Submitted to ICFP 2011

Experience Report: Functional Instrumentation of ActionScript Programs

Arie Middelkoop Alexander B. Elyasov Jurriaan Hage Wishnu Prasetya

Universiteit Utrecht

{ariem,elyasov,jur,wishnu}@cs.uu.nl

Abstract

In log-based testing, the system under test contains code to log aspects of the system's execution. Such code can be injected automatically with program transformations. Still, manual intervention is needed to identify execution steps of interest, as well as projections of the system's state. With aspect-oriented programming (AOP), we can describe the semi-automatic transformation of the system. However, AOP languages typically lack abstraction mechanisms. In this paper, we show that well-known techniques from functional programming facilitate the design of an expressive AOP EDSL for ActionScript, called Asil.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Tracing

General Terms measurement

Keywords instrumentation, execution traces

1. Introduction

The *future internet* [Hudson-Smith 2009] challenges traditional testing approaches, as future internet applications are a dynamic composition of ever-changing third party components and services. In addition to conventional unit and regression tests during development, continuous testing is required even after deployment, which can be accomplished through log-based testing. In log-based testing, the execution of the system under test (SUT) is logged and later analyzed to discover anomalies that require investigation by software testers.

In this paper, we use functional programming techniques for the design and implementation of a tool suite that facilitates the injection of logging code into client-side web applications. Our use case is Habbo Hotel [Sulake 2004], which is a large Flash application deployed on the Internet. Therefore, we restrict ourselves to programs written in ActionScript 3, the programming language of Flash. From the execution of this Flash program, we wish to obtain a log, which is an execution trace that records a projection of the program's state for each execution step.

To manually add logging to the source code of the SUT is undesirable, as it is a crosscutting concern that easily clutters up the original code. Moreover, modifications to the run-time environment are generally not possible due to security restrictions on the clients that run the application.

Fortunately, other techniques exist that help us deal with this issue more effectively. In particular, logging is a prime example of a functionality that can be implemented non-intrusively through aspect-oriented programming (AOP) [Kiczales et al. 1997]. This allows us to describe the instrumentation separately from the actual source code. In order to change the actual logging, we only need to change the instrumentation, and not the code to which instrumentation was applied.

In AOP, a *join point* is a location in the program's code to which we may transfer control. A join point has a static and dynamic context, which is defined as the reachable program state at that point. The *advice* is code that can inspect and alter this state when executed at a join point. Typical AOP languages, such as AspectJ [Kiczales et al. 2001], offer facilities for specifying *point cut*s: a particular set of join points, where a given piece of advice should be executed. An *aspect weaver* integrates the advice into the actual program at such join points via static and/or dynamic program transformation.

Unfortunately, typical AOP languages offer limited means of abstraction in the join point model. For example, in AspectJ, point cuts can only be parameterized over *symbolic values*. A symbolic value is a statically unknown value that corresponds to a *run-time value* in the state of the SUT during its execution. Due to our strongly functional background, where the use of higher-order functions is second nature, we would like point cuts to be first class in the AOP specification.

From a functional perspective, an AOP description can be viewed as a partial function that provides advice for some of the join points. In the case of logging, advice is a state transformer that additionally yields a sequence of events. Such an event typically contains some projection of the program's state, and an execution trace is then the sequence of such events as they arose during the SUT's execution.

$$\begin{aligned} \mathit{Spec} &= \mathit{Instrumentation} \\ \mathit{Instrumentation} &= \mathit{JoinPoint} \to \mathit{Maybe}\ \mathit{Advice} \\ \mathit{Advice} &= \mathit{World} \to (\mathit{World} \times [\mathit{Event}]) \end{aligned}$$

The advice functions are typically written in the programming language of the SUT. The instrumentation functions are point-cut descriptions in the AOP language, which is typically some pattern language.

For coverage analyses (which exploit the information contained in logs) we typically want to specify a more fine-grained instrumentation depending on, e.g., the nesting depth of branches, or annotations provided by the user. To log the behavior of external services, we often want to specify the instrumentation of the invocation of the service as a function of (part of) the specification of the external service. Therefore, in this paper, we deviate from standard

usage by parameterizing AOP specifications over values α. These may include symbolic values and even AOP specifications themselves.

$$Spec\ \alpha = \alpha \to Instrumentation$$
$$Coverage = Spec\ ControlFlowGraph$$
$$External\ = Spec\ ServiceContract$$

Consequently, such abstractions allow us to write more modular specifications.

From the point of view of the programmer this leads to a specification of the following type:

$$Spec\ \alpha = \alpha \times JoinPoint \times World \to Maybe\ (World \times [Event])$$

In other words, in an instrumentation specification a programmer may employ values of an arbitrary type Haskell α, a point cut, and information that will be only be available at run-time. For an example of the former, consider a control flow graph that we have previously computed for the SUT, and that we use to *generate* specifications of point cuts, and of advice to be weaved in. In other words, we can generate the AOP specification based on values and (higher-order) functions living in the Haskell world.

Although the specification by the programmer may refer to information that will be available at run-time, this information is not yet available during the early stages that transform Asil code (a DSL deeply embedded in Haskell) into AsilCore (our core instrumentation language), or that weave in the advice. From this point of view, the specifications are actually a bit more restrictive than the type suggests. The following type is a closer match.

$$Spec\ \alpha = \alpha \to JoinPoint \to Maybe\ (World \to (World \times [Event]))$$

This type makes explicit that run-time information, contained in values of type $World$, cannot be used during the first two stages of translation. Indeed, run-time information may only be used in ways that can be correctly translated into AsilCore or ActionScript. For example, in the specification given by the programmer, a value that will only be known at run-time may not be scrutinized in a case statement, because AsilCore does not have the concept of a case statement, and therefore it must be translated away. Of course, this restriction does not hold the other way around: instead of weaving in advice at join points in the point cut, one may decide to weave in advice at every join point that dynamically decides whether the joint point is an element of the point cut. For reasons of efficiency, however, this is not a very wise thing to do.

We point out that the "arrows" in the above type correspond exactly to code transformations: the first arrow transforms Asil into AsilCore, the second arrow is the process of weaving the advice into the ActionScript source (the tool *Asic*), and the final arrow (inside the $Maybe$) refers to the run-time execution of the advice.

Figure 1 Overview of the specification artifacts.

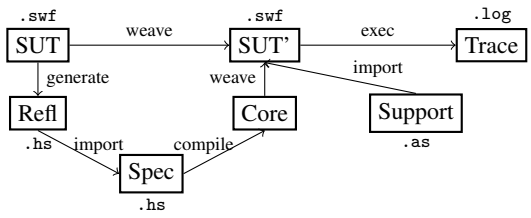

Figure 1 provides a somewhat more detailed view of the artifacts involved in the logging specification. From the SUT, we generate some reflection information Refl (symbol information, control flow graphs), which is imported by our logging specification Spec. From Spec, we generate AsilCore, which is weaved into the SUT together with the support library that provides functions for, e.g., serializing events. Finally, execution of the instrumented SUT generates the trace.

Zooming in on the languages we employ, our instrumentation language is a combinator language called *AsilCore*, that provides two primitive operations: pattern matching (on join points) and the invocation of advice. Such a language is not very easy to program in, which is why we have chosen to extend it to a deeply embedded Haskell DSL, called *Asil*. Asil uses the well-known concepts of *monadic* [Wadler 1995] and *alternative* [Mcbride and Paterson 2008] interfaces to compose instrumentations.

The implementation of Asil is ongoing work within the Fittest project[1]. The library `asil`[2] contains Asil as a library, including parsers, pretty printers, and other tools for the transformation of ActionScript byte code. The tool `asic`[3] generates reflection information and performs aspect weaving.

The remainder of this paper is organised as follows. After introducing the running example in Section 2, we consider the Asil language in more detail in Section 3. In Section 4 we show AsilCore and some aspects of its implementation; the aspect weaver asic is shortly discussed in Section 5. Finally, in Section 6 we reflect on the advantages and disadvantages offered by functional programming with respect to the implementation of Asil and Asic, and Section 7 concludes.

2. Example

In log-based testing, we log aspects of the program's execution, so that we can inspect the log to discover atypical execution patterns that may be worth an investigation by a software tester. This logging may take place in a special test environment, or after deployment, when the program is in active use.

In this paper, we consider only the instrumentation of Flash programs: the representation of the logs and the analysis of the logs are out of the scope of this paper. To set the scene, we describe a snippet of chess-like internet game as a simple running example (see Figure 2). The game proceeds in turns. In each turn, the user clicks on a square to select a pawn, then clicks on an unoccupied square to move the pawn there. Note that logging code has been explicitly inserted, logging each call to the `MyGame.clicked` method (with position information) and also the actual moves that are performed. The calls to `Log.clicked` and `Log.move` take care of generating events that are logged and thereby become part of the trace.

An *execution trace* is a sequence of events $e_1 \ldots e_n$, where an event $e = (t, i)$ is a tuple consisting of a discrete timestamp t and information i about the state of the program at t. In case of the example, this information consists either of a $clicked$ record or a $move$ record.

$$\begin{array}{lll} i ::= clicked\ (int, int) & & \text{-- clicked record} \\ \quad \mid move\ (code.Square, code.Square) & & \text{-- move record} \end{array}$$

The utility methods `Log.clicked` and `Log.move` take care of raising the events and serializing a projection of the object-graph of their parameters to a log. The actual implementation of these utility methods is out of the scope of this paper.

The calls to the logging methods obscure the source code. This gets worse when we make the logging code conditional for reasons of, e.g., privacy or performance:

```
if (Level.privacy <= 2 && Level.verbose >= 1) {
```

[1] `http://www.pros.upv.es/fittest/`

[2] `http://hackage.haskell.org/package/asil`

[3] `http://hackage.haskell.org/package/asic`

Figure 2 ActionScript snippet with logging code.

```
package code {
public class MyGame extends Sprite {
  private var selSquare : Square;

  public function MyGame() : void {
    addEventListener("click", clicked); }

  function clicked(event:MouseEvent) : void {
    var x : int = event.localX;
    var y : int = event.localY;

    Log.clicked(x,y);
    var target : Square  = getSquare(x,y);
    var taken  : Boolean = occupied(target);

    if (!this.selSquare && taken) {
      this.selSquare = target;
    } else if (this.selSquare && !taken) {
      Log.move(this.selSquare, target);
      this.move(this.selSquare, target);
      this.selSquare = null; } } } }
```

```
    Log.move(this.selSquare, target) }
```

Logging and privacy are typical examples of crosscutting concerns, so that we can apply AOP techniques to specify these concerns separately from the code. In the next section we discuss the instrumentation language Asil, a functional AOP language.

3. The Asil Instrumentation Language

The language $Asil$ is a combinator language for writing down specifications $Spec\ \alpha$ as described in Section 1. It is a functional, strongly-typed, monadic Haskell EDSL for expressing *advice* that is conditionally applied at *join points*. Join points are predefined locations in the code where control can be transferred to. Figure 3 lists the join points in our *join point model* [Cazzola 2006]. Our main join points are the conventional method entry and exit from the caller and callee side as in AspectJ [Kiczales et al. 2001], as well as sequential blocks of instructions [Juarez-Martinez and Olmedo-Aguirre 2008]. Operationally, an Asil program is a value of the type $I\ ()$ (explained below) that is applied to each encountered join point during execution.

Figure 3 Join points and their context.

Join point	Context captured
Method entry	name, parameters, and param types
Method exit	name, return value, and return type
Method abort	nane, and exception
Method call	name, parameters, and param types
Method returned	name, return value, and return type
Method failed	name, and exception
Block entry	id, cycle root, and preceding join points
Block exit	id, cycle root, and preceding join points
Coercion call	input, input type, and intended type
Coercion returned	input, output, input type, and output type
Coercion failed	input, exception, and intended type

We use combinators to construct monadic *instrumentations* of type $I\ a$ and (symbolic) *expressions* of type $E\ a$. A typed product a of expressions is represented by the type $\overline{E}\ a$. Figure 4 lists the Asil combinators that are used in the example.

Instrumentations have a notion of success: a successfully applied instrumentation of type $I\ a$ returns a value of type a, otherwise the instrumentation failed to apply. The operations $matchEnter$, $matchCall$, and $matchEnter'$ express matches against join points. A successful match returns join-point information. For example, after $matchEnter$ we can access the values of parameters to the call (symbolically). The operation $call$ invokes an ActionScript method, and returns the (symbolic) result value that is returned by that method. Method calls succeed by default, unless the method uses a special API routine to indicate an abort.

It is important to realize that the values that will only be available at run-time will only be represented symbolically: we can manipulate them only to a limited extent. Such values are the atomic elements of our *expressions* $E\ a$ that may also involve built-in operations on these values, such asinteger arithmetic, comparisons, boolean arithmetic, field dereference, and array indexing. Essentially, a value of type $E\ a$ represents a (symbolic) ActionScript value of type a. We use Haskell to build such expressions. Not all Haskell expressions (lambdas in particular) can masquerade as E-expressions, nor can E-expressions be scrutinized with Haskell's case expressions. The reason is that AsilCore does not support these types and expressions, and we chose not to provide translations for them at this time. We may do so in the future, but in this first attempt we prefer to introduce only those elements that are essential and to decide later what else we might need. Only a few Haskell *values*, such as integers and strings, are also E-expressions.

Figure 4 Excerpt from the Asil combinators.

```
    -- instrumentations (I)
matchEnter  :: Prop c (Method a b) → I (Enter a)
matchCall   :: Prop c (Method a b) → I (Enter a)
matchEnter' :: I (Enter Any)
call        :: E (Method a b) → E‾ a → I (E b)
guard       :: E Bool → I ()
onPrevious  :: I a → I a
param       :: Enter Any → Int → I (E Any)
    -- monadic combinators
(>>=)       :: I a → (a → I b) → I b
return      :: a → I a
fail        :: String → I a
    -- alternative combinators
(⊕)         :: I a → I a → I a
(⊘)         :: I a → I a → I a
(⊗)         :: I a → I a → I a
    -- expression combinators (E)
(#)         :: E c → Prop c t → E t
embed       :: I a → E a
null        :: E (Object t)
static      :: E Static
param1      :: Enter (a, r)      → E a
param2      :: Enter (a, (b, r)) → E b
    -- match context
data Enter a = ME {name :: E String, params :: E‾ a}
```

The primitive operations are composed with *monadic* and *alternative* combinators, shown at the bottom of Figure 4. The monadic bind $m \gg\!= f$ composes instrumentations sequentially, and parameterizes the continuation f with the values returned by m, if m succeeds. The monadic return $return\ x$ is an always succeeding instrumentation that is effectively a no-op, but returns x as value. We use do-notation to conveniently write monadic sequences.

In the following example, the operation $matchEnter$ matches against the method entry (join) point of the `clicked` method of

`MyGame`, and then calls the `clicked` method `Log` with information that is extracted from the event-parameter (using #).

```
instrLogClick = do
  m ← matchEnter k_code_MyGame_clicked
  let evt = param_1 m
      eX = evt # k_flash_events_MouseEvent_localX
      eY = evt # k_flash_events_MouseEvent_localY
  call (static # k_code_Log_clicked) (eX, eY)
  return ()
```

If the match succeeds, then the variable m contains information about the joint point. Different types of join points typically provide different information. In this case, we can use $param_1$ to obtain a typed reference evt to the first parameter of the method from m. From evt we can obtain the coordinates of the mouse click and pass these to the *Log_clicked* method. For static properties, we use the special name *static* as object reference.

On the other hand, $matchEnter'$ can match against method-entry join points that do not have a statically fixed name nor a statically fixed signature. For example, with the following code, we can match on any method-entry join point with a name that contains `"clicked"` as substring, and has a first parameter of the `MouseEvent` type.

```
do m   ← matchEnter'
   guard (count (params m) ≡ 1)
   guard (substr "clicked" (name m))
   p_1 ← param m 1
   evt ← p_1 `cast` t_flash_events_MouseEvent
   ...
```

In this example, *params* obtains a reference to the parameters of the method from m. The *guard* instrumentation succeeds if and only if its Boolean parameter is *True*. Remember that these operators work on symbolic values $E\ t$ instead of conventional Haskell values of type t.

Similarly, we can instrument the call to `move` in the method `clicked`. The operation *onPrevious* takes an instrumentation p and applies it to the joint points that (directly) precede in the control flow graph, the matched join point (ignoring back edges). The operation succeeds if p is successfully applied to such a preceding join point (at runtime). With this operation, we declaratively specify contextual pattern matches.

```
instrLogMove = do
  onPrevious $ matchEnter
    k_code_MyGame_clicked
  m ← matchCall k_code_MyGame_move
  let from = param_1 m
      to   = param_2 m
  call (class # k_code_Log_move) (from, to)
  return ()
```

In the context of having matched against `clicked`, we match against a call to `MyGame.move`, and insert a call to `Log.move`.

The instrumentations *instrLogMove* and *instrLogClick* are defined separately. Instrumentations can be combined with various *alternative* operators. Given some continuation f, the meaning of $(p \otimes q) \gg\!= f$ is that branch $p \gg\!= f$ and branch $q \gg\!= f$ are applied to the join point (in that order). The meaning of $(p \oplus q) \gg\!= f$ is that p and q (in that order) are both applied, and the continuation f is parametrized with the result of the last succeeding one of the two. The strict alternative $p \mathbin{①} q$ applies q only if p fails.

The ability to fail raises the question how we deal with the fact that our support library contains conventional ActionScript methods that may exhibit side effects. At this time, we do not attempt to revert these effects.

With conventional higher-order functions and parallel composition we combine the individual instrumentation.

```
myInstr :: I ()
myInstr = foldr (⊗) (fail "initial")
  [instrLogClick, instrLogMove]
```

The *fail* operation is a unit of parallel composition.

Having specified the complete instrumentation *myInstr* we can then interpret it for each join point in the Flash program according to the semantics that we specified informally in this section. However, we actually implemented an instrumentation monad such that its execution yields an AsilCore specification (Section 4) as an intermediate step. How and why we do this is the subject of the next section.

4. The AsilCore Language

The monadic bind is an obstacle in the translation of Asil to ActionScript. The right-hand side of a bind is a function, thus it would require a nontrivial translation of Haskell functions to ActionScript. Therefore, we partially evaluate the monadic code to AsilCore, which is sequential code. This step eliminates both functions and monads.

Figure 5 AsilCore abstract syntax.

```
i  ::= match m                          -- match against join point m
     | call o v̄ (x :: τ)                -- call with args v̄, binds result x
     | on previous i                    -- lift to preceeding join points
     | abort                            -- aborts instrumentation
     | nop                              -- no-op
     | coerce v_1 (x_2 :: τ)            -- coerce type of v_1 to τ
     | i_1 ; i_2                        -- sequential composition
     | i_1 ⊕ i_2                        -- parallel composition
     | i_1 ① i_2                        -- alternative composition
     | static i                         -- error if i not statically resolved
     | dynamic i                        -- defers eval of prims in i
m  ::= entry (x_1 :: String) x̄_2 :: τ̄   -- name x_1, args x̄_2
     | exit  (x_1 :: String) (x_2 :: τ) -- name x_1, result x_2
     | call  (x_1 :: String) x̄_2 :: τ̄   -- name x_1, args x̄_2
     | ...                              -- etc.
o  ::= prim x                           -- primitive with the name x
     | static s                         -- static property named s
     | dynamic v_1 v_2                  -- call closure v_1 with v_2 as this
v  ::= x | c                            -- identifiers and constants
```

The AsilCore language (Figure 5) is an intermediate language that can be mapped strightforwardly to ActionScript advice (to be precise, we map directly to AVM2 byte code). An AsilCore term has a statically fixed control flow graph, which simplifies the injection into compiled ActionScript, since it can be represented with conventional branching instructions of ActionScript.

In the translation to AsilCore (Figure 6), we implement the instrumentation monad in continuation passing style [Sussman 1975] to deal with the $\otimes$ operator. The monadic binds are replaced with static sequencing. Even if Asil programs cannot use Haskell code to scrutinize E-values, we can parametrize a function f in the right-hand side of a bind with the (typically) symbolic value v_1 computed in the left-hand side, and continue partial evaluation. As a result, such a value v_1 will be restricted in how it can be manipulated at run-time.

We desugar Asil programs a bit further to simplify the AsilCore representation. We express all branching with the local combinators

Figure 6 Asil to Core execution.

```
type I a = Uid →                                   -- unique id
  (Uid → a → Core → (Core, Uid)) →                 -- continuation
    (Core, Uid))                                   -- res. core
run f = fst $ f 1 (λn _ i → (i, n))                -- I a → Core
instance Functor I where
  fmap f g = λn k → g n (λm v → k m (f v))
instance Monad I where
  return x  = λn k → k n x nop
  m1 >>= f  = λn1 k → m1 n1 $ λn2 v1 i1 →
                (f v1) n2 $ λn3 v2 i2 → k n3 v2 (i1; i2)
  fail s    = λn k → k n (error s) abort
instance Alternative I where
  empty     =  λn1 k → k n1 ⊥ abort
  m1 ⊕ m2 =  λn1 k → m1 n1 $ λn2 v1 i1 →
               m2 n2 $ λn3 v2 i2 → k n3 v2 (i1 ⊕ i2)
  m1 ⊗ m2 =  λn1 k → let (i1, n2) = m1 n1 k
                         (i2, n3) = m2 n2 k
                     in (i1 ⊕ i2, n3)
embed i = λn k → k n () i                          -- Core → I ()
fresh     = λn k → k (n + 1) (ESym n) nop          -- I (E a)
matchBlockEntry = do                               -- I Block
  vId ← fresh
  embed (match block guid (toCoreIdent vId))
  return $ Block {blockGuid = vId}
```

⊕ and ⓪. Values in AsilCore are either identifiers that symbolically represent ActionScript objects, or constants that represent concrete ActionScript objects. Matches introduce identifiers for the contextual values of the match, and calls introduce identifiers for the result value. Calls take values as argument, which are either constants or identifiers. An identifier may only be introduced once, and must be introduced before being used. This property holds for AsilCore programs that are derived from well-typed Asil programs.

The following fragment shows the result of the translation of the running (Asil) example to AsilCore. The abstractions offered by Asil are seen to be expanded away.

```
{match entry
    name x1    :: String
    args x2    :: flash.events.MouseEvent
; call prim equals
    args x1, "code.MyGame:clicked"
    res  x3    :: Boolean
; call prim guard
    args x3
    res  x4    :: void
; call prim deref
    args x2, "flash.events.MouseEvent:localX"
    res  x5    :: int
; call prim deref
    args x2, "flash.events.MouseEvent:localY"
    res  x6    :: int
; call static "code.Log:clicked"
    args x5, x6
    res  x7    :: void
} ⊕   -- composes the two instrumentations in parallel
{on previous
    {match entry
        name x8   :: String
        inputs x9  :: flash.events.MouseEvent
    ; call prim equals
        args x8, "code.MyGame:clicked"
        res  x10 :: Boolean
    ; call prim guard
        args x10
        res  x11 :: void}
; match call
    name x12 :: String
    inputs x13 :: code.Square, x14 :: code.Square
; call prim equals
    args x12, "code.MyGame:move"
    res  x15 :: Boolean
; call prim guard
    args x15
    res  x16 :: void
; call static "code.Log:move"
    args x13, x14
    res  x17 :: void
} ⊕ abort   -- may be optimized away
```

The primitive *deref* takes an object and a fully qualified name as string, and returns the associated property if it exists, otherwise it fails. To call a property of an object, *deref* can be used to obtain a closure of type *Function*, followed by a **call dynamic**. The primitive *guard* succeeds if and only if its argument is true, and the *equals* primitive has the same semantics as the ActionScript `==`.

The declarative **on previous** instruction lifts its block to all preceding join points in the static control flow graph of the method. Operationally, it also introduces an additional local boolean that keeps track whether the instrumentation was applied. If it was applied, we can access its values via the local variables that the lifted instrumentation introduced.

Figure 7 AsilCore to ActionScript translation.

```
⟦abort⟧                  ⇝ thisJoinPoint.aborted = true
⟦nop⟧                    ⇝ ;
⟦call o v̄ (x :: τ)       ⇝ ⟦x⟧ = (τ) (⟦o⟧⟦v⟧);
⟦match exit (x1 :: String) (x2 :: τ2)⟧ ⇝
  thisJoinPoint.aborted ∨= thisJoinPoint.isMethodExit;
  if (¬ thisJoinPoint.aborted) {
    ⟦x1⟧ = thisJoinPoint.nm; ⟦x2⟧ = thisJoinPoint.ret}
⟦i1; i2⟧     ⇝ ⟦i1⟧; if (¬ thisJoinPoint.aborted) {⟦i2⟧}
⟦i1 ⊕ i2⟧    ⇝ ⟦i1⟧; thisJoinPoint.aborted = false; ⟦i2⟧
⟦i1 ⓪ i2⟧    ⇝ ⟦i1⟧; if (thisJoinPoint.aborted) {
                 thisJoinPoint.aborted = false; ⟦i2⟧}
⟦prim x⟧        as   ⇝ Primitives.⟦x⟧(as)
⟦static C : x⟧  as   ⇝ ⟦C⟧.⟦x⟧(as)
⟦dynamic v1 v2⟧ as   ⇝ v1.call (v2, as)
```

The obtained AsilCore instrumentation can be mapped to ActionScript code as defined in Figure 7, and can then be weaved in at join points in the program. This requires an ActionScript method for each primitive operation, and we need to construct a reflexive `thisJoinPoint` value [Hilsdale and Hugunin 2004] at each join point. For a typical join point, only a small part of the instrumentation is applicable (or even none at all). Therefore, our instrumentation weaver Asic (Section 5) partially evaluates the AsilCore term, and only weaves in the residual term.

5. The Instrumentation Weaver Asic

The Asic tool takes an AsilCore term and applies it to all join points of the SUT. This involves yet another partial evaluation step. The success of a match is always resolved statically, and results in static bindings for identifiers related to the name of a method, unique id of a block, and possibly also the types and number of arguments. Calls to primitive methods may be performed statically, depending on what is known statically about their parameters. Reachable calls to non-primitive methods always result in a residual call in order to cater for possible side effect.

AsilCore terms can be large when the instrumentation is *generated* from, e.g., the control-flow graph. For example, there may be a dedicated AsilCore branch for each method in the SUT. Asic applies several optimizations to the AsilCore term before and after the evaluation process. For example, for each AsilCore branch, we collect the static constraints on the join points in a trie structure, such that we can quickly obtain the branches that potentially match.

Aspect-weaving is an instance of syntax-directed translation. Attribute Grammars (AGs) [Swierstra et al. 1998] provide a declarative language to write composable implementations of such translations. Hence, we implemented Asic with AGs. An advantage of the approach, compared to a monadic one, is that aspects of the translation can be described seperately, e.g., the threading of environments, location information and substitutions.

6. Reflection

Embedding. Haskell provides Asil's abstraction facilities for free. Also, we piggy back on Haskell's type system to type check Asil specifications. For the syntactic sugar related to E-expressions and function calls we used GADTs and common type class extensions. The use of type classes was not always intuitive due to confusing conflicts between overlapped instances and functional dependencies.

In our embedding, join-point structures and the SUT's runtime values are represented by E-expressions. The syntax of E-expressions is purposefully limited so that we can use Haskell's abstraction facilities, but refrain from embedding arbitrary Haskell expressions in AsilCore and ActionScript.

Executable specifications. Specifications of Flash and the ActionScript bytecode format are publicly available. However, we discovered that these specifications are incomplete with respect to new Flash versions, and also contain errors that are hard to track down. Therefore, we initially implemented a monadic binary parser with `uuparsing-lib` [Swierstra 2009], which provides results lazily, in combination with error correction. This provided us with sufficient context to find bugs in our own implementation and the specification itself. Our parser can actually serve as a formal specification, in an executable, testable form.

Performance. Our use case, the Habbo application, is a large SWF file that consists of 3,000 classes with 25,000 methods and 800,000 instructions. Simply parsing the application takes several seconds, which is surprisingly long despite the size of the application. We reimplemented our parser as a binary deserializer using the `Data.Binary.Get` to no avail. Profiling showed us that garbage collection takes about 60% of the time. Our hypothesis is that this is an interplay between the large and relatively long lived AST that we construct, and the short-lived closure-garbage produced by the parsers. If it would be possible to allocate the AST (via an annotation on the data constructors) directly in an older memory generation, we expect it to improve our parser's performance. A modification of the parser that only verified the structure of the SWF files but does not construct the AST takes about half the time and seems to confirm our hypothesis.

A more pressing problem is that GHC crashes on the huge symbol file that we generate from the Habbo SWF. We consider generating a separate Haskell module for each ActionScript class.

Proofs. The structure of Asil is based on the functor, monad, applicative and alternative interfaces [Mcbride and Paterson 2008]. This allows us to immediately use many of the convenient functions that are written in terms of these interfaces. Also, these interfaces have to satisfy satisfy a number of laws. Proving that these laws hold gives us more confidence and insight in our implementation. For example, our monad (Figure 6) satisfies the monad laws semantically, but not structurally, since the monadic $return$ introduces no-ops. In our actual implementation, we directly eliminate superfluous no-ops and trivially-dead branches as optimization.

7. Conclusion

We presented Asil, an expressive AOP EDSL for the instrumentation of ActionScript programs. Its design and implementation is based on well-known functional programming techniques, such as monads. The implementation exploits Haskell's facilities for abstraction, syntactic sugar and static typing.

Like AspectJ, Asil allows point cuts to be specified with constrained patterns. In addition, Asil allows these constraints to be specified as runtime ActionScript methods and possibly instrumentation-time primitive functions. Finally, (higher-order) functions can be used to abstract over instrumentations and to derive instrumentations from others.

The possibility to write higher-order functions is an essential feature that we sorely miss in many DSLs. Fortunately, this comes for free for EDSLs in functional languages.

References

W. Cazzola. Semantic Join Point Models: Motivations, Notions and Requirements. In *SPLAT06*, 2006.

E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *AOSD'04*, pages 26–35, 2004.

A. Hudson-Smith. The future internet. *Future Internet*, 1(1):1–2, 2009.

U. Juarez-Martinez and J. O. Olmedo-Aguirre. A Join-point Model for Fine-grained Aspects. In *ECC'08*, pages 126–131, 2008.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97*, pages 220–242, 1997.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP'01*, pages 327–353, 2001.

C. Mcbride and R. Paterson. Applicative Programming with Effects. *JFP*, 18:1–13, 2008.

Sulake. Habbo Hotel, 2004. `http://www.habbo.com/`.

G. J. Sussman. Scheme: An Interpreter for Extended Lambda Calculus. In *Memo 349, MIT AI Lab*, 1975.

S. D. Swierstra. Combinator Parsing: A Short Tutorial. In *Language Engineering and Rigorous Software Development*, pages 252–300, 2009.

S. D. Swierstra et al. UU Attribute Grammar System. `http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem`, 1998.

P. Wadler. Monads for Functional Programming. In *AFP'95*, pages 24–52, 1995.