Escaping with future variables in HALO

Charlotte Herzeel, Kris Gybels, and Pascal Costanza

Vrije Universiteit Brussel
{charlotte.herzeel|kris.gybels|pascal.costanza}@vub.ac.be

1 Introduction

HALO is a novel aspect language introducing a logic-based pointcut language which combines history-based pointcuts and "escape" conditions for interacting with the base language. This combination is difficult to support when escape conditions access context exposed by "future" join points. This paper introduces a weaving mechanism based on copying objects for resolving such pointcuts. Though this seems a memory consuming solution, it can be easily combined with HALO's analysis for reducing the join point history. Furthermore, pointcuts with escape conditions accessing future join point context, sometimes require *less* memory than pointcuts that don't, but otherwise implement the same functionality. In this paper, we illustrate this by measuring memory usage for simulations of an e-commerce application, switching between an implementation where the pointcut definitions contain escape conditions referring to future join point context, and an equivalent implementation that doesn't.

2 HALO by example

In this section we give a brief introduction to HALO. HALO is a novel logic-based aspect language for CLOS [1], extending our previous work on logic-based AOP [2] with support for history-based aspects.

As a running example, we use an e-commerce application. This application, implemented using the Hunchentoot web application framework, was reported on in earlier work [3] and is used in the experiments in Section 4.2. For explanation, we use the simplified version of this application as shown in Figure 1. The classes `Shop`, `User` and `Article` model the e-shop, its customers and the sold articles respectively. A class `Promotions` simply maps articles to a discount rate, which can be changed using the method `set-rate`, and accessed with `current-rate-for`. The method `singleton-instance` is used to retrieve the `Promotions` class' only instance.

HALO is a logic-based pointcut language, meaning pointcuts are expressed as logic queries over logic facts giving information about join points. Two features of HALO are important for this paper. Firstly, it is history-based, meaning the pointcuts are not just about the "current" join point, but also about past join points. Secondly, HALO is not purely logic-based: it features an "escape" mechanism which allows methods to be called from the logic pointcuts. Space

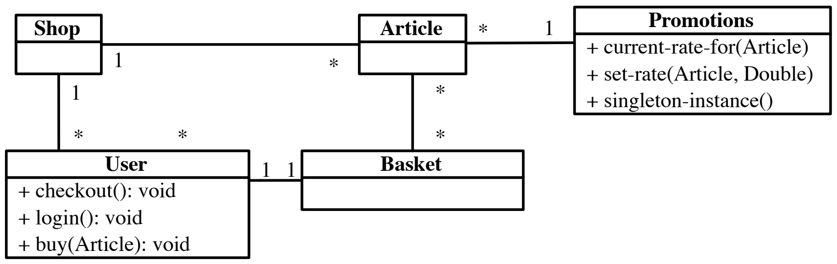


Fig. 1. Overview of the e-shop running example.

does not permit us to give a detailed discussion of HALO, we refer to earlier work for a lengthier explanation [3] , but here we illustrate HALO by means of a few advice definitions for the running example.

Figure 2 shows an example piece of advice using the two important features mentioned above. The piece of advice expresses that logging should happen when customers buy an article for which a promotion was advertised when the customer logged in to the e-shop. In more detail, the piece of advice consists of an advice body, which simply calls the log function, and a pointcut that specifies when to execute the advice body [1].

An advice body is executed each time the conditions expressed by the pointcut are satisfied by the current join point. For this example, this means the current join point must represent a call to the method named "buy" and *before* that call, a join point representing a call to the method "login" must have occured. This "before" is expressed by means of the temporal operator `most-recent`, which is one of the three built-in temporal connectives in HALO - the others are `all-past` and `since`. Also note the `escape` condition in the pointcut. Its second argument is a piece of Lisp code that accesses the discount rate `?rate` for an article `?article`.
Figure 2 also depicts a sample base program, to the right. It shows a user `<kris>` logging in and purchasing a `<cd>`[2]; In addition we see that the discount rate of the `<cd>` is changed from `10` to `5` percent. When the fourth statement is executed, all conditions in the pointcut are satisfied and the message "<kris> gets a 0.10 % discount" is logged. It's important to note that the discount rate will be the rate at the time `<kris>` logged in, not at the moment he buys the article – otherwise the `escape` condition should have been placed outside of the temporal operator. This is called *stateful* evaluation of pointcuts. This seems technically difficult, because the discount rate `?rate` needs to be computed when the `login` join point

[1] CLOS's and HALO's syntax are based on s-expressions: the generic function call `(buy r x y)` corresponds to `r.buy(x,y)` in Java, while the HALO logic condition `(gf-call 1 ?x)` corresponds to `gf-call(1, X)` in Prolog.

[2] The notation <*id*> is used to represent objects. E.g. `<kris>` is an instance of the class `User`, and `<promo>` of the class `Promotions`.

```
; advice
(at
; pointcut
  ((gf-call buy ?user ?article)
   (most-recent
     (gf-call login ?user)
     (escape ?promo (singleton-instance Promotions))
     (escape ?rate (current-rate-for ?promo ?article))))
; advice body
  (log "user ~s gets a ~s % discount" ?user ?rate))
```

```
1. (set-current-rate <promo> <cd> 0.10)
2. (login <kris>)
3. (set-current-rate <promo> <cd> 0.05)
4. (buy <kris> <cd>)
```

Fig. 2. HALO: combining temporal pointcuts with "escape".

occurs, but the article `?article` for which to compute the discount rate is only known at the "future" `buy` join point. The remainder of the paper explains how this works in HALO in more detail.

3 The HALO Weaver

In this section we first further discuss how the join point history is actually produced and stored. We discuss the details of the HALO weaver and the specific mechanism we use for matching history-based pointcuts combined with "escape" conditions referring to "future" join point data. This mechanism is based on our own extension of the well-known Rete algorithm [4].

3.1 Weaving Schema

In aspect-oriented programming (AOP), the process that is responsible for integrating aspects and base code, is called the "weaver", and this section sketches the basic workings of the HALO weaver. The HALO weaver is an extra layer on the CLOS compiler/interpreter for processing HALO code. A schema of the weaving process is depicted in Figure 3. The black boxes represent the different weaver components, whereas the transparent boxes reflect the CLOS and HALO code respectively. The arrows depict the flow of the weaving process. As the CLOS program runs, the weaver intercepts join points, and for each of these, a representation as a logic fact is recorded (in the figure, each statement of the CLOS program is mapped to such a logic fact). Each time a logic fact is added to the fact base, the "query engine" is triggered to find *solutions* for all the pointcuts; A solution for a pointcut consists of *bindings*, mapping all variables in the pointcut to a concrete value, and they are obtained by trying to pattern match the conditions of the pointcuts to the facts in the "fact base". Next, each pointcut solution is used to replace the logic variables in the advice code associated with the pointcut. As such, the advice code can be executed and inserted in the flow of the CLOS program.

In the actual implementation of the HALO weaver, the fact base and query engine are one component. More specifically, the HALO query matching process is based on the Rete forward chaining algorithm. A full motivation for this choice

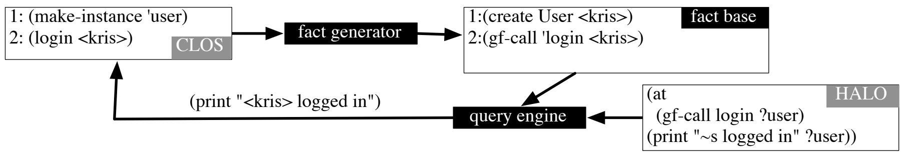


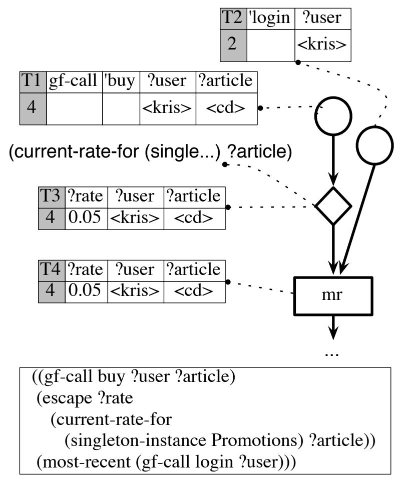
Fig. 3. HALO weaving schema

is outside the scope of this paper, but we do note that due to its pre-evaluation of partial queries and caching strategies, the Rete algorithm greatly simplifies the stateful evaluation of escape conditions, and that it has been shown to be a very efficient algorithm [4]. In the next section we discuss how (an extended version of) the Rete algorithm works for matching HALO pointcuts to join point facts.

3.2 Matching Pointcuts

The Rete algorithm [4] represents queries – or pointcuts in HALO– as a network consisting of nodes with memory tables. For each condition in a pointcut, the network contains a "filter" node. For each logical connective (the logical "and", `most-recent`, `since` or `all-past`), the network contains a "join node". As an example, consider the pointcut and Rete network in Figure 4. The circle-shaped filter nodes coincide with the two conditions in the pointcut, whereas the square-shaped join node reflects the `most-recent` operator. Note that each node is associated with a " memory table".

A full explanation of the Rete networks and our extensions are beyond the scope of this paper, but using the typical graphical representation of a Rete network (cfr. Figure 4) a basic understanding of its operation can be given as: facts are inserted in the "top" of the network, the filter nodes, and these facts "trickle down" the network. When the weaver generates a fact such as `(gf-call 'login <kris>)`, this is inserted in all filter nodes. A filter node for a condition such as `(gf-call 'login ?user)` checks that all the non-variable arguments of the fact and condition match. In this case, it checks that the name of the operation is `login` in both. It also binds the logical variable `?user` to the value `<kris>`. Two things then happen: an entry is made in a table, the so-called memory table of the node, to memorize this binding. The binding is also passed down to the next node in the network, a join node. Join nodes have two incoming nodes connected to them. Whenever they receive a new binding from one node, they combine these with the memorized bindings of the other node. This involves ensuring that the bindings have the same values for the same variables. If this is the case, they similarly make an entry in a memory table and pass the bindings to the next node. In our extension of Rete, the join nodes represent temporal conditions and also check these conditions before making new entries. We will



T2	'login	?user
2		<kris>

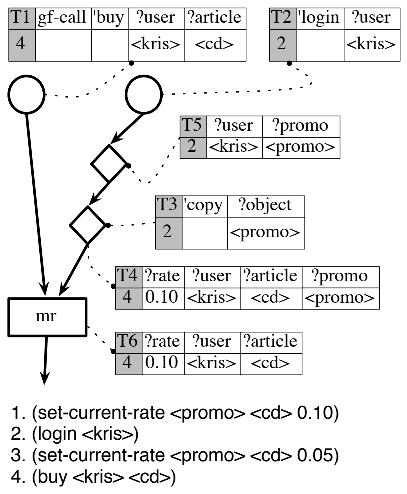
T1	gf-call	'buy	?user	?article
4			<kris>	<cd>

T3	?rate	?user	?article
4	0.05	<kris>	<cd>

T4	?rate	?user	?article
4	0.05	<kris>	<cd>

```
((gf-call buy ?user ?article)
 (escape ?rate
   (current-rate-for
     (singleton-instance Promotions) ?article))
 (most-recent (gf-call login ?user)))
```

Fig. 4. Rete representation of "escape".



T1	gf-call	'buy	?user	?article
4			<kris>	<cd>

T2	'login	?user
2		<kris>

T5	?user	?promo
2	<kris>	<promo>

T3	'copy	?object
2		<promo>

T4	?rate	?user	?article	?promo
4	0.10	<kris>	<cd>	<promo>

T6	?rate	?user	?article
4	0.10	<kris>	<cd>

1. (set-current-rate <promo> <cd> 0.10)
2. (login <kris>)
3. (set-current-rate <promo> <cd> 0.05)
4. (buy <kris> <cd>)

Fig. 5. Copying object state in Rete.

not delve into further depth here, but for example a node for a `most-recent` operator will check that only the combination using the most recently created matching bindings is made.

We focus in this paper on the "escape" feature of HALO, and our experiments with a number of different implementations of this feature. This is specifically related to allowing the use of "future" variables in `escape` conditions. The problem is that an `escape` condition such as `(escape ?rate (current-promotion-for ?promo ?article))` implies invoking the Lisp function `current-promotion-for`. For this to be possible, all of the logic variables used in the condition should have a value, but there can be conflicts between the availability of these values and the right time to evaluate the Lisp function. The different implementations define different variations of the HALO language and/or the weaver's operation.

No future variables This is the simplest HALO variation. In this version an `escape` condition that is inside a temporal operator is limited to using the following variables: variables used in logic conditions or as the result variable of other escape conditions, but only if these conditions are also inside the temporal operator (including any conditions in a nested temporal operator). This restriction disallows "future" variables: variables that are given values only by conditions outside the temporal operator. The semantics of `escape` conditions inside a temporal operator is that the Lisp function they invoke is invoked at the moment the logic conditions inside the same operator are matched. In the Rete networks for pointcut matching, an `escape` condition is therefore represented as a node taking as input one of the nodes representing the logic conditions, and is connected to the join node representing the temporal operator. In Figure 4, the diamond-shaped node represents the escape condition in the pointcut in the same figure.

Future variables, argument copying To increase the expressive power of HALO, we are investigating how the use of future variables in `escape` conditions can be allowed. The conflict this creates is that invocation of the Lisp function has to be postponed until the future variables are given a value. But at the same time, if the values of the other variables are objects, the state of these objects may be changed by the time the future variable is given a value. The result of the Lisp function may then not be the same as the case where it is invoked before these changes are made. Consider the pointcuts in Figure 4 and Figure 2: both capture an invocation of `buy` after an invocation of `login` and determine the discount rate of the bought article. But in the first case, this should be the rate at the time the buy invocation happens, and in the other at the time the `login` invocation happens. These can be different, as the rate can be changed between these invocations (using `set-rate` as explained in Section 2). In Figure 2, the `?article` variable is a future variable for the `escape` condition, because it will only be given a value outside the `most-recent` operator. The `escape` can only be evaluated when the `buy` happens, but of course, it should still return the rate at the time of `login`. To solve this, this variation of HALO takes copies of the arguments that will be passed to the Lisp function. In the example of Figure 2, the value of `?promo` will be copied when the `login` happens, the `escape` condition is evaluated when the `buy` happens, but with the copied state of the Promotions object and will thus return the right rate. Figure 5 displays the Rete network for the pointcut in Figure 2. In order to keep track of the object copies, the escape node is extended with an extra memory table.

Future variables, state change copying Argument copying only works when the invoked Lisp function only depends on its arguments, and not on global variables. A further variation of HALO takes this into account with a more extensive copying strategy. In this strategy, the state changes of *all* objects are intercepted, by making a copy of an object whenever the object is changed. Next the CLOS slot access protocol is extended to retrieve this copy whenever the object's field is read for evaluating the escape condition.

4 Optimizing Memory Usage in HALO

An apparent drawback of history-based AOP is exactly the need to store the history. A straightforward implementation of the weaver scheme in Figure 3 would mean a quickly growing history needs to be kept forever. Several techniques can be used to optimize this history. In this section, we first classify these techniques according to the kind of history information they remove. We first further detail how HALO deals with the last kind of information, and then explain the impact of the escape extension.

Irrelevant facts Some join point facts are not relevant to generate in the first place, because they will simply never be used to match pointcut definitions. For example, if there is only a pointcut capturing calls of the "buy" function and not of the "login" function, there is no need to generate join point facts for calls to the "login" function at all. This can be handled with static optimisation techniques, known as *shadow weaving* [5]. This is not currently done in HALO, but these techniques are orthogonal to those needed for the next two categories.

Facts relevant for one time step only Some information is only relevant for one step in the program execution. A pointcut that captures calls to "buy" if there was a past call to "login" with matching variables, only requires the weaver to store a history of "login" calls, but not of "buy" calls. In HALO, this is handled by the same technique as for the next category.

Facts that become irrelevant The last kind of information is the one more attention is paid to in HALO research. This is information that becomes irrelevant *after a while*. Suppose we have a pointcut that captures calls to "buy" with a certain user and article as argument, and which also gets the discount rate of that article at the *most recent* call to "login" for the same user. In this case, a history of "login" calls needs to be kept. But since only the most recent call for a particular user is accessed by the pointcut, parts of this history can be removed as the same user logs in again. This is handled in HALO by "memory table garbage collection" of the Rete nodes. The garbage collection is actually performed as part of the functionality of the nodes, there is not a separate algorithm like in memory garbage collection that intervenes once in a while. Though it can be turned on and off, and for explanative purposes we consider it separately from an explanation of node functionality.

4.1 Escape Nodes in Memory Table Garbage Collection

A full explanation of the memory table garbage collection is beyond the scope of this paper, but it can be illustrated with an example: consider the pointcut and Rete in Figure 6. The black tables labelled "LT", next to the memory tables, display the "life time" of memory table entries. The life time of a memory table entry consists of the time at which an entry was created and the time at which the entry can be removed. An entry can be removed as soon as it cannot be used anymore to derive new conclusions. The decision to remove a memory table entry depends on whether the node the entry belongs to is the input of a `most-recent` or `all-past` temporal join node. For example, the filter node labelled 1 in Figure 6 is not the right input of one of these types of temporal join nodes. This means that entries in this node will never be accessed to derive new conclusions after they're first inserted. The same is true for the nodes labelled 4, 5 and 6. As such, the life time of the entries residing at these nodes is constrained to one time step in Figure 6. If however the entry resides in a node that is the right input of a temporal join node, then it can only be removed if it is "replaced" by a new entry. For example, entries in the right input of `most-recent` join nodes can be removed when new entries with the same values for the memory table variables are added. In fact, only the values for the variables that are in common with the left input memory table need to be the same. This is because when an entry is added to the left input's memory table, the join node will combine it with the most recent matching entry in the right input node. The match requires that the values for the variables in common between the two input nodes are the same. Thus, if there is an older entry in the right memory table that also matches with the new entry in the left, it will still not produce a combination. Thus, such entries can be removed. E.g. in Figure 6, when the second entry in the node labeled 2 is added, the first entry is deleted, since it hase the same value `<kris>` for `?user`.

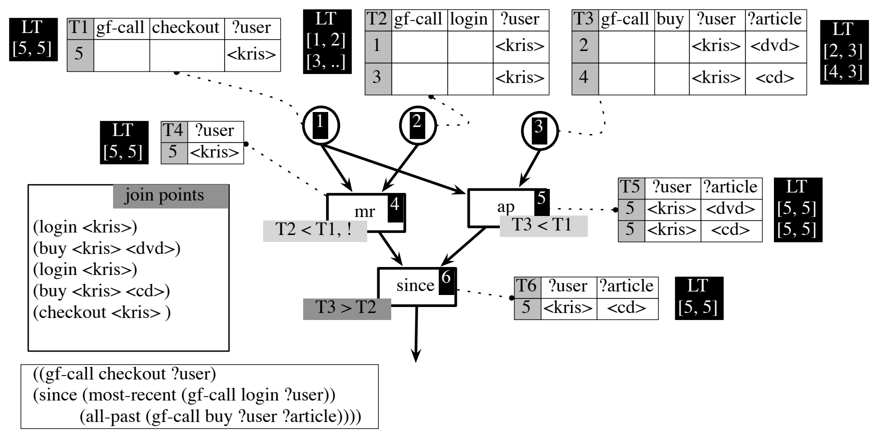


Fig. 6. Lifetime of memory table entries when garbage collected.

The above scheme was developed for the version of HALO with `escape` conditions that cannot refer to future variables. But this extension did not require a fundamental change to the above scheme. For `escape` nodes whose condition uses a future variable, a table needs to be kept with copies of the values of the bindings that the node takes as input, for later evaluation. The key point is however that these entries are completely linked to the entries of the `escape` node's input. Therefore, the entry in the `escape` node's memory can be removed when the parent entry in its input node is removed.

4.2 Benchmarks

We have evaluated the effect of the "escape" extension on memory usage using a few simulations. These were run on the e-commerce application implemented on top of the Hunchentoot web application framework [3]. The pointcuts used in this application can be written in different ways, with or without the use of future variables in `escape` conditions. To illustrate, consider for example the two pointcuts below. They both implement the same functionality, namely computing the discount rate active at login for an article bought at a later time. The first version however does not make use of a future variable. Instead, it is ensured that the `?article` variable already gets a value when the `buy` happens, by using an `all-past` to get all possible articles. In contrast, in the second pointcut in the code listing, the variable `?article` is a future variable. The Rete networks for matching both pointcuts are depicted in Figure 8.

```
((gf-call buy ?user ?article)
 (most-recent
   (gf-call login ?user)
   (escape ?promo (singleton-instance Promotions))
   (all-past (create article ?article))
   (escape ?rate (current-rate-for ?promo ?article))))

((gf-call buy ?user ?article)
 (most-recent
   (gf-call login ?user)
   (escape ?promo (singleton-instance Promotions))
   (escape ?rate (current-rate-for ?promo ?article))))
```

Each simulation defines a different number of articles, customers etc. and subsequently lets the customers randomly login, checkout their basket etc. for a number of times. Figure 7 depicts the total number of memory table entries made, and how many were already removed at the point the simulation stopped. In total, three different scripts were selected and run for two sets of aspects (see the labels $S1 - S3$ denoting the different scripts and (*non*)*copy* denoting a different set of aspects). Both sets of aspects implement the same functionality, but in one version the aspects are rewritten to make use of the extended version of "escape", referring to "future" join point data.

Figure 8 also displays a sample base program and the content of the memory tables obtained after executing this program. As is to be expected, the network for the first pointcut contains many more entries: each time a login occurs, the promotion rate for *all* articles is computed and cached. However in the second network, at each login, a copy of the `<promo>` object is cached. Note that the second network keeps track of much less memory table entries than the first. A surprising conclusion is therefore that if the memory cost for a copy of the Promotions object is lower than the memory cost for keeping the many entries in the first nework, then the escape condition accessing future join point data is woven *less* costly. We found this to be actually the case in our current shop application, as the Promotions object is implemented as a hash table mapping articles to discount rates. A copy of this table takes up less memory space than a Rete memory table holding the combinations of `?article`, `?user`, `?promo` and `?rate` as is the case for the highlighted node in Figure 8. Of course, this does not mean that this is a general conclusion about the Rete networks, as this depends on the specific application and its implementation.

Due to the fact that the HALO query engine records partial solutions to pointcuts, we need to take into account that additional memory is required for recording join point facts. However Figure 7 shows that the entries in the memory tables of join nodes, which represent the latter partial solutions, are also greatly reduced by the memory table garbage collection process (see second block for each simulation, where black is used to denote the remaining number of entries). Overall this means that for each join point fact that was or is memorized, the Rete network (temporarily) keeps track of partial solutions making use of it. However this is an improvement over recording all join point facts forever.

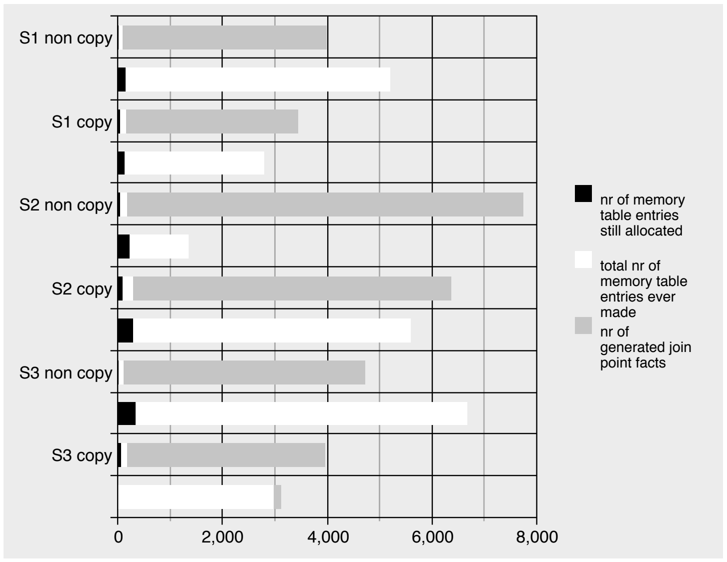
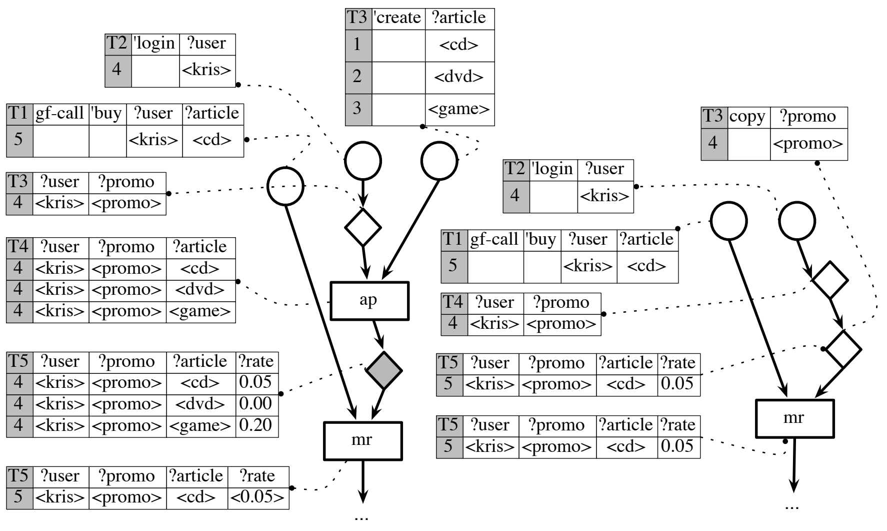


Fig. 7. Benchmarks for the memory table garbage collection.

5 Related Work

Alpha [6] is a Prolog-based aspect language (for a Java-like language) for expressing pointcuts over the join point history. The language provides a built-in set of temporal relations for comparing time stamps of join points, but the programmer can define new ones. Interaction with the base language is allowed, but only for the “current” join point. Alpha’s weaver performs a static analysis of source code to determine whether join point facts are necessary to generate for matching pointcuts, and at what time they can be removed from the join point history. Currently, this technique only removes join point facts when these are just used for matching pointcuts about the “current” join point. Other facts are kept indefinitely. Hence Alpha falls into the second category from Section 3.2.

Tracematches [7] is an AspectJ extension in which program traces can be formulated as regular expressions over “symbols”. These symbols are AspectJ



T2	'login	?user
4		<kris>

T3	'create	?article
1		<cd>
2		<dvd>
3		<game>

T1	gf-call	'buy	?user	?article
5			<kris>	<cd>

T3	?user	?promo
4	<kris>	<promo>

T4	?user	?promo	?article
4	<kris>	<promo>	<cd>
4	<kris>	<promo>	<dvd>
4	<kris>	<promo>	<game>

T5	?user	?promo	?article	?rate
4	<kris>	<promo>	<cd>	0.05
4	<kris>	<promo>	<dvd>	0.00
4	<kris>	<promo>	<game>	0.20

T5	?user	?promo	?article	?rate
5	<kris>	<promo>	<cd>	<0.05>

T3	copy	?promo
4		<promo>

T2	'login	?user
4		<kris>

T1	gf-call	'buy	?user	?article
5			<kris>	<cd>

T4	?user	?promo
4	<kris>	<promo>

T5	?user	?promo	?article	?rate
5	<kris>	<promo>	<cd>	0.05

T5	?user	?promo	?article	?rate
5	<kris>	<promo>	<cd>	0.05

Fig. 8. Rete for pointcut avoiding the use of future variables (left) and Rete for pointcut using future variable “?article” (right).

pointcuts containing “free variables”, referring to join point context. The base level can be accessed through the “let” construct, but currently it is only allowed to refer to join point context exposed by its enclosing symbol. The Tracematches weaver does shadow weaving and performs a three staged static analysis to reduce the set of join point shadows [7]. As such, this falls in the first category defined in Section 3.2. Shadow weaving is orthogonal to the dynamic analysis technique in HALO, and we plan to extend the HALO weaver with shadow weaving in the near future.

6 Conclusions and Future Work

In this paper we’ve given an initial discussion of the use of “future” variables in `escape` conditions in the logic pointcut language HALO, and the relation to HALO’s implementation using Rete networks, including how these fit into the memory table garbage collection scheme we previously developed for HALO without this feature. Supporting the feature involves copying objects to postpone the evaluation of these `escape` conditions. In the last part of the paper we presented benchmarks for the memory table garbage collection, showing the impact of the “escape” extension. Surprisingly, pointcuts written by means of this extension sometimes require less memory than their equivalents that don’t. Future work consists of investigating the question of whether the Rete networks can be designed so that these differences can be removed. For example, by postponing computation of memory table combinations. There's also a need to further study the different ways in which pointcuts with a same behavior can be implemented in HALO, and the impact on memory usage.

Acknowledgements

This work was supported in part by the AOSD-Europe Network of Excellence, European Union grant no. FP6-2003-IST-2-004349. Charlotte Herzeel is funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium. Pascal Costanza is funded by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

References

1. Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., Moon, D.: Common lisp object system specification. Lisp and Symbolic Computation **1**(3-4) (January 1989) 245–394
2. Gybels, K., Brichau, J.: Arranging language features for more robust pattern-based crosscuts. In: Proceedings of the Second International Conference on Aspect-Oriented Software Development. (2003)
3. Herzeel, C., Gybels, K., Costanza, P.: Modularizing crosscuts in an e-commerce application in lisp using halo. In: proceedings of the International Lisp Conference (ILC). (2007)
4. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence **19**(1) (September 1982) 17 – 37
5. Masuhara, H., Kiczales, G., Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In: Compiler Construction (CC2003). (2003)
6. Ostermann, K., Mezini, M., Bockisch, C.: Expressive pointcuts for increased modularity. In: European Conference on Object-Oriented Programming. (2005)
7. Bodden, E., Hendren, L., Lhoták, O.: A staged static program analysis to improve the performance of runtime monitoring. Technical report, ABC Group (2007)