results in order to detect whether an overflow has occurred. Only non-overflowed results are converted to the new type. Overflowed or missing results are computed using the new version. Last, the compensation uses the arbitrary precision operator in order to perform the addition. The compensation handles caller activations in a similar way, taking into account the fact that the type of the call result has already been converted. The code of this update is outlined in Section 3 and detailed in Section 6 to illustrate ReCaml.

2.3 Introducing Memoization

Second, we emphasize difficulties that occur when changing the algorithmic structure. In our example, there is a well-known algorithm with linear time complexity, while the initial one has exponential time complexity. The new version of the program is[4]:

```
let rec fib' n i fi fi1 =
  if i=n then fi
  else fib' n (i+1) (fi +/ fi1) fi
in let fib n =
  if n < 2 then num_of_int n
  else fib' n 2 1/ 1/
```

We can safely mix new and old versions and rebind dynamically the name `fib` as the type of the function is not changed. However, in this case, the effective behavior still has polynomial time complexity. Indeed, in the worst case, there is a stack of `n` activations of the old function, each of which subsequently performs up to one call to the new version. The effective behavior is worse than aborting and starting over the program, which is not satisfactory.

A better way to perform this update is to look out for two consecutive Fibonacci numbers in intermediate results. The new version is evaluated from the greatest pair, passed as parameters to the `fib'` function. If there is no such pair, it is not worth reusing any intermediate result and the program would rather start over.

2.4 Discussion

Using these two simple examples, we aim at showing that updating a software at runtime and in the right way is a difficult task. There is no general scheme that applies well to all of the cases. In the first case (Section 2.2), each activation is converted independently of the others to the new version. In the second case (Section 2.3), as the algorithm changes radically, all of the activations are cancelled and there is a lookup for specific intermediate results. These update schemes are complex despite the simplicity of the application.

[4] To keep the program simple, we extend Caml with `1/` to denote the arbitrary precision literal 1 similarly to the `+/` notation for arbitrary precision addition.

In addition, our examples show that even for a single application, the right scheme depends on the update itself. This is the reason why we argue in favor of a mechanism that allows developers to design specific schemes for each update. This approach would not prevent proposing some update schemes "off-the-shelf", e.g., relying on some tools such as code generators, thus avoiding burdening developers when possible. Makris and Bazzi [25] for instance have already proposed such automatic generation strategies.

3. Overview of the Approach

In the above examples, the key mechanism is the ability to introspect activations when updating. Updates of Section 2 require intermediate results from activations. They also need to identify what has been done and what has still to be evaluated in each activation. For the implementer, this means that we need a mechanism to reify the state of the execution, including the call stack. To achieve this, we use continuations to model activations and we propose a new pattern matching operator `match_cont`, abbreviated as `mc`. Given a continuation, it matches the return address of the top stack frame as an indication of what remains to be done in the activation. It pops this stack frame and picks values from it in order to retrieve intermediate results. To do this, we extend the semantics with low-level details of the dynamics of the runtime stack.

In the following, we give an overview of how this operator helps in the `fib` example (Section 2.2). Here we give only part of it to make it easier to comment and understand. Section 6 gives more details and the full source code is in Figure 7.

The version below of the `fib` function is annotated for the purpose of update. Call sites' labels may be given by the update developer or generated by some assisting tool. The labelling strategy is not discussed here because it is beyond the scope of this paper.

```
let rec fib n =
  if n < 2 then n
  else (let fn1 = <L1> fib (n−1) in
        let fn2 = <L2> fib (n−2) in
          fn1+fn2)
in <Lroot> fib 12345
```

Using these labels, the update developer can write a function that chooses the most appropriate strategy for each activation of `fib` depending on the point it has reached.

The main function compensating the effect of the update from `int` to `num` is given below. At each step, this function `match_fib_callers_` proceeds by finding what is the state of the activation at the top of the current continuation (`k`) using `match_cont`. The second parameter (`r`) is the result value that would have been used to return to the top stack frame.

```
let rec match_fib_callers_ k r =
 match_cont k with
  | <L1:n> :: k' →        (* (1) complete with new version *)
  | <L2:n fn1> :: k' →                  (* (2) convert fn1 *)
    let nfn1 = if (n−1)>44 then fib (n−1)
               else num_of_int nfn1 in
    let r' = (fn1' +/ r) in
      match_fib_callers_ k' r'
  | <Lroot> :: _ →         (* (3) resume normal execution *)
```

Notice that when filtering a case the update developer can specify values that he wants to extract from the current activation. For example, in case (1), he may use the rank of the Fibonacci number being calculated (here it is bound to `n`) and in case (2), he may also access the intermediate result of `fib (n-1)` named here `fn1`.

As described in Section 2.2, when the top stack frame matches `L2`, the compensation has first to check whether `fib (n−1)` has overflowed. Assuming that integers are coded by, e.g., 31-bits signed integers, we statically know that the biggest correct (smaller than $2^{30}-1$) Fibonacci number has rank 44. So the compensation compares the rank `n-1` (where `n` is picked from the stack frame on top of the continuation `k`) to 44 in order to decide wether `fn1` can be reused. We assume here that `r` has already been handled appropriately by the compensation, hence its type is `num`. See Section 6 for details on how it switches from `int` to `num`. Then the compensation completes the popped activation in `r'`. Last, we have to compensate the tail `k'` of the continuation. Because the next stack frame is also suspended at a call of `fib` (`L2` originates from `fib`), we have to check once again for the callers of `fib`. Hence the tail `k'` is compensated by a recursive call of `match_fib_callers_`.

4. The ReCaml Language

Building on the λ-calculus, ReCaml adds a model of stack frames, which are generated by the compiler. On top of this model and of a continuation framework, it implements the `mc` operator. In doing so, developers programming updates in ReCaml can manipulate runtime states using the same language. Embedding the operator in the language allows us to extend the type system in order to eliminate statically unsound update programs.

Triggering and executing an update is the responsibility of the execution platform. It is done by some kind of interrupt that can preempt execution at any time. However, updates must deal on their own with their timing with respect to the application execution. The execution platform captures the execution state and passes it as an argument to the update. In return, updates have to guess when the execution has been preempted to select appropriate actions. To mitigate the issue in bootstraping the compensation and to align continuation extremities on stack frame boundaries, as a first implementation, we check for the trigger only when the execution control returns to a caller. This restriction is equivalent to explicit update points. The application developer can cause additional points thanks to dummy calls, each of which incurs a return.

4.1 Syntax

We first describe the syntactical constructs and notations (Figure 1) then we discuss the choices in the design of the grammar.

4.1.1 Description of the grammar

Because we use an environment-based semantics, we need explicit closures and environment management. While $\lambda x.e$ is the usual abstraction construct, $(\lambda x.e, \mathcal{E})$ denotes a closure such that the captured environment $\mathcal{E}$ is used to evaluate the body of the function upon application. The syntax of the application operator $({<}l{>}\ e\ e)$ is extended with a label <*l*> that names the call site. The $(\mathrm{env}_{\mathcal{E}}\ e)$ operator evaluates its subterm e in the environment $\mathcal{E}$ instead of the current evaluation environment. Recursive functions are defined as usual $(\text{let rec } x = \lambda x.e \text{ in } e)$.

Our continuation framework defines first-class instantiable prompts and first-class delimited continuations. Intuitively, prompts are delimiters that bound the outermost context that shall be captured within a continuation. Hence a delimited continuation represents only part of the remainder of execution. The newprompt operator instantiates a fresh prompt. The $(\mathrm{setprompt}_{<l>}\ e\ e)$ operator inserts a delimiter in the evaluation context. Given a prompt, the $(\mathrm{capture}_{<l>} \text{up to } e \text{ with } e)$ operator captures and replaces the current continuation up to the innermost delimiter. The continuation is wrapped by the $\mathrm{cont}\,(k)$ constructor. The $(\mathrm{reinstate}_{<l>}\ e\ e)$ operator reinstates and evaluates a continuation. We shall explain later in Section 4.1.2 the $\left(\mathrm{cap}_{<l>,\mathcal{E}} \text{up to } p \text{ with } v\right)$ operator, which is an explicit intermediate step in the capture of a continuation.

In order to model the state structure, we introduce an operator $(\mathrm{frame}_{<l>,\mathcal{E},p'}\, e)$, which annotates activation boundaries. The operator denotes that e is evaluated in a new stack frame that results from the call/return site <*l*>. At the boundary, a prompt is possibly set if the third annotation p' is not $\perp$ (i.e., it is the name of a prompt). $\mathcal{E}$ recalls the evaluation environment of the enclosing context of the operator thus keeping track of the values accessible in this frame.

In the following, v is a value; e denotes a term; x is a variable; k is a continuation, i.e., an evaluation context; p denotes a prompt; <*l*> names a call site; and $\mathcal{E}$ is an environment.

$$
\begin{array}{rcl}
v & ::= & (\lambda x.e, \mathcal{E}) \;\mid\; p \;\mid\; \mathrm{cont}\,(k) \\
e & ::= & v \;\mid\; x \;\mid\; \lambda x.e \;\mid\; \text{let rec } x = \lambda x.e \text{ in } e \;\mid\; {<}l{>}\ e\ e \\
 & \mid & \mathrm{frame}_{<l>,\mathcal{E},p'}\, e \;\mid\; \mathrm{env}_{\mathcal{E}}\ e \;\mid\; \mathrm{mc}\ e \text{ with } ({<}l{>}, x, x, \overline{x}, e)\ e\ e \\
 & \mid & \mathrm{capture}_{<l>} \text{up to } e \text{ with } e \;\mid\; \mathrm{cap}_{<l>,\mathcal{E}} \text{up to } p \text{ with } v \\
 & \mid & \mathrm{reinstate}_{<l>}\ e\ e \;\mid\; \mathrm{setprompt}_{<l>}\ e\ e \;\mid\; \mathrm{newprompt} \\
k & ::= & \square \;\mid\; {<}l{>}\ k\ v \;\mid\; {<}l{>}\ e\ k \;\mid\; \mathrm{frame}_{<l>,\mathcal{E},p'}\, k \;\mid\; \mathrm{env}_{\mathcal{E}}\ k \\
 & \mid & \mathrm{mc}\ k \text{ with } ({<}l{>}, x, x, \overline{x}, e)\ e\ e \\
 & \mid & \mathrm{capture}_{<l>} \text{up to } k \text{ with } v \;\mid\; \mathrm{capture}_{<l>} \text{up to } e \text{ with } k \\
 & \mid & \mathrm{reinstate}_{<l>}\ k\ v \;\mid\; \mathrm{reinstate}_{<l>}\ e\ k \;\mid\; \mathrm{setprompt}_{<l>}\ k\ e \\
p' & ::= & p \;\mid\; \perp \qquad\qquad \mathcal{E} \;::=\; [] \;\mid\; (x \mapsto v) :: \mathcal{E}
\end{array}
$$

Additional constraint:

- A continuation $\mathrm{cont}\,(k)$ is either empty (k is $\square$) or its innermost operator is `frame` (k ends with $\mathrm{frame}_{<l>,\mathcal{E},p'}\square$).

Figure 1. Grammar of terms and continuations

The last operator $(\mathrm{mc}\ e \text{ with } ({<}l{>}, x_1, x_2, \overline{x_3}, e_1)\ e_2\ e_3)$ deconstructs a continuation relying on its stack frame structure. It compares <*l*> and the return address on top of the continuation. If the labels match, the continuation is split at the second innermost frame operator in a head (the inner subcontinuation) bound to x_1 and a tail (the outer subcontinuation) bound to x_2. Furthermore, the variables $\overline{x_3}$ are bound to the values of the topmost stack frame. Then e_1 is executed in the so extended evaluation environment. There are two other cases: either the return address does not match (e_2 is executed) or the continuation is empty (e_3 is executed).

The language has 3 kinds of values: closures, prompts and continuations.

4.1.2 Discussion

Having explicit closures and the `env` operator is the usual approach for the implementation of lexical scoping in small-step environment-based semantics. As a side-effect, the `env` operator also ensures that continuations are independent of any evaluation environment, i.e., any continuation brings its required environment in an `env` construct. To some extent, this is similar to the destruct-time λ-calculus [8, 33], which delays substitutions until values are consumed. That way, bindings can be marshalled and move between scopes.

Delimited continuations are a natural choice in our context. Indeed, when the `mc` operator splits a continuation into smaller ones, it instantiates continuations that represent only parts of execution contexts. This is what delimited continuations are designed for. Our framework is similar to the ones of Gunter et al. [17] and Dybvig et al. [13]. The following table approximates how our operators match with existing frameworks. Readers can refer to Shan [34], Kiselyov [21] and Dybvig et al. [13] for more complete comparisons.

ReCaml	Dybvig et al. [13]	Gunter et al. [17]
`newprompt`	`newPrompt`	`new_prompt`
`setprompt`	`pushPrompt`	`set`
`capture`	`withSubCont`	`cupto`
`reinstate`	`pushSubCont`	fun. application

In addition, we adapt the framework:

- We align the delimiters of continuations with the delimiters of stack frames. To do so, we annotate the `frame` operator with an optional prompt in order to delimit where prompts are set. Furthermore, the continuation operators must have a call site label <*l*> in order to insert `frame` constructs.
- We have to introduce a dummy `cap` operator to align a stack frame delimiter with the inner delimiter of the continuation. To do so, a `frame` operator (which needs the evaluation environment) is inserted at the innermost position of the continuation, in place of the `capture` operator. The `cap` operator saves the needed evaluation environment (the one at the position of the `capture` operator) before the continuation is actually captured.
- Like Dybvig et al. [13], we encode continuations in a specific `cont` form rather than a closure [17]. That way, the linear structure of continuations (a stack in the implementation; the nesting of evaluation contexts in the language) is maintained and can be used by the `mc` operator. Furthermore, encoding a continuation as a closure would introduce a variable, which would infringe the type preservation lemma due to the typing of call site labels, as we will see later (Section 4.4). Last, making the distinction between continuations and closures, the `mc` operator does not have to handle regular closures.

Intuitively, a `frame` operator is inserted when a call is done and disappears when the callee terminates. Thus, when a continuation is captured, all its activations are delimited by `frame` operators. The `mc` operator uses them to split continuations into smaller ones. One can note that the environment of a `frame` is redundant. This environment indeed comes from the enclosing `env` construct. While our choice imposes a dummy `cap` operator in the continuation framework, it makes `mc` simpler. Indeed, it does not need to look for `env` constructs to collect environments when a continuation is split.

4.2 Semantics

The small step operational semantics of Figure 2 formalizes the above description of ReCaml. We adopt an environment-based approach with lexical scoping of variables. The judgment $\mathcal{E} \vdash e \to e'$ asserts that the term e reduces to e' in the evaluation environment $\mathcal{E}$. Rules SUBST, CLOSE and LETREC are the classical ones for substituting a variable, building a closure and recursive definitions, respectively. As usual with environment-based semantics, the `env` operator installs a local environment in order to evaluate the nested term (rule ENV). Because the `frame` operator bounds activations, the local environment used to evaluate the nested term is empty (rule FRAME). Here, it is the role of the inner `env` operator to give the actual execution environment. Figure 2 gives only primitive reduction rules. Except `frame` and `env`, which need special treatment of the environment, the CONTEXT rule generically reduces contexts according to the grammar of k. Because it is constrained with values, it fixes a strict right-to-left call-by-value evaluation order.

The management of the `frame` operator is one originality of the semantics. It implements the life cycle of activations. This operator is instantiated when a closure is applied (rule APPLY), when a prompt is set (rule SETPROMPT) and when a continuation is reinstated (rule REINSTATE). It collapses when a callee activation returns a value (rule FRAMEVAL). Paired with the `frame` operator, the `env` operator provides the local evaluation environment for the instantiated activation. For instance, applying a closure, e.g., the identity function, proceeds as follows:

$$\begin{aligned}
\mathcal{E} \vdash \text{<}l\text{>}\ (\lambda x.x, \mathcal{E}_2)\ v &\xrightarrow{\text{APPLY}} \text{frame}_{\text{<}l\text{>},\mathcal{E},\perp}\left(\text{env}_{(x\mapsto v)::\mathcal{E}_2}\ x\right)\\
&\xrightarrow{\text{FRAME,ENV,SUBST}} \text{frame}_{\text{<}l\text{>},\mathcal{E},\perp}\left(\text{env}_{(x\mapsto v)::\mathcal{E}_2}\ v\right)\\
&\xrightarrow{\text{FRAME,ENVVAL}} \text{frame}_{\text{<}l\text{>},\mathcal{E},\perp} v\\
&\xrightarrow{\text{FRAMEVAL}} v
\end{aligned}$$

Capturing a continuation is done in two steps. First, the evaluation environment at the `capture` operator is saved, mutating the operator into `cap` (rule CAP1). The second step is the standard continuation capturing. A `cap` operator using prompt p is only reduced within a `frame` tagged by p. If such a `frame` exists, the context k between this `frame` and `cap` is reified as a continuation `cont(`k`)`. A `frame` is inserted in place of `cap` consistently with the constraint of our language (see at the bottom of Fig. 1). The closure argument of `cap` is applied to the resulting continuation (rule CAP2). In rule CAP2, the enclosing prompt p is consumed. The system proceeds as follows:

$$\begin{aligned}
&\mathcal{E}_1 \vdash \text{frame}_{\text{<}l_1\text{>},\mathcal{E}_1,p}\left(\text{env}_{\mathcal{E}_2}\left(\text{capture}_{\text{<}l_2\text{>}}\text{up to}\,p\text{ with }(\lambda x.e, \mathcal{E}_3)\right)\right)\\
&\xrightarrow{\text{FRAME,ENV,CAP1}} \text{frame}_{\text{<}l_1\text{>},\mathcal{E}_1,p}\left(\text{env}_{\mathcal{E}_2}\left(\text{cap}_{\text{<}l_2\text{>},\mathcal{E}_2}\text{up to}\,p\text{ with}(\lambda x.e, \mathcal{E}_3)\right)\right)\\
&\xrightarrow{\text{CAP2}} \text{frame}_{\text{<}l_1\text{>},\mathcal{E}_1,\perp}\left(\text{env}_{(x\mapsto\text{cont}(k))::\mathcal{E}_3}\ e\right)\\
&\qquad\qquad\text{with } k = \text{env}_{\mathcal{E}_2}\left(\text{frame}_{\text{<}l_2\text{>},\mathcal{E}_2,\perp}\square\right)
\end{aligned}$$

We proceed in two steps in order to handle easily any context in place of $\text{env}_{\mathcal{E}_2}$. If no frame tagged by p encloses `cap` p (terms structured like $k\left[\text{cap}_{\text{<}l_2\text{>},\mathcal{E}_2}\text{up to}\,p\text{ with }v\right]$ where k does not contain any $\text{frame}_{\text{<}\_\text{>},\_,p}$), a runtime error occurs.

The `mc` operator splits a continuation at the second innermost `frame`, which delimits the top stack frame (rule MCMATCH). The rule MCMATCH' handles the case where the continuation contains a single stack frame. The tail subcontinuation is therefore the empty continuation. The rules for `mc` assume that the continuation is either empty (rule MCEMPTY) or the innermost operator within the continuation is `frame` (rules MCNOMATCH, MCMATCH and MCMATCH'). As shown in Fig. 1, this is enforced as a structural constraint on the language. It is trivial to show that the semantics produces only continuations that conform to this constraint.

4.3 Type System

The type system adheres to the usual design of the simply-typed λ-calculus. Types may be type variables[5], usual functional types, prompt types or continuation types. The type of a prompt is parameterized by the type of the values that flow through delimiters tagged by that prompt. The type of a continuation is parameterized by the type of the parameter and the type of the result of the continuation. The grammar for types is:

$$\tau \quad ::= \quad \alpha \quad | \quad \tau \to \tau \quad | \quad \tau\ \text{prompt} \quad | \quad \tau \xrightarrow{\kappa} \tau$$

Fig. 3 gives the type system for the term language. The judgement $E, P, L, \tau \vdash e : \tau_e$ asserts that given the typing environments E, P and L, in an enclosing function whose return type is τ, the term e has type τ_e. E (resp. P) maps variables (resp. prompts) to types. L maps call site labels to label types, which are triplets $\{\tau_{par}, \tau_{res}, V\}$ where τ_{par} and τ_{res} are types; and V is an environment that maps variables to types. The inference algorithm computes τ_e and L.

The L environment is intended for splitting continuations at activation boundaries. Figure 4 gives an intuition of its interpretation, based on the semantics of the `mc` operator. A $\tau_1 \xrightarrow{\kappa} \tau_n$ continuation

[5] We use type variables for convenience to solve the type inference problem. As ReCaml is simply typed, type variables are never generalized as type parameters. Instead, they are unknown types that shall later be instantiated by unification. This is similar to Caml's *weak type variables* such as `'_a` in the type `'_a list ref` of `ref []`.

$$\text{SUBST: } \mathcal{E} \vdash x \to \mathcal{E}(x) \qquad \text{CLOSE: } \mathcal{E} \vdash \lambda x.e \to (\lambda x.e, \mathcal{E}) \qquad \text{APPLY: } \mathcal{E}_1 \vdash \langle l\rangle\ (\lambda x.e, \mathcal{E}_2)\ v \to \text{frame}_{\langle l\rangle,\mathcal{E}_1,\bot}\left(\text{env}_{(x\mapsto v)::\mathcal{E}_2}\ e\right)$$

$$\text{LETREC: } \mathcal{E} \vdash \text{let rec } x_1 = \lambda x_2.e_1 \text{ in } e_2 \to \text{env}_{(x_1\mapsto(\lambda x_2.\text{let rec } x_1=\lambda x_2.e_1 \text{ in } e_1,\mathcal{E}))::\mathcal{E}}\ e_2 \qquad \text{FRAMEVAL: } \mathcal{E}_1 \vdash \text{frame}_{\langle l\rangle,\mathcal{E}_2,p'} v \to v$$

$$\text{ENVVAL: } \mathcal{E}_1 \vdash \text{env}_{\mathcal{E}_2}\ v \to v \qquad \text{MCEMPTY: } \mathcal{E} \vdash \text{mc cont}(\square) \text{ with } (\langle l\rangle, x_1, x_2, \overline{x_3}, e_1)\ e_2\ e_3 \to e_3$$

$$\text{MCNOMATCH: } \frac{l_1 \neq l_2}{\mathcal{E}_1 \vdash \text{mc cont}\left(k\left[\text{frame}_{\langle l_1\rangle,\mathcal{E}_2,p}\square\right]\right) \text{ with } (\langle l_2\rangle, x_1, x_2, \overline{x_3}, e_1)\ e_2\ e_3 \to e_2}$$

$$\text{MCMATCH: } \frac{k_1 \text{ does not contain any frame} \qquad \overline{\mathcal{E}_1(x_3) = v_3}}{\begin{array}{l}\mathcal{E} \vdash \text{mc cont}\left(k_2\left[\text{frame}_{\langle l_2\rangle,\mathcal{E}_2,p_2}\left(k_1\left[\text{frame}_{\langle l_1\rangle,\mathcal{E}_1,p_1}\square\right]\right)\right]\right) \text{ with } (\langle l_1\rangle, x_1, x_2, \overline{x_3}, e_1)\ e_2\ e_3 \\ \to \text{env}_{\left(x_1\mapsto\text{cont}\left(k_1\left[\text{frame}_{\langle l_1\rangle,\mathcal{E}_1,p_1}\square\right]\right)\right)::\left(x_2\mapsto\text{cont}\left(k_2\left[\text{frame}_{\langle l_2\rangle,\mathcal{E}_2,p_2}\square\right]\right)\right)::\overline{(x_3\mapsto v_3)}::\mathcal{E}}\ e_1\end{array}}$$

$$\text{MCMATCH': } \frac{k_1 \text{ does not contain any frame} \qquad \overline{\mathcal{E}_1(x_3) = v_3}}{\begin{array}{l}\mathcal{E} \vdash \text{mc cont}\left(k_1\left[\text{frame}_{\langle l_1\rangle,\mathcal{E}_1,p_1}\square\right]\right) \text{ with } (\langle l_1\rangle, x_1, x_2, \overline{x_3}, e_1)\ e_2\ e_3 \\ \to \text{env}_{\left(x_1\mapsto\text{cont}\left(k_1\left[\text{frame}_{\langle l_1\rangle,\mathcal{E}_1,p_1}\square\right]\right)\right)::(x_2\mapsto\text{cont}(\square))::\overline{(x_3\mapsto v_3)}::\mathcal{E}}\ e_1\end{array}} \qquad \text{NEWPROMPT: } \frac{p \text{ is fresh}}{\mathcal{E} \vdash \text{newprompt} \to p}$$

$$\text{CAP1: } \mathcal{E} \vdash \text{capture}_{\langle l\rangle}\text{up to } v_1 \text{ with } v_2 \to \text{cap}_{\langle l\rangle,\mathcal{E}}\text{up to } v_1 \text{ with } v_2$$

$$\text{CAP2: } \frac{k \text{ does not contain any frame}_{\langle\_\rangle,\_,p}}{\mathcal{E}_1 \vdash \text{frame}_{\langle l_1\rangle,\mathcal{E}_2,p} k\left[\text{cap}_{\langle l_2\rangle,\mathcal{E}_3}\text{up to } p \text{ with } (\lambda x.e, \mathcal{E}_4)\right] \to \text{frame}_{\langle l_1\rangle,\mathcal{E}_2,\bot}\left(\text{env}_{\left(x\mapsto\text{cont}\left(k\left[\text{frame}_{\langle l_2\rangle,\mathcal{E}_3,\bot}\square\right]\right)\right)::\mathcal{E}_4}\ e\right)}$$

$$\text{SETPROMPT: } \mathcal{E} \vdash \text{setprompt}_{\langle l\rangle}\ p\ e \to \text{frame}_{\langle l\rangle,\mathcal{E},p}\left(\text{env}_{\mathcal{E}}\ e\right) \qquad \text{REINSTATE: } \mathcal{E} \vdash \text{reinstate}_{\langle l\rangle}\text{ cont}(k)\ v \to \text{frame}_{\langle l\rangle,\mathcal{E},\bot} k[v]$$

Context rules:

$$\text{FRAME: } \frac{[] \vdash e \to e'}{\mathcal{E}_1 \vdash \text{frame}_{\langle l\rangle,\mathcal{E}_2,p'} e \to \text{frame}_{\langle l\rangle,\mathcal{E}_2,p'} e'} \qquad \text{ENV: } \frac{\mathcal{E}_2 \vdash e \to e'}{\mathcal{E}_1 \vdash \text{env}_{\mathcal{E}_2}\ e \to \text{env}_{\mathcal{E}_2}\ e'} \qquad \text{if no other rule matches, CONTEXT: } \frac{\mathcal{E} \vdash e \to e'}{\mathcal{E} \vdash k[e] \to k[e']}$$

$k[a]$ substitutes a for $\square$ in k, where a is either a term, hence resulting in a term, or a continuation, hence resulting in a continuation.

Figure 2. Operational semantics

k is split into k_{head} ($\tau_1 \xrightarrow{\kappa} \tau_2$) and k_{tail} ($\tau_2 \xrightarrow{\kappa} \tau_n$). Composing the two subcontinuations results obviously in the original continuation. τ_2 is the return type of the function that encloses l_1. This is the reason why the type judgment has τ (the type of the enclosing function) in its left-hand side. τ_1 is the return type of the call l_1. In order to type values that `mc` retrieves from the popped activation, e.g., the value of x_1, the type of l_1 contains the type environment at the call l_1. Consequently, the type of l_1 is:

- $\tau_{par_{l_1}} = \tau_1$ is the type of the value that flows at the boundary;
- $\tau_{res_{l_1}} = \tau_2$ is the return type of the enclosing function;
- $V_{l_1} = [x_1 \mapsto \tau_{x_1}]$ binds types to the activation variables.

In the example (Section 3), the types of labels are:

$$\begin{array}{l}\texttt{L1}\mapsto\{\tau_{par} = \texttt{int}; \tau_{res} = \texttt{int}; \\ \quad V = [\texttt{fib} \mapsto \texttt{int} \to \texttt{int}; \texttt{n} \mapsto \texttt{int}]\} \\ \texttt{L2}\mapsto\{\tau_{par} = \texttt{int}; \tau_{res} = \texttt{int}; \\ \quad V = [\texttt{fib} \mapsto \texttt{int} \to \texttt{int}; \texttt{n} \mapsto \texttt{int}; \texttt{fn1} \mapsto \texttt{int}]\} \\ \texttt{Lroot}\mapsto\{\tau_{par} = \texttt{int}, \tau_{res} = \texttt{unit}, V = [\texttt{fib} \mapsto \texttt{int} \to \texttt{int}]\}\end{array}$$

As usual, when typing an application (APPLY), the two subexpressions are typed using the same hypotheses. The first subexpression must be a function accepting values of the type of the second subexpression. The originality of our rule concerning application is the calculus of the type of the label. This type captures the type of the enclosing function τ_1, the current environment E and the type τ_3 that *flows* at the label, i.e., the type of the result.

Some constructs introduce frames and therefore modify the type of the enclosing function of a subexpression. For example, the type of the enclosing function of e_2 in setprompt$_{\langle l\rangle}$ e_1 e_2 is τ_2 because the setprompt operator encloses e_2 in a frame whose prompt is of type τ_2 (see SETPROMPT in Figures 2 and 3).

Typing a continuation expression (CONT) requires a specific type system. It is mutually recursive with the type system for terms. The judgment $E, P, L, \tau \vdash_\kappa k : \tau_1 \xrightarrow{\kappa} \tau_2$ is similar to the one for terms. Most of the rules derive from the type system for terms. For instance, the following rule is immediate from rule APPLY (Fig. 3):

$$\text{APPLYL: } \frac{\begin{array}{c}E, P, L, \tau_1 \vdash_\kappa k : \tau_2 \xrightarrow{\kappa} (\tau_3 \to \tau_4) \\ E, P, L, \tau_1 \vdash v : \tau_3 \\ L(l) = \{\tau_{par} = \tau_4, \tau_{res} = \tau_1, V = E\}\end{array}}{E, P, L, \tau_1 \vdash_\kappa \langle l\rangle\ k\ v : \tau_2 \xrightarrow{\kappa} \tau_4}$$

We therefore omit the rules, except the following additional one for empty continuations:

$$\text{HOLE: } E, P, L, \tau_1 \vdash_\kappa \square : \tau_2 \xrightarrow{\kappa} \tau_2$$

4.4 Soundness

We consider soundness as the conjunction of type preservation and progress, stated as follows.

LEMMA 1 (Type preservation). *Given a term e_1 and an evaluation environment $\mathcal{E}$ such that $T(\mathcal{E}), P, L, \tau_1 \vdash e_1 : \tau_2$. If e_1 reduces to e_2 in $\mathcal{E}$, then there exists an extension P' of P ($\forall p$ and τ_p, $P(p) = \tau_p \Rightarrow P'(p) = \tau_p$) such that in P', e_2 has the same type as e_1, i.e., $T(\mathcal{E}), P', L, \tau_1 \vdash e_2 : \tau_2$.*

$$\textsc{Closure:}\ \frac{(x \mapsto \tau_2) :: T(\mathcal{E}), P, L, \tau_3 \vdash e : \tau_3}{E, P, L, \tau_1 \vdash (\lambda x.e, \mathcal{E}) : \tau_2 \to \tau_3} \qquad \textsc{Prompt:}\ E, P, L, \tau \vdash p : P(p)\ \text{prompt} \qquad \textsc{Cont:}\ \frac{[\,], P, L, \tau_3 \vdash_\kappa k : \tau_2 \xrightarrow{\kappa} \tau_3}{E, P, L, \tau_1 \vdash \text{cont}(k) : \tau_2 \xrightarrow{\kappa} \tau_3}$$

$$\textsc{Var:}\ E, P, L, \tau \vdash x : E(x) \qquad \textsc{Abs:}\ \frac{(x \mapsto \tau_2) :: E, P, L, \tau_3 \vdash e : \tau_3}{E, P, L, \tau_1 \vdash \lambda x.e : \tau_2 \to \tau_3}$$

$$\textsc{LetRec:}\ \frac{(x_1 \mapsto \tau_3 \to \tau_4) :: (x_2 \mapsto \tau_3) :: E, P, L, \tau_4 \vdash e_1 : \tau_4 \qquad (x_1 \mapsto \tau_3 \to \tau_4) :: E, P, L, \tau_1 \vdash e_2 : \tau_2}{E, P, L, \tau_1 \vdash \text{let rec } x_1 = \lambda x_2.e_1 \text{ in } e_2 : \tau_2}$$

$$\textsc{Apply:}\ \frac{E, P, L, \tau_1 \vdash e_1 : \tau_2 \to \tau_3 \qquad E, P, L, \tau_1 \vdash e_2 : \tau_2 \qquad L(l) = \{\tau_{par} = \tau_3, \tau_{res} = \tau_1, V = E\}}{E, P, L, \tau_1 \vdash {<}l{>}\ e_1\ e_2 : \tau_3}$$

$$\textsc{Frame:}\ \frac{[\,], P, L, P(p) \vdash e : P(p) \qquad L(l) = \{\tau_{par} = P(p), \tau_{res} = \tau, V = E\} \qquad E = T(\mathcal{E})}{E, P, L, \tau \vdash \text{frame}_{{<}l{>}, \mathcal{E}, p} e : P(p)} \qquad \textsc{Env:}\ \frac{T(\mathcal{E}), P, L, \tau_1 \vdash e : \tau_2}{E, P, L, \tau_1 \vdash \text{env}_{\mathcal{E}}\ e : \tau_2}$$

$$\textsc{Frame':}\ \frac{[\,], P, L, \tau_2 \vdash e : \tau_2 \quad L(l) = \{\tau_{par} = \tau_2, \tau_{res} = \tau_1, V = E\} \quad E = T(\mathcal{E})}{E, P, L, \tau_1 \vdash \text{frame}_{{<}l{>}, \mathcal{E}, \perp} e : \tau_2} \qquad \textsc{NewPrompt:}\ E, P, L, \tau_1 \vdash \text{newprompt} : \tau_2\ \text{prompt}$$

$$\textsc{MC:}\ \frac{\begin{array}{c} E, P, L, \tau_1 \vdash e_1 : \tau_3 \xrightarrow{\kappa} \tau_5 \qquad L(l) = \{\tau_{par} = \tau_3, \tau_{res} = \tau_4, V = \mathcal{E}\} \\ \left(x_1 \mapsto \tau_3 \xrightarrow{\kappa} \tau_4\right) :: \left(x_2 \mapsto \tau_4 \xrightarrow{\kappa} \tau_5\right) :: \overline{(x_3 \mapsto \mathcal{E}(x_3))} :: E, P, L, \tau_1 \vdash e_2 : \tau_2 \qquad E, P, L, \tau_1 \vdash e_3 : \tau_2 \qquad E, P, L, \tau_1 \vdash e_4 : \tau_2 \end{array}}{E, P, L, \tau_1 \vdash \text{mc } e_1 \text{ with } ({<}l{>}, x_1, x_2, \overline{x_3}, e_2)\ e_3\ e_4 : \tau_2}$$

$$\textsc{Capture:}\ \frac{E, P, L, \tau_1 \vdash e_1 : \tau_3\ \text{prompt} \qquad E, P, L, \tau_1 \vdash e_2 : \left(\tau_2 \xrightarrow{\kappa} \tau_3\right) \to \tau_3 \qquad L(l) = \{\tau_{par} = \tau_2, \tau_{res} = \tau_1, V = E\}}{E, P, L, \tau_1 \vdash \text{capture}_{{<}l{>}} \text{up to } e_1 \text{ with } e_2 : \tau_2}$$

$$\textsc{Cap:}\ \frac{T(\mathcal{E}), P, L, \tau_1 \vdash v : \left(\tau_2 \xrightarrow{\kappa} P(p)\right) \to P(p) \qquad L(l) = \{\tau_{par} = \tau_2, \tau_{res} = \tau_1, V = E\} \qquad E = T(\mathcal{E})}{E, P, L, \tau_1 \vdash \text{cap}_{{<}l{>}, \mathcal{E}} \text{up to } p \text{ with } v : \tau_2}$$

$$\textsc{Reinstate:}\ \frac{E, P, L, \tau_1 \vdash e_1 : \tau_3 \xrightarrow{\kappa} \tau_2 \qquad E, P, L, \tau_1 \vdash e_2 : \tau_3 \qquad L(l) = \{\tau_{par} = \tau_2, \tau_{res} = \tau_1, V = E\}}{E, P, L, \tau_1 \vdash \text{reinstate}_{{<}l{>}}\ e_1\ e_2 : \tau_2}$$

$$\textsc{SetPrompt:}\ \frac{E, P, L, \tau_1 \vdash e_1 : \tau_2\ \text{prompt} \qquad E, P, L, \tau_2 \vdash e_2 : \tau_2 \qquad L(l) = \{\tau_{par} = \tau_2, \tau_{res} = \tau_1, V = E\}}{E, P, L, \tau_1 \vdash \text{setprompt}_{{<}l{>}}\ e_1\ e_2 : \tau_2}$$

Where $T(\mathcal{E}) = [x \mapsto \tau_x |\, [\,], P, L, \tau \vdash \mathcal{E}(x) : \tau_x]$, i.e., function T computes a type environment from an evaluation environment.

Figure 3. Type system for terms

The existential quantification of P' is the technique of Gunter et al. [17][6] in order to handle the `newprompt` case. Assume $T(\mathcal{E}), P, L, \tau_1 \vdash \text{newprompt} : \tau_2$ prompt. `newprompt` reduces to a fresh prompt p in $\mathcal{E}$. p is not in the domain of P. Hence choosing $P' = (p \mapsto \tau_2) :: P$ trivially ensures type preservation. In the other cases, we systematically choose $P' = P$.

Unlike usual proofs, we do not use a lemma showing that extending the environment would preserve typing. Instead, we use a context invariance approach. While Pierce [30], Pierce et al. [31] do so for pedagogical reasons, we have to because the standard weakening lemma is false due to the typing of call sites. Indeed, in L, the V field of the type associated with the label stores the typing environment (rules APPLY, FRAME, FRAME', CAPTURE, CAP, REINSTATE and SETPROMPT). Hence adding new variables to the environment, even if they do not occur free, may change label types in L. Intuitively, it would change the structure and content of stack frames, hence their types. Nevertheless, we must prove that the type of a value is independent of the context.

LEMMA 2 (Typing values). *Given a value v, the type of v is independent of any context: $E, P, L, \tau \vdash v : \tau_v$, implies $E', P, L, \tau' \vdash v : \tau_v$ for any E' and τ'.*

This lemma is trivial following the CLOSURE, PROMPT and CONT typing rules. Type preservation for the SUBST reduction rule is therefore immediate. Restricting evaluation environments to values is a pragmatic solution to avoid any variable capture issue upon substitution.

In order to prove each of the other cases, we proceed in two stages. We first show that in order to type subterms, the rules build exactly the same environment before and after reduction. Hence reduction preserves the type of subterms. Then we use these results as premises of the typing rules for the reduced term.

Let's sketch for instance the case of the APPLY reduction rule. Before reduction, assuming the parameter v has type τ_v, the body

[6] Gunter et al. [17] note $e_1/P_1 \subset e_2/P_2$, where P_1 and P_2 are sets of prompts. The $\subset$ relation denotes that given a typing environment over P_1, there exists an extension over P_2 such that e_1 and e_2 have the same type in their respective prompt environments. Using our P and P' as typing environments (respectively over P_1 and P_2), the $\subset$ relation is (part of) what our type preservation lemma states.

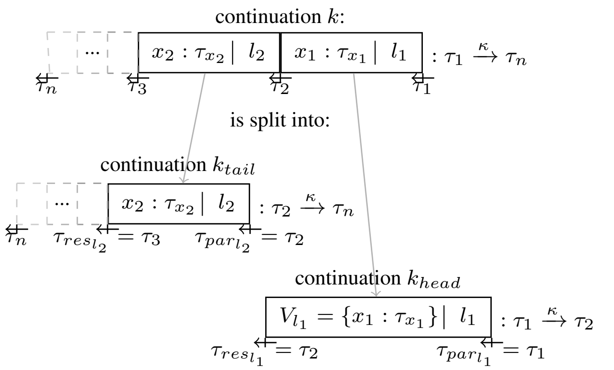

Figure 4. Intuition for typing activation boundary annotations.

e of the closure is typed in the $(x \mapsto \tau_v) :: T(\mathcal{E})$ environment and the return type of the enclosing function is τ_e the type of e (CLOSURE typing rule). After reduction, it is typed in the environment $T((x \mapsto v) :: \mathcal{E})$ according to the FRAME' and ENV typing rules. From the definition of T, and invoking the lemma on typing values, the two environments are equal. Hence the type of subterm e is preserved. Using the ENV and FRAME' typing rules, we conclude that the type is the same before and after reduction. Last we check that the APPLY typing rule (before reduction) and the FRAME' typing rule (after reduction) compute the same label type for l. Hence the APPLY reduction rule preserves types.

Traversing the evaluation context to the redex, evaluation rules CONTEXT, FRAME and ENV compute at each step a new evaluation environment for each subcontext. Typing rules do the same with typing environments. Along the path to the redex, we observe that the rules recursivelly ensure that the evaluation and typing environments are equal up to T. This completes the proof.

LEMMA 3 (Progress). *Given e_1 such that $[], P, L, \tau \vdash e_1 : \tau$. Then e_1 is either a value; or e_1 is a runtime error (redex position is* cap$_{<_>,_}$ up to p with v *but it is not enclosed by any* frame$_{<_>,_,p}$*); or e_1 reduces to some term e_2 in the empty evaluation environment.*

In order to prove progress, we inductively analyze the typing rules. This proof is classical.

The proofs have been mechanized using the Coq theorem prover and the library of Aydemir et al. [6], which together help to do machine-verified formal proofs on language semantics and type systems. For commodity reason, our Coq scripts differ in the following from the system of this paper. We explode the MCMATCH, MCMATCH', CAP1/CAP2 and REINSTATE reduction rules into detailed small steps. For example, we instantiate the CAP2 for each operator in the language k of evaluation contexts. For this purpose, we introduce additional dummy operators for in-progress `mc` and `reinstate`. In addition, the implementation of the `mc` operator has to look for the innermost (`frame`) operator of the continuation operand. Instead, it is much more convenient to reverse the nesting of operators in the continuation. At the cost of yet another dummy operator and of additional rules, we therefore represent continuations inside out. We use the technique of Gunter et al. [17] to implement the freshness of instantiated prompts. Last, we move from the grammar to the type system the constraint on the form of continuations (bottom of Fig. 1).

4.5 Alternatives

One of the constraints that guides our work is to leave unchanged the application compiler. The rationale behind this constraint is that it makes it easier to integrate the ReCaml approach into existing compilers. To fulfill this constraint, we need to accommodate the choices done in legacy compilers. We identify several alternatives that shall impact dynamic updates. In the following, we present how these points integrate our formal system. We focus on the specificities of our language. Hence we do not discuss variations, e.g., of the continuation framework, which have already been studied by Dybvig et al. [13].

Usually, the implementation of execution states is not of great interest in the design of a language. This issue regards the compiler. But because ReCaml focuses on modelling state manipulations, we have to take into consideration the implementation. For instance, label types depend on the context, and therefore on captured environments when building closures.

Regarding variables, we implement the following rules in the semantics and type system:

- When a closure is built, it captures all the variables in the scope of which the λ operator lies, regardless these variables occur free in the body of the function.
- The parameter of a function is systematically added to the evaluation environment, regardless it occurs free in the body of the function. We do the same for `let rec`.

This is a coarse behavior. Indeed, many compilers optimize closures in order to capture only the variables that occur free. In order to model this behavior in ReCaml, we can replace the CLOSE reduction rule with the following one:

$$\text{RESTRICT-CLOSE: } \mathcal{E} \vdash \lambda x.e \rightarrow (\lambda x.e, restrict_e(\mathcal{E}))$$

where $restrict$ computes the restriction of the environment, e.g., $[x \mapsto \mathcal{E}(x) \,|\, x \in fv(e)]$ to capture only the variables that occur free in the body. We have to change the type system accordingly, replacing ABS with:

$$\text{RESTRICT-ABS: } \frac{(x \mapsto \tau_2) :: restrict_e(E), P, L, \tau_3 \vdash e : \tau_3}{E, P, L, \tau_1 \vdash \lambda x.e : \tau_2 \rightarrow \tau_3}$$

Type soundness obviously still holds.

This implementation does the restriction when the closure is built. This is what happens in many compilers. Instead, we could have delayed the restriction until application, hence inserting $restrict$ in the APPLY reduction rule and in the CLOSURE and ABS typing rules. As of ReCaml, both implementations have the same behavior. We can also restrict parameters and `let rec`-bound variables using the same technique.

Accurate modelling of the variables is important as it impacts type labels and the amount of values the `mc` operator is able to retrieve from continuations. Other aspects, such as tail-call optimization and function inlining, impact when new stack frames are created. Consequently, they (indirectly) impact the outcome of the `mc` operator as well.

Tail-call optimization consists in destroying the calling activation at the time of a call when it occurs at the return position. We can implement this optimization thanks to additional rules, e.g., duplicating the APPLY reduction rule for the specific case, such that it does not insert any new `frame` operator. Possibly, there are also several `env` constructs that must collapse with the stack frame.

$$\text{TAIL-APPLY: } \frac{k \text{ contains only (0 or more) env}}{\begin{array}{l}\mathcal{E} \vdash \text{frame}_{<l_1>,\mathcal{E}_1,p'_1}\left(k\left[\texttt{<}l_2\texttt{>}\ (\lambda x.e, \mathcal{E}_2)\ v\right]\right)\\ \rightarrow \text{frame}_{<l_1>,\mathcal{E}_1,p'_1}\left(\text{env}_{(x\mapsto v)::\mathcal{E}_2}\ e\right)\end{array}}$$

Notice that the `frame` in the right-hand side is the exact copy of the left-hand side one. Indeed, the properties of the enclosing stack frame (return address, local environment) are unaffected.

In order to handle inlined calls, the idea is coarsely the same, without any constraint on the context of the call. Nevertheless, there

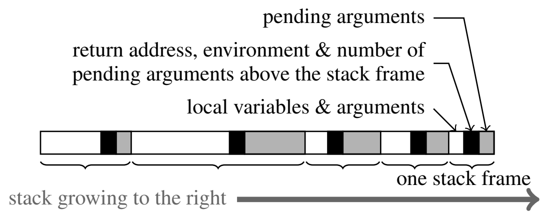

Figure 5. Structure of the stack in the virtual machine.

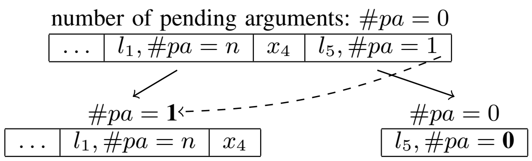

Figure 6. Splitting a continuation.

are additional difficulties: call sites within the inlined function are replicated; the environment of the caller and callee environments shall merge. We do not run into deeper details in this paper, leaving the issues to further contributions.

5. Compiler Implementation

As a proof of concept, we have developed a prototype compiler of ReCaml, which targets a modified ZAM2 [23] virtual machine. The machine has a single stack for locals, function arguments, register backup and return addresses. In addition to the stack pointer, the machine has 4 registers:

- the program counter points at the next instruction to execute;
- the environment points at the values stored in the closure;
- the argument counter tells how many pending arguments have been pushed, as the machine implements the push / enter uncurrying technique [27];
- the accumulator holds an intermediate result.

As shown in Figure 5, stack frames are delimited by blocks that save the return program counter, the environment and the argument counter registers. Pending arguments, if any (possibly 0), are pushed immediately above this block. The virtual machine provides a specific instruction for tail calls. Like our TAIL-APPLY rule (Section 4.5), this instruction pops the local environment; it pushes a new one; and it branches to the body of the callee. The push / enter uncurrying technique lets the caller of a function push all the available arguments onto the stack. The callee is responsible of popping only those it can immediately handle (or to build a partial-application closure if there are not enough parameters on the stack). While there are some pending arguments on the stack, the `return` instruction assumes that the return value is a closure, and makes a call. When all the pending arguments are consumed, the instruction returns back to the caller.

We extend the virtual machine to support continuations. A continuation is implemented as a slice of the machine stack with a copy of the argument counter register. Other registers (program counters, closure environment and accumulator) are saved within the slice of the stack by the generated code as required by the ZAM2. A prompt is a pointer to a position in the stack. The `capture` operator copies to the heap the slice between the prompt and the top of the stack; it saves the argument counter; and it makes a call to the body function. The `reinstate` operator copies from the heap back to the stack; it restores the argument counter; and it performs a `return` instruction with the argument. In addition to retrieving the stack pointer, setting a prompt makes a call such that the lower bound of a continuation is always aligned with a stack frame boundary, consistently with our semantics.

Based on this implementation of continuations, the `mc` operator first checks whether the continuation is empty. If not, it uses the recorded number of pending arguments in order to skip data down to the first return address. The retrieved address is compared with the operand of the `mc` operator. Static knowledge gives the structure and size of the matching stack frame at the top of the continuation. This information allows to split the continuation at the stack frame boundary and retrieve values from the popped stack frame.

Tail call optimization does not need any special treatment. Indeed, activation annotations of tail calls simply never match as there is no corresponding location in the code.

In order to handle currying, the generated code uses the recorded number of pending arguments in order to find the location of the return address. Pending arguments are simply skipped, as if the callee was η-expanded according to the call. Following the same principle, arguments between the two subcontinuations belong to the tail. Therefore, the number of pending arguments has to be adjusted in subcontinuations like in Figure 6. In the head subcontinuation, the number of pending arguments in the stack frame is set to 0, as there is no pending argument below the stack frame. In the tail subcontinuation, the number of pending arguments on top of the stack comes from the popped stack frame (1 in the example).

As Marlow and Peyton-Jones [27] have previously noticed, the push / enter uncurrying technique is not the most favorable setup in order to walk the stack, which is what our `mc` operator achieves. More precisely, we remark that problems arise only when push / enter is combined with tail call optimization.

Assume the following code:

$$\begin{array}{l} \text{let } f_2 = \lambda a.\left(\text{capture}_{<l_5>} \text{up to } p \text{ with } v\right) \text{ in} \\ \text{let } f_1 = \lambda a.\lambda b.\,(<l_4>\ (<l_3>\ f_2\ x_3)\ x_4) \text{ in} \\ \text{let } x = <l_2>\ (<l_1>\ f_1\ x_1)\ x_2 \text{ in } e \end{array}$$

As uncurrying is done, l_1 and l_2 (resp. l_3 and l_4) refer to the same code location. They differ in the number of pending arguments above the return address, respectively 0 and 1. Due to tail call optimization, l_1 and l_3 (resp. l_2 and l_4) cannot be distinguished. Given the above description of the compiler, the captured continuation is split like in Figure 6. If the tail subcontinuation is subsequently compared to l_1, it matches as there is 1 pending argument. Our formal system assumes that the type of the produced head subcontinuation is $(\tau_{x_2} \to \tau_x) \xrightarrow{\kappa} \tau$. However, its effective (runtime) type is $(\tau_{x_4} \to \tau_x) \xrightarrow{\kappa} \tau$. The problem arises because, due to tail call optimization, there is no means at this point to know where the pending parameter comes from, i.e., to distinguish between l_1 and l_3.

Since our formal system does not implement uncurrying or tail-call optimization, it does not raises the problem, consistently with our type soundness result. Indeed, our formal system produces the following continuation, which is different from Figure 6:

$$\text{let } x = \text{frame}_{<l_2>,r,p}\ (<l_4>\ (\text{frame}_{<l_3>,r,p}\square)\ x_4) \text{ in } e$$

Notice that this continuation is actually the same as the one an eval / apply compiler would produce: as the arity of the f_2 closure is 1, l_4 is not applied and l_3 is not a tail call. In order to solve this problem in our prototype, we simply prevent uncurrying tail calls. Alternatively, we could have implemented the push / enter technique in our formal system, for instance extending our `frame` operator with pending arguments. We have identified the following options the `mc` operator can handle pending arguments:

- Pending arguments can go to the tail subcontination, as depicted in Figure 6 and described earlier in this section. Adding tail

```
1  prompt p →
2  (*****************************************)
3  (* Initial version                       *)
4  let rec fib n =
5    if n<2 then n
6    else (let fn1 = <L1>fib (n-1) in
7          let fn2 = <L2>fib (n-2) in
8            fn1+fn2) in
9  (*****************************************)
10 (* New version                           *)
11 let rec fib_num n =
12   if n<2 then num_of_int n
13   else (let fn1 = fib_num (n-1) in
14         let fn2 = fib_num (n-2) in
15           fn1+/fn2) in
16 (*****************************************)
17 (* update from fixed-size to arbitrary   *)
18 (* precision integer                     *)
19
20 (* if n is after 44, r has overflowed so  *)
21 (* return fib_new n else r is correct     *)
22 let ifnotover n r =
23   if n>44 then fib_num n else num_of_int r in
24
25 (* call graph:                           *)
26 (* fib    : L1  → fib                    *)
27 (* fib    : L2  → fib                    *)
28 (* [root]: Lroot → fib                   *)
29 (* _      : Lupdt → update               *)
30 (* to the fib node in the call graph:    *)
31 let rec match_fib_callers_ r k =
32   match_cont k with
33 (* - L1: r is fib (n-1)                  *)
34       <L1:n> :: tl →
35       let fn2 = fib_num (n-2) in
36         (* back to the caller: fib        *)
37         match_fib_callers_ (r+/fn2) tl
38 (* - L2: r is fib (n-2) & fn1 is fib (n-1) *)
39     | <L2:n fn1> :: tl →
40       (* check whether fn1 has overflowed  *)
41       let nfn1 = ifnotover (n-1) fn1 in
42         (* back to the caller: fib        *)
43         match_fib_callers_ (nfn1+/r) tl
44 (* - Lroot: r is the result of the program *)
45     | <Lroot> :: tl → reinstate tl r
46     | _  → (* error *) (0/-/1/) in
47 let match_fib_callers r k =
48   match_cont k with
49 (* - L1: r is fib (n-1)                  *)
50       <L1:n> :: tl →
51       (* check whether r has overflowed    *)
52       let fn1 = ifnotover (n-1) r in
53       let fn2 = fib_num (n-2) in
54         (* back to the caller: fib        *)
55         match_fib_callers_ (fn1+/fn2) tl
56 (* - L2: r is fib (n-2) & fn1 is fib (n-1) *)
57     | <L2:n fn1> :: tl →
58       (* check whether fn1 has overflowed  *)
59       let nfn1 = ifnotover (n-1) fn1 in
60       (* check whether r has overflowed    *)
61       let nfn2 = ifnotover (n-2) r in
62         (* back to the caller: fib        *)
63         match_fib_callers_ (nfn1+/nfn2) tl
64 (* - Lroot: r is the result of the program *)
65     | <Lroot> :: tl →
66         reinstate tl (ifnotover 12345 r)
67     | _  → (* error *) (0/-/1/) in
68
69 (* compensation fib → fib_num            *)
70 let compensate r k = match_cont k with
71         (* we "know" that we are in fib   *)
72       <Lupdt> :: tl → match_fib_callers r tl
73     | _  → (* error *) (0/-/2/) in
74
75 (*****************************************)
76 (* main program                          *)
77 (* register the compensation             *)
78 let _ = set_update_routine
79   (fun r → capture<Lupdt> upto p as k in
80              compensate r k) in
81 (* initial call                          *)
82   num_of_int (<Lroot>fib 12345)
```

Figure 7. Real ReCaml code: from fixed-size to arbitrary precision integers.

call optimization breaks type preservation for call site labels because each tail call can push new pending parameters with types different than those previously popped from the stack; and the types of pending arguments appear in label types. Hence we confirm what Marlow and Peyton-Jones [27] say with a stronger argument: push / enter with tail-call optimization and `mc`-like stack-walking is not type-sound.

- Pending arguments can go to the head subcontinuation. In this case, the type of the subcontinuations depends on how many arguments are pending on the stack. In the example, we would have to discriminate between l_5 / no pending argument, l_5 / 1 pending argument, and so on.
- Pending arguments can be dropped. In this case, part of the calculation captured by the original continuation is lost.

Nevertheless, notice that the updated program actually walks the stack. We feel that one of the weaknesses of our current approach is that our `mc` operator handles one stack frame independently of any context. We leave the issue for future works.

6. Detailed Example

Figure 7 contains the full source code that updates `fib` from `int` to `num`. The `set_update_routine` primitive (line 78) registers the function that is called when the virtual machine receives an update signal. In addition we use the following syntactic sugar:

line 1 `prompt p` $\rightarrow e$
$\hookrightarrow$ let p = newprompt in (setprompt$_{<>}$ p e)
line 79 capture<Lupdt> upto p as k in e
$\hookrightarrow$ capture$_{\texttt{\textless Lupdt\textgreater}}$ up to p with λk.e

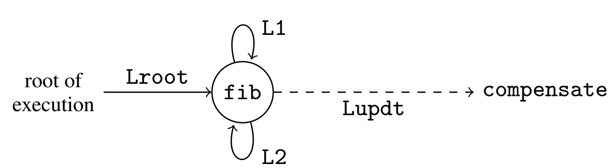

Figure 8. Static call graph of the program Figure 7.

The captured continuation corresponds to a path in the static call graph of the program (Figure 8) going from the root of execution to the compensation. The compensation is implemented by the `compensate` function (line 70). As represented by the dashed `Lupdt` edge in the call graph, the top stack frame is an activation of the anonymous function (line 79) registered by the `set_update_routine` primitive. It comes from the update infrastructure. Hence line 70, `match_cont` pops this useless stack frame before entering the effective compensation. In a more realistic application, we would have to find out which function the update is called from. In the example, as it can only be the `fib` function the compensation calls the `match_fib_callers` function (lines 47–67) to handle the calls to `fib` according to the strategy described in Section 2.2:

`L1` The compensation function receives the result of `fib (n−1)`. Using `ifnotover`, we ensure that it is correct (line 52). Notice that if the result has overflowed, the function `ifnotover` recomputes the Fibonacci number using the new version (line 23). To complete the `fib` function, we compute `fib (n−2)` with the new version (line 53) then we sum the two results (line 55). Last, we recursively compensate the tail of the continuation (line 55) as if the popped stack had returned the newly computed value.

`L2` The compensation function receives the result of `fib (n−2)`. Futhermore, the `match_cont` gets the value of `fib (n−1)` from the call stack frame naming it `fn1`. Using the `ifnotover` function, we ensure that those intermediate results are correct (lines 58–61). Last we complete the `fib` function and we recursively compensate the tail of the continuation (line 63).

`Lroot` At this point, `r` is `fib 12345` and the compensation has completed. We use `ifnotover` to ensure `r` is correct before reinstating the tail subcontinuation (line 66).

The `match_fib_callers_` function (lines 31–46) is almost a clone of `match_fib_callers`, except that it assumes the compensation has already dealt correctly with the received result (parameter `r`). So recursive calls in `match_fib_callers` do in fact switch to `match_fib_callers_`.

In these functions, we assume that (1) the evaluation order is known, i.e., that `fib (n-1)` is evaluated before `fib (n-2)`; and (2) intermediate results have names. To make this explicit, we use `let`. Instead, intermediate results could have had system-generated or *a posteriori* names. The evaluation order shall be inferred by the compensation.

Because we have not integrated any exception handling in our prototype, a negative number is returned (lines 46, 67 and 73) to notify errors. Runtime errors can occur if the continuation does not match, when the update developer forgets to handle some call sites.

7. Discussions and Conclusions

In this paper, we have presented two dynamic software updates (Sec. 2 – though only one example is detailed in Sec. 6 and Fig. 7) that many current systems are unable to implement. Even if we consider a toy example, we have argumented that the technique is still relevant in realistic applications. Despite the apparent simplicity of our use case, the two updates show high complexity both in design and in implementation. These examples contrast with the usual simple updates of complex applications in related works. In our work, we accept that updates might be difficult to design and implement. We have first focused in this paper on being able to achieve these updates. Still, we acknowledge that our current proposal is not very handy yet. In the context of a similar approach, Makris and Bazzi [25] have for instance proposed automatic generators for some of the updates, which could be used as building blocks for a higher level update language.

The ReCaml language is the cornerstone of our work. It provides an operator (`match_cont` or `mc`) in order to introspect and walk continuations. Our examples have indeed emphasized how this operation helps in updating. We have formalized its environment-based semantics and defined a type system whose soundness is proved mechanicaly. Even if we have not discussed it in this paper, we have also developed a sound substitution-based semantics. Our prototype compiler of ReCaml is able to execute all the updates of Section 2. The two examples of this article, the compiler and proofs (the coq scripts) can be found at `http://perso.telecom-bretagne.eu/fabiendagnat/recaml`.

In this paper, we have built ReCaml on top of a simply typed λ-calculus for simplicity reasons. It is well known that polymorphism with continuations needs restrictions in order to ensure soundness [5, 24, 35, 37]. As the `mc` operator splits continuations at activation boundaries, any type variable involved in an application might cause problems if it is generalized. One of the future challenges is therefore to reconcile ReCaml with polymorphism and to infer more precise types.

We have adopted a strict functional language and the ZAM2 virtual machine [23]. The ZAM2 machine has allowed us quick and easy prototyping. Strict evaluation has made it easier to understand and therefore to manipulate the execution state. Unlike similar approaches [20, 25], ReCaml does not require any specific code generation. Instead, relying on low level details of the underlying machine, it is adapted to the form of the code generated by the legacy Caml compiler. Using continuations is not a necessity. Yet it provides sound formal foundations for our work. As works that provide production level JVM and CLR with continuations [29, 32] use specific code generation, targeting such machines might not be in the scope of ReCaml. On the contrary, call site types are actually close to usual debug information. Therefore the debugging infrastructures of JVM and CLR could be used to implement ReCaml for these platforms. While these infrastructures provides mechanisms to manipulate states, ReCaml brings static typing. We therefore plan experiences to ensure that our approach also fits these platforms. To do so, we will have to enhance ReCaml to support imperative features, especially shared data. We will also have to consider multithreading, reusing previous work such as [25].

Acknowledgments

We would like to kindly thank Kristis Makris and Ralph Matthes for their comments. We also thank Michael Hicks for shepherding the revision of the paper. The work presented in this paper has been partly funded by the French ministry of research through the SPaCIFY consortium (ANR 06 TLOG 27).

References

[1] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: online patches and updates for security. In *USENIX Security Symposium*, pages 287–302, Baltimore, Maryland, USA, August 2005.

[2] Pascalin Amagbégnon, Loïc Besnard, and Paul Le Guernic. Implementation of the dataflow synchronous language SIGNAL. *ACM SIGPLAN Notices*, 30(6):163–173, June 1995. doi: 10.1145/223428.207134.

[3] Jonathan Appavoo, Kevin Hui, Craig Soules, Robert Wisniewski, Dilma Da Silva, Orran Krieger, Marc Auslander, David Edelsohn, Ben Gamsa, Gregory Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[4] Jeff Arnold and M. Frans Kaashoek. Ksplice: automatic rebootless kernel updates. In *European Conference on Computer Systems*, pages 187–198, Nuremberg, Germany, April 2009. doi: 10.1145/1519065.1519085.

[5] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *Asian Symposium on Programming Languages and Systems*, volume 4807 of *LNCS*, pages 239–254, Singapore, December 2007. doi: 10.1007/978-3-540-76637-7_16.

[6] Brian Aydemir, Aaron Bohannon, Benjamin Pierce, Jeffrey Vaughan, Dimitrios Vytiniotis, Stephanie Weirich, and Steve

Zdancewic. Using proof assistants for programming language research or, how to write your next popl paper in coq. `http://www.cis.upenn.edu/~plclub/popl08-tutorial/`, 2008. POPL 2008 tutorial.

[7] Andrew Baumann, Jonathan Appavoo, Robert Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware: challenges and solutions to updating an operating system on the fly. In *USENIX Annual Technical Conference*, Santa Clara, California, USA, June 2007.

[8] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for mashalling and update, with destruct-time λ. In *International Conference on Functional Programming*, pages 99–110, Uppsala, Sweden, August 2003. doi: 10.1145/944705.944715.

[9] Eric Bruneton, Thierry Coupaye, Matthieu Leclerq, Vivien Quéma, and Jean-Bernard Stefani. The Fractal component and its support in java. *Software: Practice & Experience, special issue on experiences with auto-adaptive and reconfigurable systems*, 36(11-12):1257–1284, September 2006. doi: 10.1002/spe.767.

[10] Jérémy Buisson and Fabien Dagnat. Introspecting continuations in order to update active code. In *Workshop on Hot Topics in Software Upgrades*, Nashville, Tennessee, USA, October 2008. doi: 10.1145/1490283.1490289.

[11] Acacio Cruz. Official Gmail Blog: Update on today's Gmail outage. `http://gmailblog.blogspot.com/2009/02/update-on-todays-gmail-outage.html`, February 2009.

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

[13] Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, November 2007. doi: 10.1017/S0956796807006259.

[14] 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`.

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

[16] Stephen Gilmore, Dilsun Kirli, and Christopher Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-379, University of Edinburgh, December 1997.

[17] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *International Conference on Functional Programming Languages and Computer Architecture*, pages 12–23, La Jolla, California, USA, June 1995. doi: 10.1145/224164.224173.

[18] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996. doi: 10.1109/32.485222.

[19] Jennifer Hamilton, Michael Magruder, James Hogg, William Evans, Vance Morrison, Lawrence Sullivan, Sean Trowbridge, Jason Zander, Ian Carmichael, Patrick Dussud, John Hamby, John Rivard, Li Zhang, Mario Chenier, Douglas Rosen, Steven Steiner, Peter Hallam, Brian Crawford, James Miller, Sam Spencer, and Habib Heydarian. Method and system for program editing and debugging in a common language runtime environment. Patent US7516441, Microsoft Corporation, April 2009.

[20] Christine Hofmeister and James Purtilo. Dynamic reconfiguration in distributed systems: adapting software modules for replacement. In *International Conference on Distributed Computing Systems*, pages 101–110, Pittsburgh, Pennsylvania, USA, May 1993. doi: 10.1109/ICDCS.1993.287718.

[21] Oleg Kiselyov. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Technical Report TR611, Indiana University, March 2005.

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

[23] Xavier Leroy. The ZINC experiment, an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[24] Xavier Leroy. Polymorphism by name for references and continuations. In *Principles of Programming Languages*, pages 220–231, Charleston, South Carolina, USA, January 1993. doi: 10.1145/158511.158632.

[25] Kristis Makris and Rida Bazzi. Multi-threaded dynamic software updates using stack reconstruction. In *USENIX Annual Technical Conference*, San Diego, California, USA, June 2009.

[26] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *European Conference on Computer Systems*, pages 327–340, Lisboa, Portugal, March 2007. doi: 10.1145/1272996.1273031.

[27] Simon Marlow and Simon Peyton-Jones. Making a fast curry: push/enter vs eval/apply for higher-order languages. *Journal of Functionnal Programming*, 16(4-5):415–449, July 2006. doi: 10.1017/S0956796806005995.

[28] Iulian Neamtiu, Micheal Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Conference on Programming Language Design and Implementation*, pages 72–83, Ottawa, Ontario, Canada, June 2006. doi: 10.1145/1133981.1133991.

[29] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *International Conference on Functional Programming*, pages 216–227, Tallinn, Estonia, September 2005. doi: 10.1145/1090189.1086393.

[30] Benjamin Pierce. Lambda, the ultimate TA: Using a proof assistant to teach programming language foundations, September 2009. Keynote address at *International Conference on Functional Programming*.

[31] Benjamin Pierce, Chris Casinghino, and Michael Greenberg. *Software foundations*. 2010. `http://www.cis.upenn.edu/~bcpierce/sf/`.

[32] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS transform. In *International Conference on Functional Programming*, Edinburgh, Scotland, UK, September 2009. doi: 10.1145/1596550.1596596.

[33] Peter Sewell, Gareth Stoyle, Michael Hicks, Gavin Bierman, and Keith Wansbrough. Dynamic rebinding for marshalling and update, via redex-time and destruct-time reduction. *Journal of Functional Programming*, 18(4):437–502, July 2008. doi: 10.1017/S0956796807006600.

[34] Chung-Chieh Shan. Shift to control. In *ACM SIGPLAN Scheme Workshop*, Snowbird, Utah, USA, September 2004.

[35] Mads Tofte. Type inference for polymorphic references. *Information and computation*, 89(1):1–34, November 1990. doi: 10.1016/0890-5401(90)90018-D.

[36] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo 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.

[37] Andrew Wright. Polymorphism for imperative languages without imperative types. Technical Report TR93-200, Rice University, February 1993.