Effect Handlers All the Way Down

BRIAN CAMPBELL, University of Edinburgh, United Kingdom
SAM LINDLEY, University of Edinburgh, United Kingdom
WILMER RICCIOTTI, University of Edinburgh, United Kingdom
IAN STARK, University of Edinburgh, United Kingdom

Effect handlers are a popular programming language idiom providing a unified interface to a range of features: exceptions, state, I/O, concurrency, dynamic binding, and more. Effect handlers are typically introduced in high-level languages and implemented through existing lower-level constructions: by contrast, in this work we introduce AsmFX, an abstract assembly language where a handful of instructions directly express the processor-level control-flow requirements for effectful programming. We demonstrate the expressiveness of AsmFX with a compiler from a functional core calculus that generalises all of deep, shallow, and parameterised handlers with multi-shot resumptions. A key contribution is our proof of compiler correctness, obtained by enriching the source program with annotations that track the behaviour of compiled code fragments.

1 Introduction

Effect handlers are a powerful and elegant programming language abstraction providing facilities to define and control a wide range of program behaviours classed as "side effects". Operationally, they are similar to resumable exception handlers, but they can also be used to model state, I/O, asynchronous programming, concurrency, dynamic binding, and probabilistic programming [6, 8, 9]. Effect handlers are particularly helpful in achieving clean separation of concerns and modularity and have recently been implemented in many research and mainstream languages. Implementations of effect handlers in high-level languages often convert them to existing features like closures, continuations, or prompts [15, 20, 28, 30]; while low-level approaches, on the other hand, often manipulate an existing stack [2, 12, 25]

Our approach in this work is different: instead of translating effect handlers away into other constructs, we provide a foundation for studying their direct implementation by propagating them all the way down to an abstract instruction set architecture with a primitive notion of effect handlers. In particular, we want this language to allow reasoning about effect handlers and concrete low-level implementations of them in the same calculus. We do not know of existing studies of a direct instruction-set representation of effect handlers.

As a starting point for our investigation, we define SFX, a core calculus with effect handlers to act as our high-level effectful source language (Section 3): its design goal is to provide a sufficiently rich effect-handling infrastructure, where resumptions may be single-shot, affine, or multi-shot, where the traditional setting of *deep* handlers can be extended by *parameterised* handlers, or replaceable handlers (such as the *sheep* handlers of WAsmFX [25]). The source language is thus capable of expressing a large range of effects, including but not limited to state, exceptions, nondeterminism, and dynamic scoping. Following prior work [16, 17], we express the semantics of SFX via a CEK-like abstract machine (Section 4).

We then introduce AsmFX (Section 5), an abstract register-based instruction set architecture with primitives to define, install, and invoke effect handlers. The goal of this ISA is to support all forms of effect handler expressible in SFX (and possibly more). Concretely, AsmFX adds to a traditional register-based architecture seven additional instructions, four of which allow the extended control-flow behaviour of effectful programs; the remaining three implement additional load operations

Authors' Contact Information: Brian Campbell, University of Edinburgh, United Kingdom, Brian.Campbell@ed.ac.uk; Sam Lindley, University of Edinburgh, United Kingdom, Sam.Lindley@ed.ac.uk; Wilmer Ricciotti, University of Edinburgh, United Kingdom, research@wilmer-ricciotti.net; Ian Stark, University of Edinburgh, United Kingdom, Ian.Stark@ed.ac.uk.

needed for creating special data structures used by AsmFX for representing continuations, handlers, and external code pointers as first-class values. As a reasoning tool, AsmFX does not prescribe particular memory layouts or restrict the amount of data that can be stored in registers. The semantics of the ISA is expressed by means of an abstract machine, which uses, along with main memory, an independent stack-like data structure describing the current effect handler state.

AsmFX provides an interesting computational model in its own right and can be used to study effectful programs written directly in a suitable ISA or as the intermediate language of a compiler for high-level languages. In Section 6 we describe a compiler for SFX targeting AsmFX. The compiler yields both AsmFX assembly code and an augmented version of the original source code annotated to denote which AsmFX code addresses implement each program expression. The augmented (ASFX) code is crucial for stating and proving our main correctness theorems.

We conclude our investigation of the expressive power of AsmFX with a proof of correctness of the SFX compiler, described in Section 7. Due to fundamental differences between declarative and procedural languages, proving directly that compiled AsmFX source code simulates corresponding SFX source code directly is particularly challenging. Instead, we split our proof into two parts, using ASFX as an intermediate step. ASFX provides a generalisation of both SFX and AsmFX by tracking intermediate AsmFX machine states directly in (an extension of) SFX: specifically, ASFX extends the SFX notion of continuation with *epilogues*, which denote fragments of machine code that have no direct representation in the source, but must be executed nonetheless. To our knowledge, this proof technique, and in particular our use of epilogues, is novel. The proof constitutes a key technical contribution of our work.

2 Overview

In order for us to understand the requirements of an assembly-level implementation of effect handlers, we will present a few examples in high-level pseudo-code, highlighting a few different uses of handlers and what we would expect their compiled version to look like.

Before we begin, let us establish terminology for the various layers of an effectful program: we distinguish *external code*, which is any part of the program which may install handlers but does not necessarily run in the context of a handler, and *client code*, which is called by the external code after installing a handler, and will return to the external code that invoked it after terminating normally; *handler code* includes globally defined handlers in the form **handler** h {**return** $x \to M \mid \#t(y) \to (r).N \mid \ldots$}, consisting of a *return clause* whose purpose is to post-process the result of a client computation before returning to the external code, and zero or more *operation clauses* expressing the semantics of effectful operations such as $\#t(y)$, where $\#t$ is an operation tag, and y its formal argument, which may resume the client code that invoked the effect by means of a resumption r, or return control to the external code.

An effectful assembly language should provide instructions to transition between these code layers, installing and uninstalling effect handlers as needed.

2.1 Runners

One of the simplest applications of effect handlers is the implementation of *runners* [1]: the functionality of runners corresponds to handlers in which a resumption may only be invoked once by means of a tail call. To illustrate this functionality, we consider an example in which a certain program needs to track a limited resource budget by means of a handler. The handler stores its budget within a mutable variable *budget*.

var *budget* := 100

handler *track_budget* {

```
    return x → return ⟨x, budget⟩
  | #charge(n) → (r).
    if (budget ≥ n) then budget := budget − n; resume r(⟨⟩)
    else return ⟨0, −1⟩
}
let ⟨x, balance⟩ ←
  handle {
    let _ ← do #charge(10) in
    let a ← p() in let b ← q() in return a + b
  } with track_budget
in print_int(balance)
```

The program uses the **handle** keyword to execute client code within the context of the handler *track_budget*, providing access to an effect #*charge*(*n*). The client code charges the budget for an amount of 10, then runs two subroutines $p()$ and $q()$ which may charge the budget as well; finally it sums the results of the two subroutines and returns. The handler *track_budget* provides a return clause that pairs the result x of the client code with the remaining budget; the operation clause for #*charge* performs different actions depending on the available budget: if there is enough budget to grant the request, it subtracts the requested value from the variable *budget* and then resumes the client code; otherwise, it aborts the client code and returns $\langle 0, -1 \rangle$, where 0 is a dummy result, and the negative balance -1 signals to the external code that the client failed to complete.

At an assembly level, to implement the code above, we can assume that the client code is at a memory location denoted by label `budget_client`, and that the handler is implemented by clauses identified by labels `track_budget.ret` and `track_budget.charge`. The client would use instructions `do` and `return` respectively to perform an effect and to return to the external code. Registers `$a0`, `$a1`, ... are used to exchange arguments/results between function and operation calls.

```
budget_client:
  $a0 := 10
  do #charge
  ; [...] code to call p()
  $t0 := $a0       ; save result of p() to scratch register
  ; [...] code to call q()
  $a0 := $t0 + $a0 ; add result of q() to previously saved result
  return
```

To execute this code under the handler *track_budget*, our assembly language provides an instruction `resume` to run a continuation under a handler, and instructions `loadk` and `loadh` to load continuations and handlers. To tell the machine where to jump once the handler terminates correctly, we will use instruction `loadl`, initialised to the address of the instruction after `resume`. The external code would therefore be able to run `budget_client` by means of the following code:

```
  loadk budget_client
  loadh (return=track_budget.ret, #charge=track_budget.charge)
  loadl budget_client_done
  resume
budget_client_done:
```

The `loadk` instruction creates a continuation invoking the code at label `budget_client`; similarly, `loadh` creates a handler from its return and operation clauses; and `loadl` a *leave record* pointing to code label `budget_client_done`. The continuation, the handler, and the leave record may be held in distinguished registers `$kr`, `$hr`, and `$lr`.

The `track_budget.ret` clause needs to leave the client result in `$a0` untouched, and copy the current value of `budget` into `$a1`, then yield control to the external code by means of a suitable instruction `exit` (operationally, this will jump to the code address specified by register `$lr`).

```
track_budget.ret:
  $a1 := mem[budget]
  exit
```

When running the `track_budget.charge` clause, the machine automatically loads the current handler, leave record, and the continuation to the client into the corresponding registers: this allows the client to be resumed by a simple `resume` instruction, without any additional `loadh`, `loadl`, or `loadk` instructions. To abort the client and yield control to the external code, we may be able to use the same `exit` instruction used in the return clause:

```
track_budget.charge:
  $t0 := mem[budget]
  if ($t0 < $a0) branch abort
  $t0 := $t0 - $a0
  mem[budget] := $t0
  resume
abort:
  $a0 := 0
  $a1 := -1
  exit
```

2.2 Parameterised handlers

The budget example we gave uses a mutable global variable to store the budget balance; while this variable should only be used by the handler, there is no way to prevent unauthorised code from altering it. A common idiom with effect handlers is to represent the private state of a handler as a *parameter*, which is initialised when the handler is installed and can be updated before invoking a client resumption. By using parameterised handlers, the budget example can be rewritten as follows:

```
handler track_budget(budget) {
      return x → return ⟨x, budget⟩
  | #charge(n) → (r).
      if (budget ≥ n) then resume r(⟨⟩) with track_budget(budget − n)
      else return ⟨0, −1⟩
}
let ⟨x, balance⟩ ←
  handle {
    let _ ← do #charge(10) in
    let a ← p() in let b ← q() in return a + b
  } with track_budget(100)
in print_int(balance)
```

We would like to support parameterised handlers at the assembly level, using registers to pass the parameters. The `loadh` instruction can accept an additional `state` operand declaring a list of registers containing the handler state that must be saved. For instance, the external code would load the *track_budget* handler with an initial budget of 100 as follows:

```
$s0 := 100
loadh (return=track_budget.ret, #charge=track_budget.charge),state:($s0)
```

This ensures that whenever the handler clauses are run, the register `$s0` contains the same value it contained when the `loadh` instruction was executed. Similarly, the `track_budget.charge` code, before resuming the client, would update the state and refresh the handler:

```
track_budget.charge:
  ; $s0 contains the current budget
  if ($s0 < $a0) branch abort
  $s0 := $s0 - $a0
  loadh (return=track_budget.ret, #charge=track_budget.charge),state:($s0)
  resume
  ; ...
```

Refreshing the handler is crucial to inform the machine that the updated content of `$s0` should be used the next time the handler code is run.

2.3 Producer/consumer

In some cases, when implementing an effect, we may want to resume a client under a different handler from the one that was originally invoked. Consider the following example consisting of two functions *prod* and *cons*: the first uses the *#send* operation to send two integer values over a shared channel, while the second invokes a *#recv* operation to get two integer values from the same channel and return their sum.

```
// Producer sends a couple of values and then stops
fun prod(_) {
  do #send(1); do #send(2); return ⟨⟩
}

// Consumer receives a couple of values and then returns their total
fun cons(_) {
  let x ← do #recv(⟨⟩) in let y ← do #recv(⟨⟩) in return x + y
}
```

The two operations are implemented in two different handlers. The producer handler *ph* parameterised over a consumer resumption *cr* handles *#send* by resuming *cr* under the consumer handler *ch*, which is passed as a parameter the producer resumption *pr*:

```
handler ph(cr) {
    return ⟨⟩ → handle prod(⟨⟩) with ph(cr)
  | #send(x) → (pr).let r ← (cr with ch(pr)) in resume r(x)
}
```

Symmetrically, the consumer handler *ch*, parameterised over a producer resumption *pr*, handles *#recv* by resuming *pr* under the producer handler *ph*, which is passed as a parameter the consumer resumption *cr*:

```
handler ch(pr) {
    return result → return result
  | #recv(_) → (cr).let r ← (pr with ph(cr)) in resume r(⟨⟩)
}
```

The main program is started by handling the consumer *cons* under an appropriate handler: ideally, this would be *ch*, but the producer has not started yet, so we do not have a producer resumption to pass as a parameter yet; instead, we use an initial consumer handler *ih* which handles the *#recv* operation by starting *prod* rather than calling a producer resumption.

```
// Program starts the consumer under an initial handler
fun main(_) {
```

```
  handle cons(⟨⟩) with ih(⟨⟩)
}

// An initial version of the consumer handler, which starts the producer rather than resuming it
handler ih(_) {
    return x → return x
  | #recv(_) → (cr).handle prod(⟨⟩) with ph(cr)
}
```

When control is transferred to an operation clause, the resumption received by that clause would normally continue execution under the handler defining that operation, according to *deep handler* semantics. However, one can view a resumption as a pair consisting of a head continuation and of the handler delimiting it, and allow the handler to be replaced. The **resume** keyword works by calling the continuation after reinstalling the delimiting handler. The **with** keyword allows us to replace the delimiting handler with a handler of our choice, recovering the semantics of *sheep handlers* [25]. At the machine level, a resumption can be represented as a continuation value and a handler value (which need to be loaded into the registers `$kr` and `$hr` to be executed), and the handler replacement functionality of **with** can be expressed by means of the `loadh` instruction. For example, we could implement the *#send* clause of handler *ph* by means of the following assembly:

```
ph.send:
  ; handler parameter cr (consumer resumption) in ($s0, $s1)
  ; value to be sent in $a0, producer resumption pr in ($kr, $hr)
  $t0 := $kr
  $t1 := $hr
  $kr := $s0 ; continuation of cr
  $s0 := $t0 ; continuation of pr
  $s1 := $t1 ; delimiting handler of pr
  ; load $hr with ch(pr)
  loadh (return=ch.ret,#recv=ch.recv),state:($s0,$s1)
  ; resume (cr with ch(pr))
  resume
```

2.4 Multi-shot resumptions

Our final example illustrates more complex handlers by evaluating propositional formulas under a non-deterministic choice of variables. We use a Boolean valued effect *#flip* to generate values for the propositional variables, then combine them with standard Boolean operators to evaluate the formula of interest. The tautology $(A \wedge B) \vee \neg A \vee \neg B$ can then be expressed by the following code:

```
fun prop(_) {
  let a ← do #flip(⟨⟩) in
  let b ← do #flip(⟨⟩) in
  return (a && b) || not a || not b
}
```

The value of the propositional variables depends on the handler implementing the *#flip* effect. We can have trivial handlers always returning **true** or **false**, or randomly returning either value (given a source of randomness). Most interestingly, we are not limited to one truth value, but we may see what happens when the propositional variables are assigned all possible combinations of truth values. The following *tautology* handler implements *#flip* by resuming the (Boolean-valued) client twice, once with **true** and once with **false**, and then taking the conjunction of the results of the two resumption calls. The final result will be **true** if and only if the client evaluates to **true** for all the truth assignments of the propositional variables, i.e. if the client represents a tautology.

```
handler tautology(_) {
```

```
      return x → return x
  | #flip(⟨⟩) → (r).
      let x ← resume r(false) in
      let y ← resume r(true) in
      return x && y
}
handle prop(⟨⟩) with tautology(⟨⟩)
```

This example differs from the previous ones in two ways: first, the resumption is invoked more than once; and second, the resumption call is not the last action in the operation clause — it is not a tail call. This means that when the resumption (really, the client computation) completes its execution, it must not return to the external code, but to the operation clause that invoked it; in other words, when performing a non-tail resumption, the handler code that invoked the resumption becomes the external code with respect to the client code it resumed.

An assembly-level machine needs to be made aware of this fact to correctly implement the control flow. In AsmFX we have allocated a distinguished register `$lr` to contain the address of the code that needs to be executed after a client completes its work. Note that the handler code invoking the resumption will want to preserve some registers, and in particular the original value of `$lr`, which still contains the address the handler must jump to upon termination: these registers can be specified as an additional `save` parameter to `loadl`, and will be restored to their original value when an `exit` instruction is performed.

We can then implement the *#flip* operation as follows:

```
tautology.flip:
  $a0 := 0    ; false
  loadl l1, save:($lr)
  resume      ; will jump to l1 when the resumption ends
l1:
  ; $lr restored to original value
  $t0 := $a0  ; save result
  $a0 := 1    ; true
  loadl l2, save:($lr,$t0)
  resume      ; will jump to l2 when the resumption ends
l2:
  ; $lr and $t0 restored to the original value
  $a0 := $t0 && $a0
  exit        ; return r(false) && r(true)
```

3 A source calculus for effect handlers

The first step towards our goal of studying the compilation of effect handlers is to introduce a core declarative language expressive enough to easily implement the idioms we are interested in. For our purposes, it makes sense to restrict ourselves to programs using only first order global functions, since techniques to convert nested functions to global definitions are well established (see for instance [18, 23]) and we do not intend to discuss them in this work; we do however support simple and mutual recursion. For similar reasons, we will use globally defined handlers. Based on these requirements, we define the source calculus SFX, whose syntax is given by the following grammar (where we have highlighted non-standard expressions).

Value types	A, B	::=	$\tau \mid \langle \overline{A} \rangle \mid A \xrightarrow{C} D$	Effect types	E	::=	$\{\overline{\#t : A \to B}\}$
Comp. types	C, D	::=	$A!E$	Top-level types	G	::=	$A \Rightarrow C \mid (A)C \Rightarrow D$

Open values	U, V, W	::=	$x \mid c \mid \langle \overline{V} \rangle$
Computations	M, N	::=	$\mathbf{return}\ V \mid \oplus(V) \mid f(V) \mid \mathbf{if}(V, M, N)$
		$\mid$	$\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \mid \mathbf{unpack}\ \overline{x} \leftarrow V\ \mathbf{in}\ M$
		$\mid$	$\mathbf{do}\ \#t(V) \mid \mathbf{handle}\ M\ \mathbf{with}\ h(V)$
		$\mid$	$\mathbf{newbroom}(f)\ \mathbf{with}\ h(V) \mid U\ \mathbf{with}\ h(V) \mid \mathbf{resume}\ U(V)$
Literal values	u, v, w	::=	$c \mid \langle \overline{v} \rangle \mid (\kappa, h(v))$
Environments	γ	::=	$\overline{x \mapsto v}$
Continuations	κ, κ'	::=	$\mathbf{0} \mid (x).M[\gamma] \cdot \kappa \mid h(v) \cdot \kappa$
Functions	F	::=	$\mathbf{fun}\ f(x)\ \{M\}$
Handlers	H	::=	$\mathbf{handler}\ h(x)\ \{\mathbf{return}\ y \to M \mid \overline{\#t(z) \to (r).N}\}$
Specifications	Σ	::=	$\overline{F}; \overline{H}$
Programs	P	::=	$\Sigma; M$

As syntactic sugar, we use a vector notation such as $\overline{x_n}$ to indicate the sequence $x_0 \ldots x_{n-1}$; the same notation, without subscript, indicates a sequence whose length is unspecified because it is irrelevant or can be inferred from the context.

SFX is a fine-grain call-by-value source language with a basic effect type system; terms fall into two categories: *values*, representing what the program computes, and *computations* describing how values are computed. Value types A comprise primitive types τ, n-ary tuples of value types $\langle \overline{A} \rangle$, and the novel *broom* type $A \xrightarrow{C} D$ generalising functions to resumptions, where a type A argument produces a computation of type C under a handler giving a result of type D. Computation types $C, D = A!E$ combine a return type A with an effect type E specifying the effects that computations of this type may invoke. Effect types assign signatures $A \to B$ to a finite number of operation tags $\#t$, where A is its input type and B is its output type. If E is empty then by abuse of notation we write computation type $A!\{\}$ simply as A. Handlers are *parameterised* [19, 26], so a handler type is of the form $(A)C \Rightarrow D$ where A is the type of the parameter, C is the type of the computation being handled, and D is the type of the resulting computation. With resumptions and handlers limited to a single argument parameter we use tuples to used to express n-ary versions.

Value terms U, V, W comprise variables x, constants c, and n-tuples: these are open, since they may contain variables that reference values stored in an environment. We distinguish them from *literal* values u, v, w, which are closed, and beside constants and tuples can also be resumptions consisting of a continuation paired with a distinguished handler in the form $(\kappa, h(v))$ — we call these *broom* resumptions, or just *brooms*, because like the tool used for sweeping they consist of a head κ and a handle $h(v)$. Programs can only refer explicitly to value terms: in particular, broom values cannot appear in a program, except as variables of continuation type. At runtime, value terms will evaluate to literal values.

Besides returning a value, primitive operations, conditionals, sequencing (**let**), tuple decomposition (**unpack**), function application ($f(V)$, where f is the name of a globally defined function), we provide computation terms $\mathbf{do}\ \#t(V)$ to perform an effect $\#t$ with input V. Computations can also handle effects with the syntax $\mathbf{handle}\ M\ \mathbf{with}\ h(V)$, where M is the client computation, h the name of a globally defined handler, and V its actual parameter, representing its initial state.

Our reason for the "broom" terminology is to manage all of deep, shallow, and sheep handlers: a broom resumption has a head (the continuation) and a handle (its effect handler); and in some cases either component may be replaced to refresh the broom. Resuming a broom invokes the continuation and also installs the handler: syntax $\mathbf{resume}\ U(V)$ invokes broom U with argument V. Brooms most often represent a suspended computation that has requested performance of an effect, being passed to the corresponding operation clause of the closest matching handler installed. It is

also possible to convert a function f into a broom by means of the syntax **newbroom**(f) **with** $h(V)$: this broom will take an argument as input, install the handler $h(V)$, and then apply f to the argument received. Fitting a broom U with a new handle $h(V)$ is written as U **with** $h(V)$, allowing us to implement sheep handlers.

Top-level handlers **handler** $h(x)$ $\{$**return** $y \to M \mid \overline{\#t(z) \to (r).N}\}$ accept a parameter x and provide operation clauses for each effect $\#t$ they handle: the effect receives an input argument z, and additionally a broom r that allows the handling code to resume execution within the client, by *resuming* r with a value which will appear, to the client, as the result of evaluating the effect $\#t(z)$; the handler also provides a return clause receiving in the variable y the value computed by the client. Given such a handler, we will use the following metalinguistic operations to refer to its bound variables: $dom(h)$ for the parameter variable x, $dom(h_{\mathbf{return}})$ and $dom(h_{\#t})$ for the formal arguments of the return and operation clauses (respectively, y and z), and $res(h_{\#t})$ for the continuation argument r of the operation clause. The syntax to define top-level functions is self-explanatory.

Continuations κ are sequences of continuation frames: the base case $\mathbf{0}$ expresses program termination. Pure continuation frames $(x).M[\gamma]$ consist of a closure $M[\gamma]$ abstracted over a variable x: given a literal value, say v, returned by a computation and stored in variable x, the pure frame continues as M, where free occurrences of x have been replaced by v, and any other free variable is defined by γ; pure computations are used in the intermediate steps of sequencing. Handler continuation frames $h(v)$ consist of a handler identified by its name h, applied to a literal value v representing its state; when a computation is running in the context of such a handler frame, the effects defined by h are available, and if that computation returns a value u, u is passed to the **return** clause of h.

Programs consist of any number of mutually defined handlers and functions plus a main computation that may use them.

3.1 Typing

Typing judgements for the source calculus include four main forms for values, computations, continuations, and handlers/functions. We also provide two minor judgements for environments and for the program. Judgements that depend on the specification Σ are annotated with Σ on the turnstile ($\vdash_\Sigma$). However, since Σ does not change in recursive typing assumptions, we will often assume it is fixed for a given program and omit it from the subscript. Figure 1 presents the typing rules.

Value typing. Value typing $\Gamma \vdash_\Sigma V : A$ or $\Gamma \vdash_\Sigma v : A$ assigns a value type A to an open value V or to a literal value v (even though we distinguish the two cases syntactically, they are typed under the same judgement and the same rules) under a typing context Γ associating variables to their value type. The interesting rule T-Broom says that a broom $(\kappa, h(v))$ of type $A \xrightarrow{C} D$ consists of a continuation head κ taking a value of type A and returning a computation of type C, and a handle which is its delimiting handler h of type $(B)C \Rightarrow D$ instantiated with a value v of type B: $h(v)$ must transform the computation type C returned by κ to another computation type D.

Computation typing. Computation typing uses the judgement $\Gamma \vdash_\Sigma M : C$. The interesting rules are those for invoking and handling effects. An effectful operation **do** $\#t(V)$ (rule T-Do) has type $B!E$ if the effect type E assigns the type $A \to B$ to $\#t$ and V has type A. Rule T-Handle assigns the type D to **handle** M **with** $h(V)$ when M has type C, the handler's parameter V has type A, and that h is defined in the signature as a handler of type $(A)C \Rightarrow D$ linking all three together.

$$\textsc{T-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash_\Sigma x : A} \qquad \textsc{T-Val}\ \frac{c \in \mathrm{Val}(\tau)}{\Gamma \vdash_\Sigma c : \tau} \qquad \textsc{T-Tuple}\ \frac{\Gamma \vdash_\Sigma V : A}{\Gamma \vdash_\Sigma \langle \overline{V} \rangle : \langle \overline{A} \rangle} \qquad \textsc{T-Prim}\ \frac{\Gamma \vdash_\Sigma V : A \qquad \mathrm{Type}(\oplus) = A \to B}{\Gamma \vdash_\Sigma \oplus(V) : B}$$

$$\textsc{T-Broom}\ \frac{\vdash_\Sigma \kappa : A \to C \qquad \Gamma \vdash_\Sigma v : B \qquad \Sigma \vdash h : (B)C \Rightarrow D}{\Gamma \vdash_\Sigma (\kappa, h(v)) : A \xrightarrow{C} D} \qquad \textsc{T-Ret}\ \frac{\Gamma \vdash_\Sigma V : A}{\Gamma \vdash_\Sigma \mathbf{return}\ V : A!E} \qquad \textsc{T-App}\ \frac{\Gamma \vdash V : A \qquad \Sigma \vdash f : A \Rightarrow C}{\Gamma \vdash f(V) : C}$$

$$\textsc{T-If}\ \frac{\Gamma \vdash_\Sigma V : \mathrm{Bool} \qquad \Gamma \vdash_\Sigma M : C \qquad \Gamma \vdash_\Sigma N : C}{\Gamma \vdash_\Sigma \mathbf{if}(V, M, N) : C} \qquad \textsc{T-Let}\ \frac{\Gamma \vdash_\Sigma M : A!E \qquad \Gamma, x : A \vdash_\Sigma N : B!E}{\Gamma \vdash_\Sigma \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : B!E}$$

$$\textsc{T-Unpack}\ \frac{\Gamma \vdash_\Sigma V : \langle \overline{A} \rangle \qquad \Gamma, \overline{x : A} \vdash_\Sigma M : C}{\Gamma \vdash_\Sigma \mathbf{unpack}\ \overline{x} \leftarrow V\ \mathbf{in}\ M : C}$$

$$\textsc{T-Do}\ \frac{\Gamma \vdash_\Sigma V : A \qquad E(\#t) = A \to B}{\Gamma \vdash_\Sigma \mathbf{do}\ \#t(V) : B!E} \qquad \textsc{T-Handle}\ \frac{\Gamma \vdash_\Sigma M : C \qquad \Gamma \vdash_\Sigma V : A \qquad \Sigma \vdash h : (A)C \Rightarrow D}{\Gamma \vdash_\Sigma \mathbf{handle}\ M\ \mathbf{with}\ h(V) : D}$$

$$\textsc{T-With}\ \frac{\Gamma \vdash_\Sigma U : A \xrightarrow{C} D \qquad \Gamma \vdash_\Sigma V : B \qquad \Sigma \vdash h : (B)C \Rightarrow D'}{\Gamma \vdash_\Sigma U\ \mathbf{with}\ h(V) : A \xrightarrow{C} D'} \qquad \textsc{T-Resume}\ \frac{\Gamma \vdash_\Sigma U : A \xrightarrow{C} D \qquad \Gamma \vdash_\Sigma V : A}{\Gamma \vdash_\Sigma \mathbf{resume}\ U(V) : D}$$

$$\textsc{T-NewBroom}\ \frac{\Sigma \vdash f : A \Rightarrow C \qquad \Gamma \vdash_\Sigma V : B \qquad \Sigma \vdash h : (B)C \Rightarrow D}{\Gamma \vdash_\Sigma \mathbf{newbroom}(f)\ \mathbf{with}\ h(V) : A \xrightarrow{C} D} \qquad \textsc{T-Env}\ \frac{dom(\gamma) = dom(\Gamma) \qquad (\vdash_\Sigma \gamma(x) : \Gamma(x))_{x \in dom(\gamma)}}{\vdash_\Sigma \gamma : \Gamma}$$

$$\textsc{T-KId}\ \frac{}{\vdash \mathbf{0} : A \to A!E} \qquad \textsc{T-KLet}\ \frac{\vdash \gamma : \Gamma \qquad \Gamma, x : A \vdash N : B!E \qquad \vdash \kappa : B \to B'!E}{\vdash (x).N[\gamma] \cdot \kappa : A \to B'!E} \qquad \textsc{T-KHandle}\ \frac{\Sigma \vdash h : (A)B!E \Rightarrow B'!E' \qquad \vdash v : A \qquad \vdash \kappa : B' \to B''!E'}{\vdash h(v) \cdot \kappa : A \to B''!E'}$$

$$\textsc{T-Handler}\ \frac{\begin{array}{c} C = B!(\{(\#t_i : B_i \to B'_i)_i\} \cup E) \qquad D = B'!(E' \cup E) \qquad x : A, y : B \vdash_\Sigma M : B'!E' \\ (x : A, z_i : B_i, r_i : B'_i \to D \vdash_\Sigma N_i : B'!E')_i \qquad (\mathbf{handler}\ h(x)\ \{\mathbf{return}\ y \to M \mid (\#t_i(z_i) \to (r_i).N_i)_i\}) \in \Sigma \end{array}}{\Sigma \vdash h : (A)C \Rightarrow D}$$

$$\textsc{T-Fun}\ \frac{x : A \vdash_\Sigma M : C \qquad (\mathbf{fun}\ f(x)\ \{M\}) \in \Sigma}{\Sigma \vdash f : A \Rightarrow C} \qquad \textsc{T-Program}\ \frac{\vdash_\Sigma M : C \qquad (\Sigma \vdash \Sigma(f) : A_f \Rightarrow C_f)_{f \in dom(\Sigma)} \qquad (\Sigma \vdash \Sigma(h) : (A_h)C_h \Rightarrow D_h)_{h \in dom(\Sigma)}}{\vdash \Sigma; M : C}$$

Fig. 1. SFX: typing rules.

To type handle replacement U **with** $h(V)$ on a broom U with type $A \xrightarrow{C} D$, the handler h must have a type $(B)C \Rightarrow D'$ where C matches, and the parameter V must have the expected type B: then the broom after replacing the handle will have an updated type $A \xrightarrow{C} D'$.

Rule T-Resume assigns to **resume** $U(V)$ the type D whenever U has a broom type $A \xrightarrow{C} D$ and V has type A. Finally rule T-NewBroom lifts a function f from type $A \Rightarrow C$ to a broom type $A \xrightarrow{C} D$ by means of a handler h of type $(B)C \Rightarrow D$ instantiated on a value v of type B. Note that since brooms have a first class type, **newbroom** allows us to convert a function into a value that can be freely passed around as an argument or returned.

Continuation typing. The judgement $\vdash_\Sigma \kappa : A \longrightarrow C$ states that the continuation κ receives a value of type A and continues as a computation of type C. Continuations include the identity continuation $\mathbf{0}$, which takes a value of any type A and returns a (trivial) computation of type $A!E$, and composite continuations obtained by prepending to an existing continuation a pure frame or a handler frame. The typing judgement for pure frames extension T-KLet ensures that the computation N in the frame is well-typed under an environment $\Gamma, x : A$, where x is the abstracted variable, that the environment γ has type Γ, and that the base continuation κ has an input type matching values returned by N and that its output effect matches the effects available to N. Extension by handler frames (rule T-KHandle) requires h to be the name of a valid handler of type $(A)B!E \Rightarrow B'!E'$; the actual parameter v must have type A in the empty context, and the base continuation κ must have type $B' \longrightarrow B''!E$, where B' matches the type of values returned by h and E' matches the output effect type of h.

Handler/function typing. Handlers and functions must be global items defined in the static specification Σ. The judgement T-Handler states that to give h the type $(A)C \Rightarrow D$ we need to be able to express C as $B!\{(\#t_i : B_i \longrightarrow B_i')_i\} \cup E$ and D as $B'!(E' \cup E)$, where E is the type of forwarded effects, i.e. client effects that are *not* handled by h, the $\#t_i$ are the effectful operations handled by h, and E' is the type of additional effects that may be raised by h. The rest of the judgement states that the definition of h in Σ must provide a return clause M and an operation clause N_i for each handled operation $\#t_i$, all of type $B'!E'$ matching the output type of the handler; the typing contexts for M and N_i assign types to the handler parameter x, to the return parameter y (for M) and to the operation argument z_i and the resumption r_i (for N_i). The typing of functions is standard.

4 An abstract machine for SFX

To provide a semantics for SFX, we develop an abstract machine in the style of the CEK machine [10]. The machine thus manipulates triples of control-environment-continuation, where the control is a computation or a closed value.

The machine is an adaptation of previous work on deep and shallow effect handlers [17]; however, our language presents a few differences, which are reflected into the definition:

- We lack type polymorphism and effect polymorphism and do not need to account for them in the machine.
- We include brooms as first-class terms of the language, with their own type. Brooms are structurally similar to the continuation data structure of the abstract machine both in this work and in [17]; however, while in our work SFX brooms and continuations of the abstract machine are structurally the same, their semantics differs subtly: brooms appear as the argument of a **resume**, where they are applied to a value: since the argument is a value, it cannot invoke an effect. On the other hand, abstract machine continuations behave like evaluation contexts: a machine triple $\langle M, \gamma, \kappa \rangle$ must be understood as the term $\kappa[M[\gamma]]$, where all the unhandled effects invoked in $M[\gamma]$ can be handled by κ. The two notions are related, in the sense that resuming a continuation κ with a value V is semantically equivalent to applying the evaluation context represented by κ to the trivial computation **return** V.
- We do not differentiate between deep and shallow handlers; instead we have a single kind of handlers (implemented similarly to deep handlers), and we use the handle replacement operation of brooms to express sheep handlers.

To save a few transition rules that are not relevant to the topic of this paper, we choose to assume that functions are translated to trivial handlers by a pre-processing operation:

$$\textbf{fun}\ f(x)\ \{M\} \rightsquigarrow \textbf{handler}\ f(\langle\rangle)\ \{\textbf{return}\ x \to M\}$$
$$f(V) \rightsquigarrow \textbf{handle return}\ V\ \textbf{with}\ f(\langle\rangle)$$

Closures are only needed in pure continuations. When we have a value V and an environment γ, we substitute eagerly via the operation $[\![V]\!]\,\gamma$. Such eager substitutions are trivial and always yield a literal value v.

The full machine is given in Figure 2: it consists of a dozen rules, but those implementing primitive operations, conditionals, and tuple unpacking are trivial. Sequencing $\textbf{let}\ x \leftarrow M\ \textbf{in}\ N$ is also standard, but it is worth noting that it proceeds by evaluating M under the continuation extended by a pure frame $(x).N[\gamma]$, where the variable x, which will receive the result of the evaluation of M, abstracts over the closure $N[\gamma]$ — this behaviour is expressed by the dual rule evaluating the closed value v when the continuation starts with a pure frame: notice that this rule reinstates the environment stored in the closure for N, which may have been destroyed by the intermediate evaluation steps for M.

Similarly, to evaluate a handled computation $\textbf{handle}\ M\ \textbf{with}\ h(V)$, we proceed by evaluating M under a continuation extended with a handler frame $h([\![V]\!]\,\gamma)$. The dual rule evaluating the closed value v when the continuation starts with a handler frame proceeds to evaluate the **return** clause of the handler h, completely replacing the environment with only two elements: the actual value w of the handler parameter, and the actual return value v received as input.

The evaluation of an effectful operation $\textbf{do}\ \#t(V)$ depends on the current continuation, which must be of the form $\kappa^{-t} \cdot h^{+t} \cdot \kappa'$, where the superscript $-t$ means that κ does not handle $\#t$, and the superscript $+t$ expresses the fact that the handler h specified in this frame must provide a clause for $\#t$: in other words, h is the first handler in scope which will handle $\#t$. To execute this operation, we proceed to evaluate $h_{\#t}$ (the body of the operation clause for $\#t$ defined by h); the continuation is split: the base continuation κ' is kept as the current continuation, whereas the initial fragment $(\kappa^{-t}, h^{+t}(w))$ is stored in the environment as the value of the resumption variable $res(h_{\#t})$, along with the values for the handler parameter and for argument of $\#t$. The original environment γ is discarded, but may be reinstated by subsequent resumptions.

The expression $\textbf{newbroom}(f)\ \textbf{with}\ h(V)$ immediately produces the broom corresponding to the function f: this consists of a pure frame $(x).f_{\textbf{return}}$ (remember that functions are encoded as the return clause of a trivial handler), delimited by the handler $h([\![V]\!]\,\gamma)$. The handle replacement expression $U\ \textbf{with}\ h(W)$ is evaluated by obtaining the broom $[\![U]\!]\,\gamma$, discarding its handle, and replacing it with $h([\![W]\!]\,\gamma)$.

To evaluate a simple resumption $\textbf{resume}\ U(V)$, we apply the environment γ to U to obtain the corresponding literal continuation $(\kappa', h(w))$: then we proceed by evaluating **return** V, where the continuation κ has been extended by $\kappa' \cdot h(w)$.

We evaluate **return** V by converting it to the closed value $[\![V]\!]\,\gamma$.

5 An abstract assembly language with effect handlers

Having formalised the high-level effect idioms we intend to support by means of SFX and its abstract machine, it is now time to develop a low-level target language in the style of the typical instruction set architectures supported in hardware, but providing instructions to install effect handlers and perform effects: we call this language AsmFX.

AsmFX comprises a small set of instructions operating on registers, a main memory containing data and code, and an *effect context*, special storage manipulated as a stack and separated from main memory. Though close to typical instruction sets, AsmFX leaves several features abstract:

control	env.	continuation	$\hookrightarrow$	control	env.	continuation
$\oplus(V)$	γ	κ		$[\![\oplus]\!]\,([\![V]\!]\,\gamma)$	γ	κ
if (V, M, N) where: $[\![V]\!]\,\gamma$ = **true**	γ	κ		M	γ	κ
if (V, M, N) where: $[\![V]\!]\,\gamma$ = **false**	γ	κ		N	γ	κ
let $x \leftarrow M$ **in** N	γ	κ		M	γ	$(x).N[\gamma] \cdot \kappa$
unpack $\overline{x_n} \leftarrow \langle \overline{V_n} \rangle$ **in** N	γ	κ		N	$(x_n \mapsto [\![V_n]\!]\,\gamma)$ $\cdot\gamma$	κ
handle M **with** $h(V)$	γ	κ		M	γ	$h([\![V]\!]\,\gamma) \cdot \kappa$
do $\#t(V)$	γ	$\kappa^{-t} \cdot h^{+t}(w) \cdot \kappa'$		$h_{\#t}$	$(res(\#t) \mapsto (\kappa^{-t}, h^{+t}(w))$ $\cdot(dom(\#t) \mapsto [\![V]\!]\,\gamma)$ $\cdot(dom(h) \mapsto W)$	κ'
newbroom(f) **with** $h(V)$	γ	κ		$((x).f_{\text{return}}\,[\,],$ $h([\![V]\!]\,\gamma))$ where: $x = dom(f_{\text{return}})$	γ	κ
U **with** $h(W)$ where: $[\![U]\!]\,\gamma = (\kappa', \_)$	γ	κ		$(\kappa', h([\![W]\!]\,\gamma))$	γ	κ
resume $U(V)$ where: $[\![U]\!]\,\gamma = (\kappa', h(w))$	γ	κ		$[\![V]\!]\,\gamma$	γ	$\kappa' \cdot h(w)$ $\cdot\kappa$
return V	γ	κ		$[\![V]\!]\,\gamma$	γ	κ
v	γ	$(x).N[\gamma'] \cdot \kappa$		N	$(x \mapsto v) \cdot \gamma'$	κ
v	γ	$h(w) \cdot \kappa$		h_{return}	$(dom(h_{\text{return}}) \mapsto v)$ $\cdot(dom(h) \mapsto w)$	κ

Fig. 2. SFX abstract machine

- Every register and every memory location is large enough to store any value, including certain complex values with special architectural meaning that we will call *records*, used to implement control flow.
- There are as many registers as may be needed by any program. Registers are referred by their name, starting with a dollar sign '$'. While all registers can contain any value, we will use naming conventions to divide them in different categories for different purposes. The register $pc is the program counter and it contains the memory address of the next instruction that will be executed.
- Only instructions explicitly needed for implementing effect handlers are included. Other standard instructions including unconditional jumps, conditional branches, arithmetic and logical operations, register moves, or load and store from main memory are left unspecified, but are intended to be available as needed, with their natural semantics.

Main memory is an array of words, and memory addresses $a, a', \ldots$ are indices in the array specified as natural numbers. Effects are specified by tags $\#tag$; the special tag *return* identifies the return clause of handlers. We now describe the syntax of AsmFX:

Handlers:	H	::=	$return \mapsto a_{return}, \overline{\#tag \mapsto a_{\#tag}}$
Records:	R	::=	$R_{\mathcal{H}} \mid R_{\mathcal{K}} \mid R_{\mathcal{L}}$
Handler rec.:	$R_{\mathcal{H}}$	::=	$\mathcal{H}(H, \overline{\$\mathsf{r_h} = v_h})$
Continuation rec.:	$R_{\mathcal{K}}$	::=	$\mathcal{K}(a_r, \overline{\$\mathsf{r_c} = v_c}, D_r)$
Leave rec.:	$R_{\mathcal{L}}$	::=	$\mathcal{L}(a_l, \overline{\$\mathsf{r_e} = v_e})$
Values:	v	::=	$R \mid a \mid I \mid \ldots$
Instructions:	I	::=	$\mathtt{loadh}\ H, state(\overline{\$\mathsf{r_h}}) \mid \mathtt{loadk}\ a_r, save(\overline{\$\mathsf{r_c}}) \mid \mathtt{loadl}\ a_l, save(\overline{\$\mathsf{r_e}})$
		$\mid$	$\mathtt{resume} \mid \mathtt{do}\ \#tag, save(\overline{\$\mathsf{r_c}}) \mid \mathtt{return} \mid \mathtt{exit}$
Effect contexts:	C, D	::=	$\overline{\mathcal{H}(H, \overline{\$\mathsf{r_h} = v_h}, R_{\mathcal{L}})}$

Records. To implement effect handlers at the assembly level, we introduce representations of the key linguistic concepts we need as special values that can be directly stored in registers: in our terminology, these are called *records*. Records are similar to closures, and will be used to represent three kinds of objects:

- Effect handlers, by means of records in the form $\mathcal{H}(H, (\overline{\$\mathtt{r}_\mathtt{h} = v_h}))$, where H is a finite partial map from operation names to memory addresses of code to handle that operation, including one address providing code for returning from the handler; the second subterm is a finite partial map from registers names to values, used to store the handler state.
- Continuations, by means of records $\mathcal{K}(a_r, (\overline{\$\mathtt{r}_\mathtt{c} = v_c}), D_r)$, where a_r is the address of the code that will be executed when resuming the continuation, and the other two parameters express the local environment that needs to be reinstated when control is passed to the continuation: this local environment consists of a finite map from register names to their values, and a delimited effect context expressing additional handlers that will need to be added to the effect context to execute the continuation.
- Leave records, in the form $\mathcal{L}(a_l, (\overline{\$\mathtt{r}_\mathtt{e} = v_e}))$, indicate what code should be executed after leaving the scope of a handler; they are similar to continuations, except that they do not provide a delimited effect context because when we leave the scope of a handler we never need to install additional handlers.

Effect context. The effect context is a stack of *handler frames* representing the active handlers at any given moment at runtime. Handler frames contain the same information as a handler record, pairing it with a leave record indicating what code should be run after returning from the handler.

Instructions. AsmFX introduces three instructions `loadh`, `loadk`, `loadl` to load handler, continuation, and leave records into registers. The three instructions use, respectively, registers `$hr`, `$kr`, and `$lr` as their targets: these registers have an architectural meaning in that their content can affect the control flow; it is however possible to copy the content of these registers to any other register, and vice-versa, using general-purpose instructions.

`loadh` H, *state* :$(\overline{\$\mathtt{r}'})$ loads a handler record into `$hr`. The handler specification H contains pairs of tag names and code addresses implementing the handler operation specified by that tag; the pseudo-tag *return* is always present. The content of registers $\overline{\$\mathtt{r}'}$, specifying the current state of the handler, is saved within the record.

`loadk` a_r, *save* :$(\overline{\$\mathtt{r}'})$ loads a continuation record into register `$kr` by specifying the address a_r of its code and the registers $\overline{\$\mathtt{r}'}$ whose value must be saved now and restored if the continuation is run and terminates regularly. The continuation is created with a trivial (empty) delimited context.

`loadl` a_l, *save* :$(\overline{\$\mathtt{r}'})$ loads a leave record into register `$lr`, where a_l is the address of the code that will be run when the leave record is triggered, and the save registers $\overline{\$\mathtt{r}'}$ contain the values that need to be restored to execute that code.

The remaining four instructions deal with control flow. The instruction `resume` invokes the continuation stored in the special register `$kr`, wrapping it in the handler specified by the special register `$hr`. Note that the two registers together essentially perform the same function as brooms in SFX, with `$kr` acting as the head, and `$hr` as the handle.[1] In order to execute this instruction, the register `$lr` must also contain the record referencing the code that will be executed if and when the resumed continuation terminates. To resume the continuation, the machine pushes into the context a new handler frame obtained by combining the contents of `$hr` and `$lr`, followed by the

[1]To be more precise, AsmFX, unlike SFX, enables a slightly more liberal policy by allowing the code to reference and replace heads and handles independently. By a cultural reference to the British sitcom *Only Fools and Horses*, we describe the pair of `$kr` and `$hr` as *Trigger's broom*.

delimited context from $kr, then restores the contents of the registers saved in $kr and jumps to the code address of the continuation.

do *#tag*, *save* :($\overline{\$r}$) performs an effectful operation by jumping to the corresponding handler code; after the jump, the registers $kr, $hr, and $lr are loaded with the continuation record for the suspended computation that triggered the effect (including saved values for the registers $\overline{\$r}$, the handler record that is handling the effect, and the leave record associated with that handler; the handler state registers are also restored from the handler record. When (and if) the operation returns a value, it will use the continuation in $kr to jump back to the next instruction after do, restoring the contents of the save registers $\overline{\$r}$.

return operates similarly to do but it always matches the handler at the top of the context and when jumping to its return clause it does not need to provide a resumption. This means it loads $lr and the state registers from the handler frame, and then jumps to the address specified by the return clause $H(\#return)$.

exit completes the exit procedure from a handler by triggering the leave record in $lr and restoring the registers saved in it.

instruction	arguments	context	$\hookrightarrow$	new context	reg. updates
loadk	a_r *save* : ($\overline{\$r_c}$)	C		C	$\$kr = \mathcal{K}(a_r, (\overline{\$r_c = \Theta(\$r_c)}), [])$
loadh	H *state* : ($\overline{\$r_h}$)	C		C	$\$hr = \mathcal{H}(H, (\overline{\$r_h = \Theta(\$r_h)}))$
loadl	a_l *save* : ($\overline{\$r_e}$)	C		C	$\$lr = \mathcal{L}(a_l, (\overline{\$r_e = \Theta(\$r_e)}))$
resume		C where: $\$hr = \mathcal{H}(H, (\overline{\$r_h = v_h}))$ $\$kr = \mathcal{K}(a_r, (\overline{\$r_c = v_c}), D_r)$ $\$lr = R_{\mathcal{L}}$		D_r $\cdot\ \mathcal{H}(H, (\overline{\$r_h = v_h}), R_{\mathcal{L}})$ $\cdot\ C$	$\$pc = a_r$ $\overline{\$r_c = v_c}$
do	*#tag* *save* : ($\overline{\$r_c}$)	D_r^{-tag} $\cdot\ \mathcal{H}(H^{+tag}, (\overline{\$r_h = v_h}), R_{\mathcal{L}})$ $\cdot\ C$		C	$\$kr = \mathcal{K}(\Theta(\$pc), (\overline{\$r_c = \Theta(\$r_c)}), D_r^{-tag})$ $\$hr = \mathcal{H}(H^{+tag}, (\overline{\$r_h = v_h}))$ $\$lr = R_{\mathcal{L}}$ $\$pc = H^{+tag}(\#tag)$ $\overline{\$r_h = v_h}$
return		$\mathcal{H}(H, (\overline{\$r_h = v_h}), R_{\mathcal{L}})$ $\cdot\ C$		C	$\$pc = H(\#return)$ $\overline{\$r_h = v_h}$ $\$lr = R_{\mathcal{L}}$
exit		C where: $\$lr = \mathcal{L}(a_l, (\overline{\$r_e = v_e}))$		C	$\$pc = a_l$ $\overline{\$r_e = v_e}$

Domain of contexts:
$dom([\,]) = \emptyset \qquad dom(\mathcal{H}(H, \overline{\$r_h = v_h}, R_{\mathcal{L}}) \cdot C) = dom(H) \cup dom(C) \qquad dom(\#t_1 \mapsto a_1, \ldots, \#t_n \mapsto a_n) = \{\#t_1, \ldots, \#t_n\}.$
D^{-t} indicates any delimited context such that $\#t \notin dom(D^{-t})$. H^{+t} denotes any handler definition where $\#t \in dom(H^{+t})$.

Fig. 3. AsmFX machine transitions.

The semantics of AsmFX is expressed by the abstract machine in Figure 3. A machine state is specified by a tuple $\langle \Xi, \Theta, C \rangle$, where Ξ, the main memory, maps addresses to values, which may be data or instructions; Θ, the register file, maps register names to their current value; and C, the effect context, is a stack of handler frames specifying what effect implementations are available to the code currently running. When it is ready to execute an instruction, the machine fetches it from memory Ξ at the address stored in the $pc register (it is assumed that the memory at that address contains a valid instruction, otherwise the machine will be stuck); at this point the machine immediately increments $pc to ensure it points to the next instruction, then it matches the current instruction (including its arguments) and the effect context C against the first three columns of the transition table: execution involves updating the context C and the registers file Θ according to the

content of the last two columns: among other things, this allows an instruction to perform a jump by updating the register $pc.

The instructions we describe do not use the main memory Ξ except as storage for the code being executed, but a concrete instance of AsmFX can read memory values into registers and write arbitrary values into memory by defining its own load/store instructions.

6 From SFX to AsmFX

We have defined an assembly language with an intuitive notion of effect handlers and the instructions we deemed essential to write effectful programs. Though these instructions, after careful analysis of their machine semantics, may appear to reasonably implement the semantics of effect handlers when considered in isolation, to provide a solid foundation for this claim, we will show that SFX programs can be compiled to AsmFX. For this abstract discussion, we can conveniently express the compiler by means of recursive procedures taking SFX syntactic forms as input, and returning an AsmFX memory image as output. For reasons that will be clear when we will prove the correctness of the compiler, we assume that, along with producing the compiled program, the compiler will also reflect the result of the compilation in the source code in the form of *annotations* stating, for each subexpression, the memory location ℓ of the AsmFX code that evaluates it: this means source values, computations, or programs V, M, P will be augmented into annotated versions $\hat{V}, \hat{M}, \hat{P}$; we call the language of augmented SFX expressions ASFX. We will refrain from giving the tedious details of this backwards annotation (entirely analogous to the debugging symbols produced by most compilers), and we will assume that it is correct (in the sense that each annotation corresponds to the address of the code produced when compiling the corresponding subexpression), and that by erasing annotations we get back the original source. We use $\mu(\cdot)$ and $\nu(\cdot)$ to denote the starting and ending AsmFX location for a given augmented expression, and $|\cdot|$ to indicate the erasure of an augmented expression into the corresponding plain SFX expression.

Fact 6.1. *Let Ξ_P be the AsmFX memory image containing the compiled version of the SFX program P, and $\hat{P}$ be the augmented version of P produced by the compiler: then for every computation $\hat{M}$ or value $\hat{V}$ appearing in $\hat{P}$, we have that the portion of Ξ_P delimited by $[\mu(\hat{M}), \nu(\hat{M}))$ (resp. $[\mu(\hat{V}), \nu(\hat{V}))$) contains the compiled code for $|\hat{M}|$ (resp. $|\hat{V}|$).*

Sometimes we will also write explicitly the annotations of expressions using the syntax $\hat{M}@\ell$ to indicate that $\mu(\hat{M}) = \ell$. A formal grammar of ASFX is included in the appendix.

Due to limitations on the type of data that can be stored in a single AsmFX register, we will use the notation $\|V\|$ or $\|A\|$ to denote the number of registers needed to store a certain SFX value V, or values of a certain type A.

By convention, the compiler will partition the register set into *variable* registers $\$x_i$ (where x is an SFX variable), *argument* registers $\$a_i$ (holding function inputs or the value of an expression), *state* registers $\$s_i$ (used for handler state), and separate special registers $\$hr, $\$kr, $\$lr, $\$pc with architectural meaning. Registers in the first three classes are available in unlimited number.

Compiling computations. To denote the compilation of SFX computations M, we use the syntax $\lfloor M \rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}'$. The operation takes two auxiliary parameters: a set of input registers $\mathbb{I}$ whose value is used in the computation and must be preserved at the end of the compiled code, corresponding to free variables in the source computation M (in particular, each free variable will correspond to zero, one, or more registers depending on its type); and the used label set $\mathbb{L}$, which is needed to avoid collisions when generating new AsmFX code labels. The result of running the compiled computation is always placed in an initial segment of the argument registers $\overline{\$a}$.

$\lfloor M \rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}'$ produces a main output, i.e. a list of AsmFX instructions annotated with labels representing the result of compiling E, and a secondary output (the updated list of used labels $\mathbb{L}'$, which is always a superset of $\mathbb{L}$). The definition of $\lfloor M \rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}'$ is given, for any expression M, by using syntax such as the following:

$$\textbf{let } \mathcal{S} = exp \ldots$$
$$\ldots \lfloor M \rfloor^{\mathbb{I}_M,\mathbb{L}_M} \Rightarrow \mathbb{L}'_M \ldots$$

The **let** syntax tells the compiler to evaluate a certain meta-expression exp once and introduces a metavariable $\mathcal{S}$ to indicate the result of said evaluation: we use this syntax to introduce fresh code labels, or to express sets of registers or labels in a short form. The recursive compilation of M inlines the compiled code for M and introduces a metavariable $\mathbb{L}'_M$ for the extended set of used labels. The ellipses may contain AsmFX instructions and labels. It is always assumed that the secondary output of the compilation is the one assigned at the end of the compilation process for a given expression to the metavariable $\mathbb{L}'$. We also assume an operation $\varphi(n, \mathbb{L})$, producing n labels fresh with respect to the used label set $\mathbb{L}$. The concrete choice of labels is irrelevant and consistent renaming is assumed to denote the same code (we use labels for the sake of readability, but each label must be considered notation for an absolute memory index).

Space constraints prevent us from presenting the details of the compiler here: instead, we will comment the code implementing the compilation of **do** and **handle**.

$\lfloor \textbf{do } \#t(V) \rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}' :=$

```
⌊V⌋
do #t, save :𝕀
let 𝕃' = 𝕃
```

To compile **do** $\#t(V)$, we first compile V, whose value will be then found in $\overline{\texttt{\$a}}$, then we use the `do` instruction (the *save* clause states that the value of the registers $\mathbb{I}$ used in the client must be saved now and restored if the effectful operation returns to this code).

$\lfloor \textbf{handle } M \textbf{ with } h(V) \rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}' :=$

```
let l_cli, l_exit = φ(2, 𝕃)                 |   loadl $lr, l_exit, save :𝕀
let 𝕃_M = 𝕃 ∪ {l_cli, l_exit}               |   resume
let 𝕀_h = $s_‖V‖                            | l_cli :
⌊V⌋                                         |   ⌊M⌋^{𝕀,𝕃_M} ⇒ 𝕃'_M
𝕀_h ← $a_‖V‖                                |   return
loadh $hr, ⌊h⌋, state :𝕀_h                  | l_exit :
loadk $kr, l_cli, save :()                  |   let 𝕃' = 𝕃'_M
```

To execute a computation under a new handler by means of **handle** M **with** $h(V)$, we compile V, dopy its evaluation in the $\overline{\texttt{\$s}}$ registers, and use it together with $\lfloor h \rfloor$ to create a handler and place it in `$hr`. We also create a continuation pointing to the code for M (l_{cli}) and place it in `$kr`, and a leave record for the code to be executed after this computation (l_{exit}), with saved registers corresponding to the input set $\mathbb{I}$). After these three registers have been loaded, we invoke the client by resuming the continuation we created, using the instruction `resume`. Notice that the client code at location l_{cli} is terminated by a `return` instruction, so that the result of M will be delivered to the return clause of the newly installed handler: we call this a **return** epilogue. Tracking epilogues is essential to prove the correctness of the compiler.

Compiling handlers. Handlers consist of a computation expression for each return and operation clause: it is easy to compile each of them separately using the same procedure we use for all other computations. Additionally, for each clause, the compiler takes care of moving the actual parameter of the handler from the state registers $\overline{s}$ to variable registers $\overline{x}$ corresponding to the formal parameter x used by the SFX handler definition. The compiler will also insert code to move the arguments

of each return clause from the argument registers $\overline{a}$ to variable registers $\overline{y}$ corresponding to the formal argument y of that clause in the SFX definition, and to move the broom resumption from the special registers $\$\mathtt{kr}$, $\$\mathtt{hr}$ to the appropriate variable registers $\$\mathtt{r}_1$, $\$\mathtt{r}_2$.

The handler result (returned when exiting the handler, either normally through the return clause or abnormally through an operation clause) is stored in an initial segment of the argument registers $\overline{\$\mathtt{a}}$. Each clause is terminated by an `exit` instruction which triggers the leave register; this instruction also constitutes an epilogue that needs to be accounted for in the soundness proof.

For each clause **return**, $\#t_i$ of a handler h, we will generate a unique label $h_{\mathbf{return}}$, $h\#t_i$ pointing to its code: this allows us to compile a handler specification $\lfloor h \rfloor$ as

$$return \mapsto h_{\mathbf{return}}, \#t_1 \mapsto h_{\#t_1}, \ldots$$

7 Soundness of compilation

As a final step in our effort to assess the expressiveness of AsmFX, we need to prove the correctness of our compiler. The most basic correctness property would state that for any SFX program P and its AsmFX counterpart $\lfloor P \rfloor$, if P terminates returning a value v, then $\lfloor P \rfloor$ also terminates and a machine representation of v can be found in an initial segment of the $\overline{\$\mathtt{a}}$ registers; however, we are also interested in proving the correctness of partial executions, by showing that the transition system induced by the AsmFX machine simulates the one induced by the SFX abstract machine.

The proof is not entirely straightforward. The main difficulty lies in the fact that executing an SFX program consumes part of the program itself, blurring the correspondence between source and compiled object code beyond recognisability: the code resulting from the recompilation of a partially executed program will not match, even partially, the code of the original program, because parts of the code are completely lost and thus ignored by the compiler. To match partially executed source code to partially executed AsmFX code we need a more sophisticated instrument.

Our solution is to prove soundness by reasoning not on the plain source code, but on its ASFX version. We refine the SFX abstract machine to operate directly on ASFX programs, which now carry along location information and other information about execution history. The execution of an ASFX program differs from that of the corresponding SFX program especially in the form of its continuations: the identity continuation and all frames are annotated with a start location; additionally, pure frames $((x).\hat{M})@\ell$ contain a naked computation instead of a computation closure; we also introduce *epilogue frames* **jump**, **exit**, **leave** representing AsmFX instructions that must be executed in order to link the various portions of compiled code together. The closures missing from pure frames resurface as an additional environment $\hat{\gamma}$ component of ASFX brooms.

Much like we did for SFX, we formalise the semantics of ASFX by means of a CEK-like abstract machine. The main difference between the two abstract machines is that the ASFX machine is aware of location annotations and propagates them during its execution, ensuring that all relevant information is preserved. The definition of continuations also differs in that it tracks certain intermediate execution steps that are implicit in SFX but explicit in AsmFX.

For example, the rule for **do** $\#t(\hat{V})$ is now refined to the following form:

$$\begin{aligned}&\langle \mathbf{do}\ \#t(\hat{V}), \hat{\gamma}, \hat{\kappa}^{-t} \cdot h^{+t}(\hat{w}, R_{\mathcal{L}})@\ell \cdot \hat{\gamma}' \rangle \hookrightarrow \\ &\quad \langle h_{\#t}, (\mathsf{LEAVE} \mapsto R_{\mathcal{L}}) \cdot (res(\#t) \mapsto (\hat{\kappa}^{-t}, h^{+t}(\hat{w})@\ell, \hat{\gamma})) \cdot \\ &\qquad (dom(\#t) \mapsto [\![\hat{V}]\!]\ \hat{\gamma}) \cdot (dom(h) \mapsto \hat{w}), \mathbf{exit}@\nu(h_{\#t}) \cdot \hat{\kappa}' \rangle\end{aligned}$$

In this rule, the AsmFX-like leave record $R_{\mathcal{L}}$ is manipulated explicitly and bound to the pseudovariable LEAVE (meaningful to the abstract machine, but not directly accessible to the code). Furthermore, an **exit** epilogue is added to the output continuation, expressing the fact that if the evaluation of the operation clause $h_{\#t}$ completes without calling a resumption, we need to execute

Values and environments

$$(\kappa, h(w)) \preccurlyeq (\hat{\kappa}, h(\hat{w})@\ell, \hat{\gamma}_\kappa) \quad := \quad \kappa \preccurlyeq (\hat{\gamma}_\kappa, \hat{\kappa}) \wedge w \preccurlyeq \hat{w}$$
$$v \preccurlyeq \hat{v} \quad := \quad v = |\hat{v}| \qquad (\text{if } v \neq k)$$
$$\gamma \preccurlyeq \hat{\gamma} \quad := \quad \forall x \in dom(\gamma).\gamma(x) \preccurlyeq \hat{\gamma}(x)$$

Continuations

$$\frac{}{\mathbf{0} \preccurlyeq (\hat{\gamma}, \mathbf{0}@\ell)} \qquad \frac{\kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa})}{\kappa \preccurlyeq (\hat{\gamma}, \mathbf{jump}@\ell \cdot \hat{\kappa})} \qquad \frac{\hat{\gamma}(\mathsf{LEAVE}) = (\_, \hat{\gamma}_\kappa) \quad \kappa \preccurlyeq (\hat{\gamma}_\kappa, \hat{\kappa})}{\kappa \preccurlyeq (\hat{\gamma}, \mathbf{exit}@\ell \cdot \hat{\kappa})} \qquad \frac{\kappa \preccurlyeq ((\mathsf{LEAVE} \mapsto \mathcal{L}(\ell + 2, [])) \cdot \hat{\gamma}, \hat{\kappa})}{\kappa \preccurlyeq (\hat{\gamma}, \mathbf{leave}@\ell \cdot \hat{\kappa})}$$

$$\frac{\forall v.\kappa \preccurlyeq ((x \mapsto v) \cdot \hat{\gamma}, \hat{\kappa}) \qquad \gamma_N \preccurlyeq \hat{\gamma} \qquad N = \left|\hat{N}\right|}{((x).N[\gamma_N]) \cdot \kappa \preccurlyeq (\hat{\gamma}, ((x).\hat{N})@\ell \cdot \hat{\kappa})} \qquad \frac{\kappa \preccurlyeq (\hat{\gamma}_\kappa, \hat{\kappa})}{h(w) \cdot \kappa \preccurlyeq (\hat{\gamma}, h(w, \mathcal{L}(\ell_h, \hat{\gamma}_\kappa))@\ell \cdot \hat{\kappa})}$$

Configurations

$$\langle M, \gamma, \kappa \rangle \preccurlyeq \langle \hat{M}, \hat{\gamma}, \hat{\kappa} \rangle := \left|\hat{M}\right| = M \wedge \gamma \preccurlyeq \hat{\gamma} \wedge \kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa}) \qquad \langle v, \gamma, \kappa \rangle \preccurlyeq \langle \hat{v}, \hat{\gamma}, \hat{\kappa} \rangle := v \preccurlyeq \hat{v} \wedge \kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa})$$

Fig. 4. Scope-preserving simulation

some AsmFX code (an `exit` instruction) that will dispose of the handler h and return to the external code. Due to space constraints, the full definition of the abstract machine and an explanation of its rationale is deferred to the appendix.

7.1 Well-scoping

We prove that single step transitions in SFX are simulated by multistep transitions in ASFX. Since ASFX epilogues essentially correspond to no operation at all in SFX, the main challenge is to show that in spite of a different approach to handling environments, ASFX provides a correct implementation of scopes. The simulation relation, summarised in Figure 4, essentially expresses a well-scoping invariant.

When an SFX configuration s is simulated by an ASFX configuration $\hat{s}$, we write $s \preccurlyeq \hat{s}$. The relation is defined by means of auxiliary definitions for the simulation of environments and continuations. An interesting fact about the simulation of continuation is that SFX continuations are simulated by pairs of ASFX continuations and environments: this is explained by the fact that SFX pure frames are closed with respect to an environment, while ASFX frames are not.

The environment simulation $\gamma \preccurlyeq \hat{\gamma}$ expresses the fact that whenever x is in the domain of γ, it is also in the domain of $\hat{\gamma}$ and the values for x in the two environments "agree". When $\gamma(x)$ is not a broom, $\hat{\gamma}(x)$ agrees with it if by erasing annotations we get the same value; if $\gamma(x)$ is a broom $(\kappa, h(w))$, then $\hat{\gamma}(x)$ must be a broom $\hat{\kappa}, h(w)@\ell, \hat{\gamma}_\kappa$ such that κ is simulated by $(\hat{\gamma}_\kappa, \hat{\kappa})$.

The continuation simulation $\kappa \preccurlyeq \hat{\kappa}$ is defined by means of an inductive judgement, with a base rule for the identity continuation and recursive rules for each type of frame. Identity continuations are simulated by identity continuations regardless of the environment. In the interesting case of **exit** frames, the active augmented environment is swapped with the one contained in the LEAVE pseudovariable (which must be defined), reflecting the semantics of the AsmFX instruction `exit`, which will be executed if the **exit** frame is activated. This kind of rule ensures that while the environment of the SFX machine and that of the ASFX machine may temporarily differ in non-trivial ways (specifically when the control is a value), the ASFX machine will eventually synchronise its environment.

The other interesting case is that of pure frames. If both continuations start with a pure frame, in order for the simulation to hold, the computations N and $\hat{N}$ in each pure frame must match (meaning that by erasing $\hat{N}$ we get N). Furthermore, the SFX pure frame is closed by an environment γ_N: that environment must be simulated by the ASFX environment $\hat{\gamma}$, reflecting the fact that in ASFX we do not need an environment in the pure frame, because the active environment is already the correct one. If these conditions hold, the simulation is valid if there is a simulation between the two remaining parts of the two continuations; however on the ASFX side we need to extend the environment with a value for the variable x expected by the pure continuation, and the simulation must hold independently of the value chosen.

This definition allows us to prove the theorem:

Theorem 7.1. *For all SFX configurations s, s' such that $s \hookrightarrow s'$, for all ASFX configurations $\hat{s}$ such that $s \preccurlyeq \hat{s}$, there exists an ASFX configuration $\hat{s}'$ such that $s' \preccurlyeq \hat{s}'$ and $\hat{s} \hookrightarrow \hat{s}'$.*

The proof, which is detailed in the appendix, is by case analysis on the transition $s \hookrightarrow s'$ and on the relation $s \preccurlyeq \hat{s}$.

7.2 Control flow simulation

To complete the proof of correctness of the compiler, we now need to establish a simulation relation between ASFX and AsmFX. We consider AsmFX configurations $a = \langle \Xi, \Theta, C \rangle$, whose elements are memory stores Ξ (containing the program), register files Θ (finite maps from registers to values), and effect contexts C. Since a property of our compiler is that the memory store for a given source program is established at compilation time and stays read-only during execution, we will allow ourselves to omit it from AsmFX configurations and write $\langle \Theta, C \rangle$ for $\langle \Xi, \Theta, C \rangle$.

A source configuration $\hat{s}$ for a program $\hat{P}$ should correspond to an AsmFX configuration a on $\lfloor |\hat{P}| \rfloor$: actually, for every $\hat{s}$ there are multiple such a, because of the plethora of registers available in the architecture: some of the registers will not have a counterpart in the source configuration and will thus be irrelevant. Therefore, the translation to AsmFX of a source configuration $\hat{s}$, which we denote by $\lfloor \hat{s} \rfloor$, will be the *set* of AsmFX configurations that are a model for $\hat{s}$ up to relevant registers. If we write $a \vDash \hat{s}$ to mean that a models $\hat{s}$, then we can define $\lfloor \hat{s} \rfloor := \{a : a \vDash \hat{s}\}$. We define what it means to be a model separately for the register file and effect context components: the register file must be a model for the whole configuration; on the other hand, the effect context only needs to model the continuation.

$$\langle \Theta, C \rangle \vDash \langle \hat{M}, \hat{\gamma}, \hat{\kappa} \rangle := (\Theta \vDash \langle \hat{M}, \hat{\gamma}, \hat{\kappa} \rangle) \wedge (C \vDash \hat{\kappa})$$

We can define what it means for an effect context to model a continuation by means of a direct translation that only needs to consider the handler frames:

$$\begin{array}{rcl}
C \vDash \hat{\kappa} & := & C = \lfloor \hat{\kappa} \rfloor \\
\lfloor \hat{\kappa} \rfloor & := & \begin{cases} \lfloor h(\hat{w}, \mathcal{R}_{\mathcal{L}}))@\ell \rfloor \cdot \lfloor \hat{\kappa}' \rfloor & \text{if } \hat{\kappa} = h(\hat{w}, \mathcal{R}_{\mathcal{L}})@\ell \cdot \hat{\kappa}' \\ \lfloor \hat{\kappa}' \rfloor & \text{if } \hat{\kappa} = \hat{\varsigma} \cdot \hat{\kappa}' \text{ and } \hat{\varsigma} \neq h(_) \end{cases} \\
\lfloor h(\hat{w}, \mathcal{R}_{\mathcal{L}})@\ell \rfloor & := & \mathcal{H}(\lfloor h \rfloor, (\overline{\$\mathsf{s}} = \lfloor \hat{w} \rfloor), \lfloor \mathcal{R}_{\mathcal{L}} \rfloor) \\
\lfloor \mathcal{L}(\ell, \hat{\gamma}) \rfloor & := & \overline{\mathcal{L}(\ell, \lfloor \hat{\gamma} \rfloor)} \\
\left\lfloor \overline{(x \mapsto \hat{v})} \right\rfloor & := & \overline{(\$\mathsf{x} = \lfloor \hat{v} \rfloor)}
\end{array}$$

A register file models a source configuration if every variable defined in the source environment is backed by (any number of) registers containing the AsmFX representation of that variable's value; if the program counter matches the address of the beginning of the computation being evaluated or, when the computation has ended and we are returning a value, when it matches the address of

Values $\quad \hat{v}\checkmark := \forall \hat{k} \in \mathit{subval}(\hat{v}) : \hat{k}\checkmark$

Environments $\quad \hat{\gamma}\checkmark := \forall x \in \mathit{dom}(\hat{\gamma}) : \hat{\gamma}(x)\checkmark$

Brooms $\quad \hat{k}\checkmark := \forall (\hat{\kappa}, h(\hat{w})@\ell, \hat{\gamma}) = \hat{k}, \forall \hat{\kappa}'\checkmark, \forall \hat{\gamma}' \supset \hat{\kappa}' :$
$\hat{\gamma} \supset (\hat{\kappa} \cdot h(\hat{w}, \mathcal{L}(\mu(\hat{\kappa}'), \hat{\gamma}'))@\ell \cdot \hat{\kappa}') \wedge (\hat{\kappa} \cdot h(\hat{w}, \mathcal{L}(\mu(\hat{\kappa}'), \hat{\gamma}'))@\ell \cdot \hat{\kappa}')\checkmark$

Coherence $\quad \hat{\gamma} \supset \hat{\kappa} := \hat{\gamma}\checkmark \wedge \hat{\gamma}(\mathsf{LEAVE}) = (\ell, \hat{\gamma}') \Longrightarrow \ell = \mathit{laddr}(\hat{\kappa}) \wedge \hat{\gamma}' \supset \mathit{lcont}(\hat{\kappa})$

Continuations

$$\frac{}{\mathbf{0}@\ell\checkmark} \qquad \frac{\hat{\kappa}\checkmark \qquad \Xi(\ell) = \mathtt{j}\ \mu(\hat{\kappa})}{\mathbf{jump}@\ell \cdot \hat{\kappa}\checkmark} \qquad \frac{\hat{\kappa}\checkmark \qquad \Xi(\ell) = \mathtt{exit}}{\mathbf{exit}@\ell \cdot \hat{\kappa}\checkmark}$$

$$\frac{\hat{\kappa}\checkmark \qquad \Xi(\ell) = \mathtt{loadl}\ (\ell + 2) \qquad \hat{\kappa} = \mathbf{jump}@(\ell + 1) \cdot ((x).\hat{N})@\mu(\hat{N}) \cdot \mathbf{exit}@\nu(\hat{N}) \cdot h(\hat{w})@(\ell + 2) \cdot \hat{\kappa}_0}{\mathbf{leave}@\ell \cdot \hat{\kappa}\checkmark}$$

$$\frac{\hat{\kappa}\checkmark \qquad \Xi(\ell) = \overline{\$\mathtt{x}} \leftarrow \overline{\$\mathtt{a}}; \lfloor \hat{N} \rfloor \qquad \nu(\hat{N}) = \mu(\hat{\kappa})}{((x).\hat{N})@\ell \cdot \hat{\kappa}\checkmark} \qquad \frac{\hat{\kappa}\checkmark \qquad \Xi(\ell) = \mathtt{return} \qquad \hat{\gamma} \supset \hat{\kappa}}{h(\hat{w}, \mathcal{L}(\mu(\hat{\kappa}), \hat{\gamma}))@\ell \cdot \hat{\kappa}\checkmark}$$

Leave operators

$$\mathit{lcont}(\hat{\kappa}) := \begin{cases} \mathit{undefined} & \text{if } \hat{\kappa} = \mathbf{0}@\ell \text{ or } \hat{\kappa} = h(\_)@\ell \cdot \hat{\kappa}' \\ \hat{\kappa}' & \text{if } \hat{\kappa} = \mathbf{exit}@\ell \cdot \hat{\kappa}' \\ \mathit{lcont}(\hat{\kappa}') & \text{if } \hat{\kappa} = \hat{\varsigma} \cdot \hat{\kappa}' \text{ and } \hat{\varsigma} \neq h(\_)@\ell \end{cases}$$
$$\mathit{laddr}(\hat{\kappa}) := \mu(\mathit{lcont}(\hat{\kappa}))$$

Configurations

$\langle \hat{M}, \hat{\gamma}, \hat{\kappa} \rangle\checkmark := \Xi(\mu(\hat{M})) = \lfloor \hat{M} \rfloor \wedge \nu(\hat{M}) = \mu(\hat{\kappa}) \wedge \hat{\gamma} \supset \hat{\kappa} \wedge \hat{\kappa}\checkmark$
$\langle \hat{v}, \hat{\gamma}, \hat{\kappa} \rangle\checkmark := \hat{v}\checkmark \wedge \hat{\gamma} \supset \hat{\kappa} \wedge \hat{\kappa}\checkmark$

Fig. 5. Augmented source validity.

the beginning of the continuation. Note that the pseudovariable LEAVE is always mapped to the register $\$\mathtt{lr}$, and every variable x is mapped to as many $\overline{\$\mathtt{x}}$ registers as it is necessary.

$$\Theta \vDash \langle \hat{M}, \hat{\gamma}, \hat{\kappa} \rangle := \Theta(\$\mathtt{pc}) = \mu(\hat{M}) \wedge \forall x \in \mathit{dom}(\hat{\gamma}).\overline{\Theta(\$\mathtt{x})} = \lfloor \hat{\gamma}(x) \rfloor$$
$$\Theta \vDash \langle v, \hat{\gamma}, \hat{\kappa} \rangle := \Theta(\$\mathtt{pc}) = \mu(\hat{\kappa}) \wedge \forall x \in \mathit{dom}(\hat{\gamma}).\overline{\Theta(\$\mathtt{x})} = \lfloor \hat{\gamma}(x) \rfloor$$

The soundness property we seek states that the modelling relation is a simulation: whenever we have a source transition $\hat{s} \hookrightarrow \hat{s}'$ and an AsmFX configuration $a \vDash \hat{s}$, then there exists a transition chain $a \overset{*}{\hookrightarrow} a'$ where $a' \vDash \hat{s}'$. However, this property cannot be proved for an arbitrary $\hat{s}$: augmented source configurations contain metadata that encodes part of the execution history; while the compiler produces correct metadata, and the abstract machine ensures that the metadata is propagated correctly, arbitrary configurations could be unsound and not allow the simulation we need for the soundness theorem. We thus need to restrict ourselves to *valid* configurations (denoted by $\hat{s}\checkmark$) and prove, as part of our theorem, that validity is preserved by transitions.

The validity predicate is defined in Figure 5: its role is to check that all the ASFX annotations are consistent with the compiled AsmFX program. A configuration is valid if its environment $\hat{\gamma}$ and continuation $\hat{\kappa}$ are valid and additionally the environment and the continuation are coherent (notation: $\hat{\gamma} \supset \hat{\kappa}$). Furthermore, we require that in a valid configuration whose control is a computation

$\hat{M}$, the AsmFX code at location $\mu(\hat{M})$ corresponds to the compiled code for $\hat{M}$, and that if instead the control is a value v, the value itself is valid.

A value is valid if any broom $\hat{k}$ syntactically contained in it is valid; an environment is valid if all the values it assigns to its domain are valid. A broom is valid if, when we resume in any valid environment γ' and continuation κ' such that the two are coherent, we get a valid continuation and, furthermore, the environment in the broom is coherent with the extended continuation.

What it means for a continuation to be valid is expressed by an inductive predicate, with a base rule for the identity continuation and a recursive rule for each frame type. These rules ensure that the annotation for each frame corresponds to suitable AsmFX code: for instance, the annotation for **jump** frames must reference a jump to the rest of the continuation. The theorem we prove states that transitions starting at valid ASFX configurations preserve validity and can be simulated:

Theorem 7.2. *Let P be a source program, and $\Xi = \lfloor P \rfloor$. For all source configurations $\hat{s}, \hat{s}'$ such that $\hat{s}\checkmark$ (valid with respect to Ξ), if $\hat{s} \hookrightarrow \hat{s}'$, then $\hat{s}'\checkmark$ and for all AsmFX configurations a such that $a \vDash \hat{s}$ there exists a' such that $a \overset{*}{\hookrightarrow} a'$ and $a' \vDash \hat{s}'$.*

The proof, detailed in the appendix, is by case analysis on the transition $\hat{s} \hookrightarrow \hat{s}'$.

7.3 Soundness

It is straightforward to show that the annotations produced by compilation are correct and therefore the initial configuration of the ASFX machine is related by simulation to the corresponding initial configurations of the SFX machine and the AsmFX machine.

Lemma 7.3. *If $\hat{M}$ is the result of annotating M according to the compiler, $\left|\hat{M}\right| = M$ and, consequently, $\langle M, [], \mathbf{0}\rangle \preccurlyeq \langle \hat{M}, [], \mathbf{0}@\mu(\hat{M})\rangle$.*

Lemma 7.4. *Let $\hat{P} = \hat{M}; \hat{\Sigma}$. If $\lfloor \hat{P} \rfloor = \Xi$, then $\Xi(\mu(\hat{M})) = \lfloor \hat{M} \rfloor$. Consequently $\langle \hat{M}, [], \mathbf{0}@\mu(\hat{M})\rangle\checkmark$ (with respect to Ξ).*

Lemma 7.5. *If $\Theta = [\$\mathsf{pc} \mapsto \mu(\hat{M})]$, then $\Xi \vDash \langle \hat{M}, [], \mathbf{0}@\mu(\hat{M})\rangle$. Consequently, $\langle \Xi, []\rangle \vdash \langle \hat{M}, [], \mathbf{0}@\mu(\hat{M})\rangle$.*

The soundness theorem then arises from Theorems 7.1 and 7.2 as a corollary:

Corollary 7.6. *Let $P = M; \Sigma$ be an SFX program and $s = \langle M, [], \mathbf{0}\rangle$. Let $\hat{P} = \hat{M}; \hat{\Sigma}$ be P annotated by the compiler, $\Theta = [\$\mathsf{pc} \mapsto \mu(\hat{M})]$, and $a = \langle \Theta, []\rangle$. Then if $s \overset{*}{\hookrightarrow} s'$, there exist $a', \hat{s}'$ such that $s' \preccurlyeq \hat{s}'$ and $a' \vDash \hat{s}'$.*

8 Discussion

We showed that AsmFX can correctly implement a large class of effect handlers. In devising our compiler we focused on the general case. However, there are many opportunities to provide alternate more efficient strategies in specific cases (exceptions, runners, tail-resumptive handlers, single-shot resumptions, etc.), whether identified by special syntax, static analysis, or dynamically at runtime. For instance, specialised compilation of tail-resumptive handlers could invoke a broom without setting up a new leave record (similarly to how tail-calls to functions are optimised to reuse the caller's stack frame). A similar optimisation is performed by existing effect handler implementations including Effekt, Koka, and libmprompt.

Another obvious optimisation concerns the implementation of functions. For the purposes of this paper it was convenient to treat functions as syntactic sugar for trivial handlers consisting of just a return clause: this trick is standard in the literature, however it may obfuscate the means by which functions are implemented. When we implement a function call as a **handle** expression, the compiler emits the code to set up a new handler frame, which includes a leave record: while the

handler functionality is trivial, the leave record plays the role of the call stack, thanks to its ability to store the caller's return address and environment (including the previous leave record). Indeed, it is easy to give a direct translation of functions to AsmFX that does not make use of handlers: all that is needed to implement functions in AsmFX is the functionality of the leave register `$lr`, with the instructions `loadl` and `exit` used to push and pop stack frames.

A desirable feature of the AsmFX implementation of handlers is that it naturally provides a certain level of memory protection: the state of a handler is stored in the effect context and cannot be directly accessed by other code fragments (including client code or other handlers); our compiler does not actively wipe registers when transitioning in and out of handler code, so (read only) information leak is still possible, but it is possible in principle to write a refined compiler that would ensure no such leak happens. Additionally, we believe AsmFX could be useful as an intermediate language, particularly for further translation to *hardware capability architectures* such as CHERI [29]. We conjecture that fine-grained compartmentalisation provided in such architectures could be used to provide an implementation of effect contexts and handler, continuation, and leave records ensuring that information can only be accessed by the code that owns it (and particularly separation between code that uses effects and code that implements them). Inspired by the *Cerise* program logic [11] and its assembler for an idealised capability architecture, we plan to eventually provide an effectful *CeriseFX* to explore the relationship between handlers and machine-level capabilities.

WasmFX [25] extends WebAssembly (Wasm) [14], a low-level portable bytecode-driven stack machine, with support for effect handlers. Though the design of AsmFX is inspired by WasmFX, the two systems have quite different characters and purposes: in particular, WasmFX is rightly matched to the existing Wasm stack model, where AsmFX aims to be agnostic about the underlying implementation of its primitives. OCaml 5 [28] incorporates a high-performance implementation of effect handlers based on low-level stack manipulation operations. The intended semantics of OCaml 5 effect handlers is given by a CEK-like abstract machine much like that of SFX. As with WasmFX, the OCaml 5 implementation is closely tied with underlying features of the host language, in this case delimited continuations and multicore fibers. C libraries such as `libseff` [2] and `libmpeff` [21] provide low-level effect handler implementations in C based respectively on coroutines and on multiprompt delimited control. It would be interesting to try to more formally relate systems such as WasmFX, OCaml 5, `libseff` and `libmprompt` with AsmFX.

Muhcu et al.[24] present an implementation technique for multi-shot resumptions in the context of *named* handlers [5, 31] and stack-allocated mutable state. Their work partially overlaps with ours as they too describe a compilation technique (building on prior work [22]); while we directly compile a high-level language with standard (unnamed) effect handlers to the effectful AsmFX, their work starts with an intermediate language with multi-prompt delimited control implementing named handlers (which they call "lexically-scoped handlers") and compiles it to an abstract assembly without primitive effect handlers, but with explicit memory allocation. A direct comparison between their work and AsmFX is challenging: not only are the two works based on different semantics for effect handlers, but they also have different purposes: AsmFX is concerned primarily with expressiveness and soundness, whereas [24] is geared towards implementation efficiency. Extending AsmFX to support named handlers would however be interesting: to achieve that, we could provide variants of the `resume` and `do` instructions respectively returning or accepting a handler name as an additional argument; the name may concretely be an index into the effect context (allowing efficient addressing of a handler frame), but the code should have no access to its representation.

Our proof of correctness of the SFX compiler relies on information on execution history that needs to be logically preserved step by step by our execution models. The ASFX language we devised to enrich source programs with annotations providing evidence for such information is reminiscent of other work in the field of *provenance* [4], studying metadata on the origin, history,

and derivation of information, and techniques to propagate it through computation [7, 13]. This is not accidental: it is easy to imagine that a compiler such as the one we have described could be written in a provenance-tracking language (for instance [27]): the compiler would then produce object code annotated with provenance information about the source code that yielded it; by reflecting that information back into SFX, we would get ASFX. Another related area is *justification logic* [3], a modal proof system where propositions can carry evidence of their validity. We have shown that these related concepts of origin, history, evidence, justification are not only important for their epistemological value, but provide an insight on logical constraints on the relationship between the input and output of a computation (in our case, source code and object code), which can be useful to validate the correctness of an algorithm: we regard this as one of the key contributions of this paper.

Acknowledgments

XXX: remember to acknowledge Daniel contributing to the awesome title!

Data-Availability Statement

We are making no distinct submission for Artifact Evaluation: the key artifact is AsmFX itself and the written proofs in the paper.

References

[1] Danel Ahman and Andrej Bauer. 2020. Runners in Action. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 29–55. doi:10.1007/978-3-030-44914-8_2

[2] Mario Alvarez-Picallo, Teodoro Freund, Dan R. Ghica, and Sam Lindley. 2024. Effect Handlers for C via Coroutines. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 358 (Oct. 2024), 28 pages. doi:10.1145/3689798

[3] Sergei Artemov. 2008. Justification Logic. In *Proceedings of the 11th European Conference on Logics in Artificial Intelligence* (Dresden, Germany) *(JELIA '08)*. Springer-Verlag, Berlin, Heidelberg, 1–4. doi:10.1007/978-3-540-87803-2_1

[4] David A. Bearman and Richard H. Lytle. 1985. The Power of the Principle of Provenance. *Archivaria* 21 (Jan. 1985), 14–27. https://archivaria.ca/index.php/archivaria/article/view/11231

[5] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.* 4, POPL, Article 48 (Dec. 2019), 29 pages. doi:10.1145/3371116

[6] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20 (2019), 28:1–6.

[7] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends® in Databases* 1, 4 (2009), 379–474. doi:10.1561/1900000006

[8] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. 2017. Concurrent systems programming with effect handlers. In *TFP*.

[9] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective Concurrency through Algebraic Effects. OCaml. http://kcsrk.info/papers/effects_ocaml15.pdf

[10] Matthias Felleisen and Daniel P. Friedman. 1986. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*. North-Holland, Amsterdam, 193–217.

[11] Aina Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2023. Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code. *J. ACM* 71 (09 2023). doi:10.1145/3623510

[12] Dan Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. 2022. High-level effect handlers in C++. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 183 (Oct. 2022), 29 pages. doi:10.1145/3563445

[13] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Beijing, China) *(PODS '07)*. Association for Computing Machinery, New York, NY, USA, 31–40. doi:10.1145/1265530.1265535

[14] Andreas Haas, Andreas Rossberg, Derek Schuff, Ben Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J.F. Bastien. 2017. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Notices* 52, 6 (June 2017), 185–200.

[15] Daniel Hillerström. 2015. *Handlers for Algebraic Effects in Links.* Master's thesis. University of Edinburgh, Scotland. http://project-archive.inf.ed.ac.uk/msc/20150206/msc_proj.pdf
[16] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, James Chapman and Wouter Swierstra (Eds.). ACM, 15–27. doi:10.1145/2976022.2976033
[17] Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect handlers via generalised continuations. *J. Funct. Program.* 30 (2020), e5. doi:10.1017/S0956796820000040
[18] Thomas Johnsson. 1985. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–203.
[19] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 145–158. doi:10.1145/2500365.2500590
[20] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), 100–126. doi:10.4204/eptcs.153.8
[21] Daan Leijen and KC Sivamarakrishnan. 2023. libmprompt and libmpeff. https://github.com/koka-lang/libmprompt.
[22] Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. 2024. Lexical Effect Handlers, Directly. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 330 (Oct. 2024), 29 pages. doi:10.1145/3689770
[23] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 271–283. doi:10.1145/237721.237791
[24] Serkan Muhcu, Philipp Schuster, Michel Steuwer, and Jonathan Immanuel Brachthäuser. 2025. Multiple Resumptions and Local Mutable State, Directly. *Proc. ACM Program. Lang.* 9, ICFP, Article 260 (Aug. 2025), 30 pages. doi:10.1145/3747529
[25] Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, K. C. Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 460–485. doi:10.1145/3622814
[26] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013), 36 pages. doi:10.2168/LMCS-9(4:23)2013
[27] Wilmer Ricciotti. 2017. A core calculus for provenance inspection. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming* (Namur, Belgium) *(PPDP '17)*. Association for Computing Machinery, New York, NY, USA, 187–198. doi:10.1145/3131851.3131871
[28] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 206–221. doi:10.1145/3453483.3454039
[29] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9).* Technical Report UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. doi:10.48456/tr-987
[30] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect handlers, evidently. *Proc. ACM Program. Lang.* 4, ICFP, Article 99 (Aug. 2020), 29 pages. doi:10.1145/3408981
[31] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (Jan. 2019), 29 pages. doi:10.1145/3290318

A An AsmFX compiler for SFX

We give here a detailed definition of the SFX compiler that we only sketched in Section 6. The compiler consists of three procedures, for values, computations, and handlers.

Compiling values. To express the compiler, we start by giving a procedure to compile SFX open values: the operation $\lfloor V \rfloor$ (Figure 6) produces code which will load an initial segment of the $\overline{\texttt{\$a}}$ registers of suitable length with the AsmFX representation of that value. We use blue symbols

to differentiate meta-operations such as the compiler and its definition from the object syntax it manipulates.

Compilation of values uses an auxiliary definition $\lfloor V \rfloor^n$ where the subvalues are loaded starting at register $\$\mathsf{a}_\mathsf{n}$The representation of constants $\lfloor c \rfloor$ is assumed as primitive. In the case of free variables x, we assume that the registers $\$\mathsf{x}_0, \ldots, \$\mathsf{x}_{\|x\|-1}$ contain the AsmFX representation of the value of x, and we move their content to the argument registers.

Compiling computations. The table in Figure 7 describes the compilation rules for each possible syntactic form of computations. Before discussing some relevant cases, we note the following preliminary facts:

- We assume that every SFX primitive operation $\oplus$ is correctly implemented by an AsmFX instruction $\lfloor \oplus \rfloor$, taking as many register arguments as needed to contain its input, and similarly storing its output in an a suitable number of output registers.
- Booleans are represented as integers in the obvious way.
- Instructions `j` *target_code* and `bz` *target_code*, $\mathsf{r} representing an unconditional jump to *target_code* and a branch to *target_code* if $\mathsf{r} is zero are available.
- The operation $\varphi(n, \mathbb{L})$ produces n labels fresh with respect to the used label set $\mathbb{L}$. The concrete choice of labels is irrelevant and consistent renaming is assumed to denote the same code (we use labels for the sake of readability, but each label must be considered notation for an absolute memory index).

We discuss the compilation rules in detail. To compile a **return**, we simply compile the returned value. A primitive operation $\oplus(\overline{V})$ is compiled by compiling V first, then executing the instruction $\lfloor \oplus \rfloor$ implementing $\oplus$ using the $\overline{\$\mathsf{a}}$ registers both as input and as output. Conditionals **if**(V, M, N) require us to compile V first and then use a `bz` "branch if zero" instruction to discriminate between true and false values: in one case we continue with the code for M, in the other case we branch to l_{else} and the code for N; at the end, both branches meet again at label l_{end}. A **let** $x \leftarrow M$ **in** N is compiled by compiling M first: after executing the compiled code for M, we will copy the result of that computation from the $\overline{\$\mathsf{a}}$ register where it has been stored into the $\overline{\$\mathsf{x}}$ registers, where N expects to find it; finally we compile N. The compilation rule for **unpack** simply needs compile its value argument and then move the result of its evaluation to the registers for the $\overline{x^k}$ variables (the rule makes use of an extended vector notation $\overline{\$\mathsf{x}_{\mathsf{m},\mathsf{n}}}$ which stands for n registers starting from $\$\mathsf{x}_\mathsf{m}$, i.e. $\$\mathsf{x}_\mathsf{m}, \$\mathsf{x}_{\mathsf{m}+1}, \ldots, \$\mathsf{x}_{\mathsf{m}+\mathsf{n}-1}$.

Now we consider computations for the effectful sublanguage. The U **with** $h(V)$ computation is compiled by compiling V first and storing the result of its evaluation in an initial segment of the $\overline{\$\mathsf{s}}$ registers; we use these registers together with the handler specification $\lfloor h \rfloor$ to load the handler $h(V)$ into register $\$\mathsf{hr}$; then we compile U: its value will be stored in $\$\mathsf{a}_0, \$\mathsf{a}_1$; finally, we replace the content of $\$\mathsf{a}_1$ with the content of $\$\mathsf{hr}$.

$$
\begin{array}{rcl}
\lfloor V \rfloor & := & \lfloor V \rfloor^0 \\
\lfloor x \rfloor^n & := & \$\mathsf{a}_\mathsf{n} \ldots \$\mathsf{a}_{\mathsf{n}+\|\mathsf{x}\|-1} \leftarrow \$\mathsf{x}_0 \ldots \$\mathsf{x}_{\|\mathsf{x}\|-1} \\
\lfloor c \rfloor^n & := & \$\mathsf{a}_\mathsf{n} \ldots \$\mathsf{a}_{\mathsf{n}+\|\mathsf{c}\|-1} \leftarrow \lfloor c \rfloor \\
\left\lfloor \langle \overline{V_p} \rangle \right\rfloor^n & := & \lfloor V_0 \rfloor^n; \lfloor V_1 \rfloor^{n+\|V_0\|}; \ldots; \left\lfloor V_{p-1} \right\rfloor^{n+\sum \|V_i\|_{i=0,\ldots,p-2}}
\end{array}
$$

Fig. 6. Compilation SFX values to AsmFX.

$$\lfloor \textbf{return}\ V \rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}' :=$$
$$\quad \lfloor V \rfloor$$
$$\quad \textbf{let}\ \mathbb{L}' = \mathbb{L}$$

$$\left\lfloor \begin{array}{l} \textbf{let}\ x \leftarrow M \\ \textbf{in}\ N \end{array} \right\rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}' :=$$
$$\quad \textbf{let}\ \mathbb{I}_M = \overline{\$\mathsf{x}_{\|M\|}}$$
$$\quad \lfloor M \rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}'_M$$
$$\quad \mathbb{I}_M \leftarrow \overline{\$\mathsf{a}_{\|M\|}}$$
$$\quad \lfloor N \rfloor^{\mathbb{I}\cup\mathbb{I}_M,\mathbb{L}'_M} \Rightarrow \mathbb{L}'$$

$$\left\lfloor \begin{array}{l} \textbf{unpack}\ \overline{x^k} \leftarrow V \\ \textbf{in}\ M \end{array} \right\rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}' :=$$
$$\quad \textbf{let}\ \mathbb{I}_M = \mathbb{I} \cup \{\overline{\$\mathsf{x}^{\mathsf{k}}_{\|\mathsf{x}_n\|}}\}$$
$$\quad \lfloor V \rfloor$$
$$\quad \overline{\$\mathsf{x}^0_{\|x^0\|}} \leftarrow \overline{\$\mathsf{a}_{\|x^0\|}}$$
$$\quad \vdots$$
$$\quad \overline{\$\mathsf{x}^i_{\|x^i\|}} \leftarrow \overline{\$\mathsf{a}_{\sum\|x^{i-1}\|,\sum\|x^i\|}}$$
$$\quad \vdots$$
$$\quad \overline{\$\mathsf{x}^{n-1}_{\|x^{n-1}\|}} \leftarrow \overline{\$\mathsf{a}_{\sum\|x^{n-2}\|,\sum\|x^{n-1}\|}}$$
$$\quad \lfloor M \rfloor^{\mathbb{I}_M,\mathbb{L}} \Rightarrow \mathbb{L}'$$

$$\lfloor U\ \textbf{with}\ h(V) \rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}' :=$$
$$\quad \lfloor V \rfloor$$
$$\quad \overline{\$\mathsf{s}_{\|V\|}} \leftarrow \overline{\$\mathsf{a}_{\|V\|}}$$
$$\quad \mathtt{loadh}\ \lfloor h \rfloor, \mathit{state} : (\overline{\$\mathsf{s}_{\|V\|}})$$
$$\quad \lfloor U \rfloor$$
$$\quad \$\mathsf{a}_1 \leftarrow \$\mathsf{hr}$$
$$\quad \textbf{let}\ \mathbb{L}' = \mathbb{L}$$

$$\lfloor \oplus(V) \rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}' :=$$
$$\quad \lfloor V \rfloor$$
$$\quad \overline{\$\mathsf{a}_{\|out(\oplus)\|}} \leftarrow \lfloor \oplus \rfloor\ \overline{\$\mathsf{a}_{\|V\|}}$$
$$\quad \textbf{let}\ \mathbb{L}' = \mathbb{L}$$

$$\lfloor \textbf{do}\ \#t(V) \rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}' :=$$
$$\quad \lfloor V \rfloor$$
$$\quad \mathtt{do}\ \#t, \mathit{save} : \mathbb{I}$$
$$\quad \textbf{let}\ \mathbb{L}' = \mathbb{L}$$

$$\lfloor \textbf{resume}\ U(V) \rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}' :=$$
$$\quad \textbf{let}\ l_{exit} = \varphi(1, \mathbb{L})$$
$$\quad \lfloor U \rfloor$$
$$\quad \$\mathsf{kr} \leftarrow \$\mathsf{a}_\emptyset$$
$$\quad \$\mathsf{hr} \leftarrow \$\mathsf{a}_1$$
$$\quad \lfloor V \rfloor$$
$$\quad \mathtt{loadl}\ \$\mathsf{lr}, l_{exit}, \mathit{save} : \mathbb{I}$$
$$\quad \mathtt{resume}$$
$$l_{exit} :$$
$$\quad \textbf{let}\ \mathbb{L}' = \mathbb{L}$$

$$\left\lfloor \begin{array}{l} \textbf{newbroom}\ f \\ \textbf{with}\ h(V) \end{array} \right\rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}' :=$$
$$\quad \textbf{let}\ l_{res}, l_{exit}, l_{end} = \varphi(3, \mathbb{L})$$
$$\quad \textbf{let}\ \mathbb{L}' = \mathbb{L} \cup \{l_{res}, l_{exit}, l_{end}\}$$
$$\quad \lfloor V \rfloor$$
$$\quad \overline{\$\mathsf{s}_{\|V\|}} \leftarrow \overline{\$\mathsf{a}_{\|V\|}}$$
$$\quad \mathtt{loadh}\ \lfloor h \rfloor, \mathit{state} : (\overline{\$\mathsf{s}_{\|V\|}})$$
$$\quad \mathtt{loadk}\ l_{res}$$
$$\quad \$\mathsf{a}_\emptyset, \$\mathsf{a}_1 \leftarrow \$\mathsf{kr}, \$\mathsf{hr}$$
$$\quad \mathtt{j}\ l_{end}$$
$$l_{res} :$$
$$\quad \mathtt{loadl}\ l_{exit}$$
$$\quad \mathtt{j}\ f_{\mathbf{return}}$$
$$l_{exit} :$$
$$\quad \mathtt{return}$$
$$l_{end}$$

$$\lfloor \textbf{if}(V, M, N) \rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}' :=$$
$$\quad \textbf{let}\ l_{end}, l_{else} = \varphi(2, \mathbb{L})$$
$$\quad \textbf{let}\ \mathbb{L}_M = \mathbb{L} \cup \{l_{end}, l_{else}\}$$
$$\quad \lfloor V \rfloor$$
$$\quad \mathtt{bz}\ l_{else}, \$\mathsf{a}_\emptyset$$
$$\quad \lfloor M \rfloor^{\mathbb{I},\mathbb{L}_M} \Rightarrow \mathbb{L}'_M$$
$$\quad \mathtt{j}\ l_{end}$$
$$l_{else} :$$
$$\quad \lfloor N \rfloor^{\mathbb{I},\mathbb{L}'_M} \Rightarrow \mathbb{L}'$$
$$l_{end} :$$

$$\left\lfloor \begin{array}{l} \textbf{handle}\ M \\ \textbf{with}\ h(V) \end{array} \right\rfloor^{\mathbb{I},\mathbb{L}} \Rightarrow \mathbb{L}' :=$$
$$\quad \textbf{let}\ l_{cli}, l_{exit} = \varphi(2, \mathbb{L})$$
$$\quad \textbf{let}\ \mathbb{L}_M = \mathbb{L} \cup \{l_{cli}, l_{exit}\}$$
$$\quad \textbf{let}\ \mathbb{I}_h = \overline{\$\mathsf{s}_{\|V\|}}$$
$$\quad \lfloor V \rfloor$$
$$\quad \mathbb{I}_h \leftarrow \overline{\$\mathsf{a}_{\|V\|}}$$
$$\quad \mathtt{loadh}\ \$\mathsf{hr}, \lfloor h \rfloor, \mathit{state} : \mathbb{I}_h$$
$$\quad \mathtt{loadk}\ \$\mathsf{kr}, l_{cli}, \mathit{save} : ()$$
$$\quad \mathtt{loadl}\ \$\mathsf{lr}, l_{exit}, \mathit{save} : \mathbb{I}$$
$$\quad \mathtt{resume}$$
$$l_{cli} :$$
$$\quad \lfloor M \rfloor^{\mathbb{I},\mathbb{L}_M} \Rightarrow \mathbb{L}'_M$$
$$\quad \mathtt{return}$$
$$l_{exit} :$$
$$\quad \textbf{let}\ \mathbb{L}' = \mathbb{L}'_M$$

Fig. 7. Compilation of SFX computations to AsmFX.

To create a continuation from a function by means of **newbroom**(f) **with** hV, we need to create a suitable broom and place it in register $\$\mathsf{a}_\emptyset, \$\mathsf{a}_1$. We first compile V, place its content in the $\overline{\$\mathsf{s}}$ registers, and use it together with $\lfloor h \rfloor$ to create a handler which will be loaded into $\$\mathsf{hr}$ first and then $\$\mathsf{a}_1$. Then we construct a continuation pointing to a label l_{res} which will contain code to invoke f: the continuation is initially stored in $\$\mathsf{kr}$, then moved into $\$\mathsf{a}_\emptyset$. Finally we jump to l_{end}, the end of the code. At location l_{res}, we to call f we actually need to invoke its return clause: this is done by jumping unconditionally to the $f_{\mathbf{return}}$ address, after installing in $\$\mathsf{lr}$ a leave record indicating what code should be executed when f terminates — f is executed within the handler $h(V)$, therefore at the end of its execution it should invoke the return clause for $h(V)$: that corresponds to the instruction `return`, at the label l_{exit}.

To invoke an effect by means of **do** $\#t(V)$, we first compile V: this places the result of its evaluation into the $\overline{\$\mathsf{a}}$, where the operation handler expects to find it. To invoke the operation handler, we

$$
\left\lfloor
\begin{array}{l}
\textbf{handler}\ h\ (x)\{ \\
\quad \textbf{return}\ y \to N^h_{\textbf{return}} \\
\quad |\#t_1(z^1) \to (r^1).N^h_{\#t_1} \\
\quad |\ldots\}
\end{array}
\right\rfloor^{\mathbb{L}} \Rightarrow \mathbb{L}' \quad = \quad
\left\{
\begin{array}{l}
\quad \textbf{let}\ h_{\textbf{return}}, \overline{h_{\#t}} = \varphi(1 + \|\overline{\#t}\|) \\
\quad \textbf{let}\ \mathbb{L}_{\textbf{return}} = \mathbb{L} \cup \{h_{\textbf{return}}, \overline{h_{\#t}}\} \\
\quad \textbf{let}\ \mathbb{I}_x = \overline{\$\mathtt{x}_{\|x\|}} \\
h_{\textbf{return}}: \\
\quad \textbf{let}\ \mathbb{I}_{N^h_{\textbf{return}}} = \overline{\$\mathtt{y}_{\|y\|}} \\
\quad \mathbb{I}_x \leftarrow \overline{\$\mathtt{s}_{\|x\|}} \\
\quad \mathbb{I}_{N^h_{\textbf{return}}} \leftarrow \overline{\$\mathtt{a}_{\|y\|}} \\
\quad \left\lfloor N^h_{\textbf{return}} \right\rfloor^{\mathbb{I}_x \cup \mathbb{I}_{N^h_{\textbf{return}}} \cup \{\$\mathtt{lr}\}, \mathbb{L}_{\textbf{return}}} \Rightarrow \mathbb{L}_{\#t_1} \\
\quad \mathtt{exit} \\
h_{\#t_1}: \\
\quad \textbf{let}\ \mathbb{I}_{N^h_{t_1}} = \overline{\$\mathtt{z}^1_{\|z\|}}, \$\mathtt{r}^1_1, \$\mathtt{r}^1_2 \\
\quad \mathbb{I}_x \leftarrow \overline{\$\mathtt{s}_{\|x\|}} \\
\quad \mathbb{I}_{N^h_{\#t_1}} \leftarrow \overline{\$\mathtt{a}_{\|z\|}}, \$\mathtt{kr}, \$\mathtt{hr} \\
\quad \left\lfloor N^h_{\#t_1} \right\rfloor^{\mathbb{I}_x \cup \mathbb{I}_{N^h_{\#t_1}} \cup \{\$\mathtt{lr}\}, \mathbb{L}_{\#t_1}} \Rightarrow \mathbb{L}_{\#t_2} \\
\quad \mathtt{exit} \\
\vdots \\
\mathbb{L}' := \mathbb{L}_{\#t_{n+1}}
\end{array}
\right.
$$

Fig. 8. Compilation of SFX handlers to AsmFX.

just execute the instruction do $\#t$, *save* $:\mathbb{I}$ (saving all the input registers will allow the resumption for the $\#t$ operation to restore the original client environment).

To execute a computation under a new handler by means of **handle** M **with** $h(V)$, we compile V, dopy its evaluation in the $\overline{\$\mathtt{s}}$ registers, and use it together with $\lfloor h \rfloor$ to create a handler and place it in $\$\mathtt{hr}$. We also create a continuation pointing to the code for M (l_{cli}) and place it in $\$\mathtt{kr}$, and a leave record for the code to be executed after this computation (l_{exit}), with saved registers corresponding to the input set $\mathbb{I}$). After these three registers have been loaded, we invoke the client by resuming the continuation we created, using the instruction `resume`. Notice that the client code at location l_{cli} is terminated by a `return` instruction, so that the result of M will be delivered to the return clause of the newly installed handler: we call this a **return** epilogue. Tracking epilogues is essential to prove the correctness of the compiler.

A **resume** $U(V)$ works similarly: registers $\$\mathtt{kr}$ and $\$\mathtt{hr}$ are filled with the result of the evaluation of U; the result of the evaluation of V is put in registers $\overline{\$\mathtt{a}}$; we create a leave record for the code l_{exit} to be run after executing this computation (again, the input registers $\mathbb{I}$ must be added to the save area of $\$\mathtt{lr}$). Finally, we invoke the `resume` instruction.

Compiling handlers. The procedure to compile handlers is detailed in Figure 8. The return and operation clauses of each handler are compiled separately using the same algorithm defined for computations. In particular, for each handler h with clauses for operations $\overline{\#t_i}$ we generate unique global labels $h_{\textbf{return}}, \overline{h_{\#i}}$ for the code implementing the operations. Each parameterised handler with x as its formal parameter will receive its state in an initial segment of the state registers $\overline{\$\mathtt{s}}$ of suitable size, and will allocate variable registers $\overline{\$\mathtt{x}}$ corresponding to the SFX parameter variable x to copy the state into (each clause performs this copy separately).

The return clause $h_{\textbf{return}}$ and all operation clauses $h_{\#t_i}$ receive their arguments in an initial segment of the argument registers $\overline{\$\mathtt{a}}$ of suitable size: those are also copied into variable registers $\overline{\$\mathtt{y}}$, $\overline{\$\mathtt{z}^{\mathtt{i}}}$ corresponding to the formal arguments y (for the return clause) and z_i (for the $\#t_i$ operation

clause). Operation clauses will also copy the content of registers $\texttt{\$kr}, \texttt{\$hr}$ (which contain the broom resumption for the operation) to variable registers $\texttt{\$r}^{\texttt{i}}_1, \texttt{r}^{\texttt{i}}_2$ corresponding to the SFX broom variable r^i.

Then the SFX computation for each clause $N^h_{\mathbf{return}}$, $N^h_{\#t_i}$ is compiled by providing the suitable set of input registers, which must contain the leave register $\texttt{\$lr}$, as this should not be accidentally overwritten.

The handler result (returned when exiting the handler, either normally through the return clause or abnormally through an operation clause) is stored in an initial segment of the argument registers $\overline{\texttt{\$a}}$. Each clause is terminated by an `exit` instruction which triggers the leave register; this instruction also constitutes an epilogue that needs to be accounted for in the soundness proof.

B Soundness of compilation

As a final step in our effort to assess the expressiveness of AsmFX, we need to prove the correctness of our compiler. The most basic correctness property would state that for any SFX program P and its AsmFX counterpart $\lfloor P \rfloor$, if P terminates returning a value v, then $\lfloor P \rfloor$ also terminates and a machine representation of v can be found in an initial segment of the $\overline{\texttt{\$a}}$ registers.

In fact, we are also interested in proving the correctness of partial executions: to show this, we need to prove that the transition system induced by the AsmFX machine correctly simulates the transition system induced by the SFX abstract machine.

The proof is not entirely straightforward. The main difficulty lies in the fact that executing an SFX program consumes part of the program itself, blurring the correspondence between source and compiled object code beyond recognisability: the code resulting from the recompilation of a partially executed program will not match, even partially, the code of the original program, because parts of the code are completely lost and thus ignored by the compiler. To match partially executed source code to partially executed AsmFX code we need a more sophisticated instrument than the compiler itself.

Our solution is to annotate key parts of the source terms with the memory locations where they were originally placed by the compiler. If we have enough annotations, we may reconstruct a location-preserving compiled code which will partially match the original AsmFX code for the full program.

B.1 Augmenting SFX

We introduce the augmented source language ASFX: a conservative extension of SFX embedding location information and other information about execution history. Compared to SFX, ASFX sports an enriched syntax and a specific abstract machine clearly derived from its SFX counterpart, but presents some key differences that makes it mimic AsmFX code more closely. For our goals, we do not need to provide a type system for ASFX.

The syntax of ASFX is shown in Figure 9. Augmented values annotate SFX values with a single location referencing the code responsible for loading the value in the appropriate registers (an instance of $\lfloor V \rfloor$). Augmented computations $\hat{M}$ are isomorphic to a plain M, but every subcomputation (including $\hat{M}$ itself, save for the case of **return**) is annotated with a start location. For every $\hat{V}$ and $\hat{M}$ we define their start locations $\mu(V)$ and $\mu(M)$ as its top-level annotation (in particular, $\mu(\mathbf{return}\ V@\ell) = \mu(V@\ell) = \ell$). We also define the end location of a value or computation as the compiled length of that value or computation plus its start location:

$$\nu(\hat{V}) = \mu(\hat{V}) + \mathit{length}(\lfloor V \rfloor)$$
$$\nu(\hat{M}) = \mu(\hat{M}) + \mathit{length}(\lfloor M \rfloor)$$

$$
\begin{array}{lrcl}
\text{Open values:} & \hat{U}, \hat{V}, \hat{W} & ::= & V@\ell \\
\text{Computations:} & \hat{M}, \hat{N} & ::= & \mathbf{return}\ \hat{V} \mid \oplus(\hat{V})@\ell \mid \mathbf{if}(\hat{V}, \hat{M}, \hat{N})@\ell \\
& & \mid & (\mathbf{let}\ x \leftarrow \hat{M}\ \mathbf{in}\ \hat{N})@\ell \mid (\mathbf{unpack}\ \overline{x} \leftarrow \hat{V}\ \mathbf{in}\ \hat{M})@\ell \\
& & \mid & (\mathbf{do}\ \ \#t(\hat{V}))@\ell \mid (\mathbf{handle}\ \hat{M}\ \mathbf{with}\ h(\hat{V}))@\ell \\
& & \mid & (\mathbf{resume}\ \hat{U}(\hat{V}))@\ell \mid (\hat{U}\ \mathbf{with}\ h(\hat{W}))@\ell \\
& & \mid & (\mathbf{newbroom}@\ell_{exit}(f)\ \mathbf{with}\ h(\hat{W}))@\ell \\
\text{Literal values:} & \hat{u}, \hat{v}, \hat{w} & ::= & c \mid \langle \overline{\hat{v}} \rangle \mid \hat{k} \\
\text{Environments:} & \hat{\gamma} & ::= & \overline{x \mapsto \hat{v}} \\
\text{Literal cont.:} & \hat{k} & ::= & (\hat{\kappa}, h(\hat{v}), \hat{\gamma}) \\
\text{Continuations:} & \hat{\kappa}, \hat{\kappa}' & ::= & \mathbf{0}@\ell \mid \hat{\varsigma}@\ell \cdot \hat{\kappa} \\
\text{Frames:} & \hat{\varsigma}, \hat{\varsigma}' & ::= & \varepsilon \mid (x).\hat{M} \mid h(\hat{v}, \mathcal{L}(\ell, \hat{\gamma})) \\
\text{Epilogues:} & \varepsilon & ::= & \mathbf{jump} \mid \mathbf{exit} \mid \mathbf{leave} \\
\text{Handlers:} & \hat{H} & ::= & \mathbf{handler}\ h(x)\ \{\mathbf{return}\ y \to \hat{M} \mid \overline{\#t(z) \to (r).\hat{N}}\} \\
\text{Specifications:} & \hat{\Sigma} & ::= & \overline{\hat{H}} \\
\text{Programs:} & P & ::= & \hat{\Sigma}; \hat{M}
\end{array}
$$

Fig. 9. ASFX: syntax.

Compared to SFX, ASFX continuations have some important differences: the identity continuation and all frames are annotated with a start location; additionally, pure frames $((x).\hat{M})@\ell$ contain a naked computation instead of a computation closure; we also introduce *epilogues* ε, a type of frame that has no counterpart in SFX but allows us to more closely track the execution of the compiled AsmFX code. The start location $\mu(\hat{\kappa})$ of a continuation κ is defined as the start location of its first frame, or in the case of the identity continuation as $\mu(\mathbf{0}@\ell) = \ell$. We do not define the end location of a frame or of a continuation.

The closures missing from pure frames resurface as an additional environment $\hat{\gamma}$ component of ASFX brooms: this structure must not be understood as a closure, in the sense that the environment does not apply to the whole broom: it is set as the initial environment when the broom is resumed, and it may be extended or entirely removed before the execution of the continuation is completed.

The other syntactic categories of ASFX are analogous to their SFX counterparts, but use the augmented versions of each nested expression.

B.2 An abstract machine for the augmented source language

Much like we did for SFX, we formalise the semantics of ASFX by means of an abstract machine. The main difference between the two abstract machines is that the ASFX machine is aware of location annotations and propagates them during its execution, ensuring that all the relevant information is preserved. As we previously mentioned, the definition of continuations is also different.

The full definition is given in Figure 10. Note that we omit the top-level location of the augmented code because it is irrelevant to the machine, however the subexpressions contain locations that are used in the transition rules.

The environment can now reference a *pseudovariable* LEAVE that has no counterpart in SFX, but is analogous to the leave records of AsmFX. Note the invariant that the evaluation of a certain computation always terminates with an environment that is an extension of the original one: for this reason, the evaluation of primitive operations terminates with the original $\hat{\gamma}$ instead of the empty environment [] — this was irrelevant in SFX, but it is crucial in ASFX to ensure the

control	env.	continuation	$\hookrightarrow$	control	env.	continuation
$\oplus(\hat{V})$	$\hat{\gamma}$	$\hat{\kappa}$		$[\![\oplus]\!]\,([\![\hat{V}]\!]\,\hat{\gamma})$	$\hat{\gamma}$	$\hat{\kappa}$
if $(\hat{V}, \hat{M}, \hat{N})$ where: $[\![\hat{V}]\!]\,\hat{\gamma} = \mathbf{true}$	$\hat{\gamma}$	$\hat{\kappa}$		$\hat{M}$	$\hat{\gamma}$	**jump**@$\nu(\hat{M})$ $\cdot\hat{\kappa}$
if $(\hat{V}, \hat{M}, \hat{N})$ where: $[\![\hat{V}]\!]\,\hat{\gamma} = \mathbf{false}$	$\hat{\gamma}$	$\hat{\kappa}$		$\hat{N}$	$\hat{\gamma}$	$\hat{\kappa}$
let $x \leftarrow \hat{M}$ **in** $\hat{N}$	$\hat{\gamma}$	$\hat{\kappa}$		$\hat{M}$	$\hat{\gamma}$	$((x).\hat{N})$@$\nu(\hat{M})$ $\cdot\hat{\kappa}$
unpack $\overline{x_n} \leftarrow \langle\overline{\hat{V}_n}\rangle$ **in** $\hat{N}$	$\hat{\gamma}$	$\hat{\kappa}$		$\hat{N}$	$\overline{(x_n \mapsto [\![\hat{V}_n]\!]\,\hat{\gamma})}$ $\cdot\hat{\gamma}$	$\hat{\kappa}$
handle $\hat{M}$ **with** $h(\hat{V})$	$\hat{\gamma}$	$\hat{\kappa}$		$\hat{M}$	$\hat{\gamma} \setminus \mathsf{LEAVE}$	$h(v, \mathcal{R}_{\mathcal{L}})$@$\nu(\hat{M}) \cdot \hat{\kappa}$ where: $v = [\![\hat{V}]\!]\,\hat{\gamma}$ $\mathcal{R}_{\mathcal{L}} = \mathcal{L}(\mu(\hat{\kappa}), \hat{\gamma})$
do $\#t(\hat{V})$	$\hat{\gamma}$	$\hat{\kappa}^{-t} \cdot h^{+t}(\hat{w}, \mathcal{R}_{\mathcal{L}})$@$\ell \cdot \hat{\kappa}'$		$h_{\#t}$	$(\mathsf{LEAVE} \mapsto \mathcal{R}_{\mathcal{L}})$ $\cdot(res(\#t) \mapsto$ $\quad(\hat{\kappa}^{-t}, h^{+t}(\hat{w})$@$\ell, \hat{\gamma}))$ $\cdot(dom(\#t) \mapsto [\![\hat{V}]\!]\,\hat{\gamma})$ $\cdot(dom(h) \mapsto \hat{w})$	**exit**@$\nu(h_{\#t}) \cdot \hat{\kappa}'$
resume $\hat{U}(\hat{V})$ where: $[\![\hat{U}]\!]\,\hat{\gamma} =$ $\quad(\hat{\kappa}', h(\hat{w})$@$\ell, \hat{\gamma}')$	$\hat{\gamma}$	$\hat{\kappa}$		$[\![\hat{V}]\!]\,\hat{\gamma}$	$\hat{\gamma}'$	$\hat{\kappa}'$ $\cdot h(\hat{w}, \mathcal{R}_{\mathcal{L}})$@$\ell$ $\cdot\hat{\kappa}$ where: $\mathcal{R}_{\mathcal{L}} = \mathcal{L}(\mu(\hat{\kappa}), \hat{\gamma})$
$\hat{U}$ **with** $h(\hat{W})$ where: $[\![\hat{U}]\!]\,\hat{\gamma} =$ $\quad(\hat{\kappa}', \_$@$\ell, \hat{\gamma}')$	$\hat{\gamma}$	$\hat{\kappa}$		$(\hat{\kappa}',$ $h([\![\hat{W}]\!]\,\hat{\gamma})$@$\ell,$ $\hat{\gamma}')$	$\hat{\gamma}$	$\hat{\kappa}$
newbroom@$\ell(f)$ **with** $h(\hat{V})$ where: $dom(f) = x$ $\ell_f = \mu(f_{\text{return}})$ $\ell'_f = \nu(f_{\text{return}})$ $\ell' = \ell + 1$ $\ell'' = \ell + 2$	$\hat{\gamma}$	$\hat{\kappa}$		$(\mathbf{leave}$@ℓ $\cdot\mathbf{jump}$@ℓ' $\cdot((x).f_{\text{return}})$@$\ell_f$ $\cdot\mathbf{exit}$@$\ell'_f,$ $h([\![\hat{V}]\!]\,\hat{\gamma})$@$\ell'',$ $[])$	$\hat{\gamma}$	$\hat{\kappa}$
return $\hat{V}$	$\hat{\gamma}$	$\hat{\kappa}$		$[\![\hat{V}]\!]\,\hat{\gamma}$	$\hat{\gamma}$	$\hat{\kappa}$
$\hat{v}$	$\hat{\gamma}$	**jump**@$\ell \cdot \hat{\kappa}$		$\hat{v}$	$\hat{\gamma}$	$\hat{\kappa}$
$\hat{v}$	$\hat{\gamma}$	**exit**@$\ell \cdot \hat{\kappa}$		$\hat{v}$	$\hat{\gamma}(\mathsf{LEAVE}).env$	$\hat{\kappa}$ where: $\mu(\hat{\kappa})$ $\quad= \hat{\gamma}(\mathsf{LEAVE}).loc$
$\hat{v}$	$\hat{\gamma}$	**leave**@$\ell \cdot \hat{\kappa}$		$\hat{v}$	$(\mathsf{LEAVE} \mapsto \mathcal{L}(\ell{+}2.[]))$ $\cdot\hat{\gamma}$	$\hat{\kappa}$
$\hat{v}$	$\hat{\gamma}$	$((x).\hat{N})$@$\ell \cdot \hat{\kappa}$		$\hat{N}$	$(x \mapsto \hat{v}) \cdot \hat{\gamma}$	$\hat{\kappa}$
$\hat{v}$	$\hat{\gamma}$	$h(\hat{w}, \mathcal{R}_{\mathcal{L}})$@$\ell \cdot \hat{\kappa}$		h_{return}	$(\mathsf{LEAVE} \mapsto \mathcal{R}_{\mathcal{L}})$ $\cdot(dom(h_{\text{return}}) \mapsto \hat{v})$ $\cdot(dom(h) \mapsto \hat{w})$	**exit**@$\nu(h_{\text{return}}) \cdot \hat{\kappa}$

Fig. 10. ASFX: abstract machine

correctness of the evaluation of **let** bindings despite the fact that we removed closures from pure frames.

Beside the change in pure frames, ASFX introduces epilogue frames, used in the target of the rules for **if** (when the guard evaluates to **true**), for **do**, for **resume**, and in the value case when the continuation starts with a handler frame. Epilogues do not correspond to any code in SFX, but do correspond to AsmFX code that needs to be executed after evaluating a certain computation, and before moving to the next one. In the case of the guard of an **if** being evaluated to true, after

evaluating the *then* branch $\hat{M}$ and before moving to the next computation, we need to jump over the code for the *else* branch: this is why we add to the continuation a **jump** epilogue, which will be located at $\nu(\hat{M})$ (immediately after the code for $\hat{M}$: this fine grained analysis will help greatly in matching SFX code, where control flow is dictated in part by the syntactic structure, and AsmFX, which is unstructured and whose control flow depends exclusively on explicit instructions.

The rule for **handle** is similar to the one in SFX, but it clears the content of LEAVE because the AsmFX implementation trashes the original content of register `$lr` (if any). The new handler frame receives an annotation $\nu(\hat{M})$, reflecting the fact that for all computation states, the end location of the current computation must match the start location of the current continuation.

Execution of a **do** differs from the corresponding rule in SFX in that the target environment has an updated resumption value, which saves the source environment $\hat{\gamma}$, ready to be restored when the resumption is triggered (SFX does not need to do this because all pure frames have an environment, and all other frames do not care about the environment). We also provide a value for LEAVE, the leave record of the handler being invoked, which is meaningful to the **exit** epilogue in the target continuation: **exit** corresponds to the `exit` instruction at the end of the AsmFX code for a handler operation clause or return clause: when we reach such an instruction, or equivalently when the epilogue **exit** is activated, control is transferred to the code referenced by that leave record.

The computation **resume** $\hat{U}(\hat{V})$ is evaluated similarly to SFX, but the environment is replaced in the target with the one found in the broom U, whereas the original environment is saved in the leave record linked to the broom handle that is being added to the target continuation. Broom handle replacement $\hat{U}$ **with** $\hat{V}$ is similar to its SFX counterpart, but it needs to take account for the additional environment field of augmented brooms.

The rule for **newbroom**@$\ell(f)$ **with** $h(\hat{V})$ is more complex than its SFX counterpart: while it does create a broom whose head is the body of f and whose handle is the specified handle h, it also needs to decorate the head with a number epilogues for administrative instructions that are executed either in the AsmFX code for **newbroom** (**leave**, **jump**) or at the end of the function f (**exit**).

The transition for **return** simply substitutesd the environment into the return value to obtain a closed value. Then we have four different transitions for states examining a closed value, one for each possible frame at the start of the continuation:

- If the continuation starts with a **jump** epilogue, the epilogue is simply removed from the continuation, reflecting the fact that we have jumped to the next frame.
- In the case of an **exit** epilogue, we similarly remove the epilogue from the continuation, but we also replace the environment according to the active leave record, reflecting the execution of an `exit` instruction.
- A **leave** epilogue corresponds to an AsmFX `loadl` instruction loading a distinguished address (two positions after the instruction itself) into the current leave record.
- When the continuation starts with a pure frame, we are passing the value under consideration as an argument to the computation $\hat{N}$ enclosed in the pure frame: the corresponding transition rule works similarly as in SFX, however since augmented pure frames do not come with an environment, we must assume that the source environment is compatible with $\hat{N}$.
- Finally if we are returning a value to a handler frame, we are really invoking its return clause: just like in SFX, we will continue execution with the return branch of that handler. Compared with SFX, we need to load the leave record of the handler into LEAVE, and also extend the target continuation with an **exit**, to denote the fact that when the evaluation of

the return clause terminates, one has to move to code external to the handler itself. Again, this corresponds to an `exit` instruction placed at the end of the return clause.

B.3 Well-scoping

We can prove that single step transitions in SFX are simulated by multistep transitions in ASFX. Since ASFX epilogues essentially correspond to no operation at all in SFX, the main challenge of this proof is to show that in spite of a different approach to handling environments, the ASFX provides a correct implementation of scopes. The simulation relation, summarised in Figure 4, essentially expresses a well-scoping invariant.

When an SFX configuration s is simulated by an ASFX configuration $\hat{s}$, we write $s \preccurlyeq \hat{s}$. The relation is defined by means of auxiliary definitions for the simulation of values, environments and continuations. There are two different cases based on whether the control element of the two configurations is a computation or a closed value: in the first case, $\langle M, \gamma, \kappa\rangle \preccurlyeq \langle \hat{M}, \hat{\gamma}, \hat{\kappa}\rangle$ means that if we erase all annotations from $\hat{M}$, we obtain M; additionally, the augmented environment $\hat{\gamma}$ simulates the simple environment γ, and the pair of augmented environment and continuation $(\hat{\gamma}, \hat{\kappa})$ simulates the simple continuation κ. If instead we are in the value case, we have that $\langle v, \gamma, \kappa\rangle \preccurlyeq \langle \hat{v}, \hat{\gamma}, \hat{\kappa}\rangle$ if and only if the augmented value $\hat{v}$ simulates the simple value v, and the pair of augmented environment and continuation $(\hat{\gamma}, \hat{\kappa})$ simulates γ (however γ need not be simulated by $\hat{\gamma}$). It may seem odd that the continuation simulation requires, on the augmented side, an environment: this is explained by the fact that SFX pure frames are closed with respect to an environment, while ASFX frames are not.

The value simulation $v \preccurlyeq \hat{v}$, for non-broom values, simply states that erasure of the augmented value results in the simple value; however, broom simulation $(\kappa, h(w)) \preccurlyeq (\hat{\kappa}, h(\hat{w})@\ell, \hat{\gamma}_\kappa)$ is recursively defined: it requires that κ be simulated by $(\hat{\gamma}_\kappa, \hat{\kappa})$ and that w be simulated by $\hat{w}$.

The environment simulation $\gamma \preccurlyeq \hat{\gamma}$ is defined pointwise for all variables in the domain of γ; $\hat{\gamma}$ is allowed to be defined on more variables.

The continuation simulation $\kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa})$ is defined by means of an inductive judgement, with a base rule for the identity continuation and recursive rules for each type of frame. A **jump** epilogue frame preserves the simulation; an **exit** frame swaps the active augmented environment with the one contained in the LEAVE pseudovariable (which must be defined). A **leave** epilogue requires the next continuation $\hat{\kappa}$ to simulate the original plain continuation κ in an environment with an updated leave record. In the case of handler frames, both continuations must start with matching handler frames: if that is the case, the simulation holds if there is a simulation between the remaining parts of the two continuations; however on the ASFX side of the simulation, we will need to swap the environment with the one contained in the leave record of the handler frame, reflecting the fact that when we exit a handler, the environment is restored from that leave record.

Finally, the most interesting case is that of pure frames. If both continuations start with a pure frame, in order for the simulation to hold, the computations N and $\hat{N}$ in each pure frame must match (meaning that by erasing $\hat{N}$ we get N). Furthermore, the SFX pure frame is closed by an environment γ_N: that environment must be simulated by the ASFX environment $\hat{\gamma}$, reflecting the fact that in ASFX we do not need an environment in the pure frame, because the active environment is already the correct one. If these conditions hold, the simulation is valid if there is a simulation between the two remaining parts of the two continuations; however on the ASFX side we need to extend the environment with a value for the variable x expected by the pure continuation, and the simulation must hold independently of the value chosen.

We now move to proving the simulation theorem.

B.3.1 Proof of Theorem 7.1. For all SFX configurations s, s' such that $s \hookrightarrow s'$, for all ASFX configurations $\hat{s}$ such that $s \preccurlyeq \hat{s}$, there exists an ASFX configuration $\hat{s}'$ such that $s' \preccurlyeq \hat{s}'$ and $\hat{s} \hookrightarrow \hat{s}'$.

In the proof, we will require the following auxiliary results.

Lemma B.1. *Suppose $\kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa})$ and $\hat{\gamma} \subseteq \hat{\gamma}'$: then we have $\kappa \preccurlyeq (\hat{\gamma}', \hat{\kappa})$.*

Proof. By induction on the well-scopedness judgement. □

Lemma B.2. *If $\gamma \preccurlyeq \hat{\gamma}$, then $\gamma \preccurlyeq \hat{\gamma} \setminus \mathsf{LEAVE}$.*

Proof. Trivial by unfolding the definitions. □

Lemma B.3. *Suppose $\kappa_1 \preccurlyeq (\hat{\gamma}_1, \hat{\kappa}_1)$, $\kappa_2 \preccurlyeq (\hat{\gamma}_2, \hat{\kappa}_2)$, and $|\hat{w}| = w$. Then we have $\kappa_1 \cdot h(w) \cdot \kappa_2 \preccurlyeq (\hat{\gamma}_1, \kappa_1 \cdot h(w, \mathcal{L}(\mu(\hat{\kappa}_2), \hat{\gamma}_2))@\ell \cdot \hat{\kappa}_2)$.*

Proof. By induction on $\kappa_1 \preccurlyeq (\hat{\gamma}_1, \hat{\kappa}_1)$. In the base case $\kappa_1 = \mathbf{0}$ and $\hat{\kappa}_1 = \mathbf{0}@\ell$, we use the hypothesis $\kappa_2 \preccurlyeq (\hat{\gamma}_2, \hat{\kappa}_2)$ to prove the thesis in the form $h(w) \cdot \kappa_2 \preccurlyeq (\hat{\gamma}_1, h(w, \mathcal{L}(\mu(\hat{\kappa}_2), \hat{\gamma}_2))@\ell \cdot \hat{\kappa}_2$. □

Lemma B.4. *If $\kappa \preccurlyeq (\hat{\gamma}, \varepsilon \cdot \hat{\kappa})$, then there exists $\hat{\gamma}^\star$ such that $\kappa \preccurlyeq (\hat{\gamma}^\star, \hat{\kappa})$ and for all $\hat{v}$ we have a transition $\langle \hat{v}, \hat{\gamma}, \varepsilon \cdot \hat{\kappa} \rangle \hookrightarrow \langle \hat{v}, \hat{\gamma}^\star, \hat{\kappa} \rangle$.*

Proof. By case analysis on $\kappa \preccurlyeq (\hat{\gamma}, \varepsilon \cdot \hat{\kappa})$. □

Now we prove the main result.

Proof of Theorem 7.1. We proceed by case analysis on $s \hookrightarrow s'$ and $s \preccurlyeq \hat{s}$, where $s = \langle M, \gamma, \kappa \rangle$ and $s' = \langle M', \gamma', \kappa' \rangle$. For each case, the SFX transition forces s and s' to have a certain shape, and similarly the source simulation forces $\hat{s}$ to have a shape that adapts to s. We will (deterministically) follow the ASFX machine on $\hat{s}$ for one or more steps, obtaining a target configuration $\hat{s}'$: then we will need to show that $s' \preccurlyeq \hat{s}'$.

$M = \oplus(V)$. In this case, $s' = \langle M', \gamma', \kappa' \rangle = \langle [\![\oplus]\!] ([\![V]\!] \gamma), \gamma, M \rangle$.
We prove $\hat{s} = \langle \oplus(\hat{V})@\ell, \hat{\gamma}, \hat{\kappa} \rangle$ where $|\hat{V}| = V, \gamma \preccurlyeq \hat{\gamma}$, and $\kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa})$. We choose $\hat{s}' = \langle [\![\oplus]\!] (|[\![\hat{V}]\!] \hat{\gamma}|), \hat{\gamma}, \hat{\kappa} \rangle$ and we prove $[\![\oplus]\!] (|[\![\hat{V}]\!] \hat{\gamma}|) = [\![\oplus]\!] ([\![V]\!] \gamma)$, to obtain $s' \preccurlyeq \hat{s}'$.

$M = \mathbf{if}(V, M_1, M_2)$. This runs differently depending on whether $[\![V]\!] \gamma = \mathbf{true}$ or $\mathbf{false}$. First, in both cases, we prove $\hat{s} = \langle \mathbf{if}(\hat{V}, \hat{M}_1, \hat{M}_2)@\ell, \hat{\gamma}, \hat{\kappa} \rangle$ where $|\hat{V}| = V, |\hat{M}_1| = M_1, |\hat{M}_2| = M_2, \gamma \preccurlyeq \hat{\gamma}$, and $\kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa})$.
Then, if $[\![V]\!] \gamma = \mathbf{true}$, we have $s' = \langle M', \gamma', \kappa' \rangle = \langle M_1, \gamma, \kappa \rangle$. We choose $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = \langle \hat{M}_1, \hat{\gamma}, \mathbf{jump}@\nu(\hat{M}_1) \cdot \hat{\kappa} \rangle$ and we prove that $\kappa \preccurlyeq (\hat{\gamma}', \hat{\kappa}')$ by applying the appropriate rule in Figure 4, to obtain $s' \preccurlyeq \hat{s}'$.
If instead $[\![V]\!] \gamma = \mathbf{false}$, we have $s' = \langle M', \gamma', \kappa' \rangle = \langle M_2, \gamma, \kappa \rangle$. We choose $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = \langle \hat{M}_2, \hat{\gamma}, \hat{\kappa} \rangle$ and we obtain $s' \preccurlyeq \hat{s}'$ trivially.

$M = \mathbf{let}\ x \leftarrow M_1\ \mathbf{in}\ M_2$. In this case, $s' = \langle M', \gamma', \kappa' \rangle = \langle M_1, \gamma, (x).M_2[\gamma] \cdot \kappa \rangle$.
We prove $\hat{s} = \langle (\mathbf{let}\ x \leftarrow \hat{M}_1\ \mathbf{in}\ \hat{M}_2)@\ell, \hat{\gamma}, \hat{\kappa} \rangle$ where $|\hat{M}_1| = M_1$, $|\hat{M}_2| = M_2$, $\gamma \preccurlyeq \hat{\gamma}$, and $\kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa})$. We choose $\hat{s}' = \langle \hat{M}_1, \hat{\gamma}, ((x).\hat{M}_2)@\nu(\hat{M}_1) \cdot \hat{\kappa} \rangle$ and we prove $|\hat{M}_2| = M_2$ and $(x).M_2[\gamma] \cdot \kappa \preccurlyeq (\hat{\gamma}, ((x).\hat{M}_2)@\nu(\hat{M}_1) \cdot \hat{\kappa}$ (the latter uses Lemma B.1). This immediately implies $s' \preccurlyeq \hat{s}'$.

$M = \mathbf{unpack}\ \overline{x_n} \leftarrow \langle \overline{V_n} \rangle\ \mathbf{in}\ M_0$. In this case, $s' = \langle M', \gamma', \kappa' \rangle = \langle M_0, \overline{(x_n \mapsto [\![V_n]\!] \gamma)} \cdot \gamma, \kappa \rangle$.
We prove $\hat{s} = \langle \mathbf{unpack}\ \overline{x_n} \leftarrow \langle \overline{\hat{V}_n} \rangle\ \mathbf{in}\ \hat{M}_0@\ell, \hat{\gamma}, \hat{\kappa} \rangle$ where $|\overline{\hat{V}_n}| = \overline{V_n}$, $|\hat{M}_0| = M_0$, $\gamma \preccurlyeq \hat{\gamma}$, and $\kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa})$. We choose $\hat{s}' = \langle \hat{M}_0, \overline{(x_n \mapsto [\![V_n]\!] \hat{\gamma})} \cdot \hat{\gamma}, \hat{\kappa} \rangle$ and we prove $|\hat{M}_0| = M_0$ and $\kappa \preccurlyeq (\hat{\gamma}', \hat{\kappa})$ (the latter uses Lemma B.1, knowing that $\hat{\gamma} \subseteq \hat{\gamma}'$). This immediately implies $s' \preccurlyeq \hat{s}'$.

$M = \textbf{handle}\ M_0\ \textbf{with}\ h(V)$. In this case, $s' = \langle M', \gamma', \kappa' \rangle = \langle M_0, \gamma, h(\llbracket V \rrbracket\ \gamma) \cdot \kappa \rangle$.

We prove $\hat{s} = \langle \textbf{handle}\ \hat{M}_0\ \textbf{with}\ h(\hat{V})@\ell, \hat{\gamma}, \hat{\kappa} \rangle$ where $\left|\hat{V}\right| = V$, $\left|\hat{M}_0\right| = M_0$, $\gamma \preccurlyeq \hat{\gamma}$, and $\kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa})$. We choose $\hat{s}' = \langle \hat{M}_0, \hat{\gamma} \setminus \mathsf{LEAVE}, h(\llbracket \hat{V} \rrbracket\ \hat{\gamma}, \mathcal{L}(\mu(\hat{\kappa}), \hat{\gamma}))@\nu(\hat{M}_0) \cdot \hat{\kappa} \rangle$ and we prove $\gamma \preccurlyeq \hat{\gamma} \setminus \mathsf{LEAVE}$ (by Lemma B.2), $\kappa \preccurlyeq (\hat{\gamma} \setminus \mathsf{LEAVE}, h(\llbracket \hat{V} \rrbracket\ \hat{\gamma}, \mathcal{L}(\mu(\hat{\kappa}), \hat{\gamma}))@\nu(\hat{M}_0) \cdot \hat{\kappa})$ (by the appropriate rule in Figure 4). This immediately implies $s' \preccurlyeq \hat{s}'$.

$M = \textbf{do}\ \#t(V)$. In this case, we have:

$$\kappa = \kappa_1^{-t} \cdot h^{+t}(w) \cdot \kappa_2$$
$$s' = \langle M', \gamma', \kappa' \rangle = \langle M^h_{\#t}, (res(h_{\#t}) \mapsto (\kappa_1^{-t}, h(w))) \cdot (dom(h) \mapsto w) \cdot (dom(h_{\#t}) \mapsto \llbracket V \rrbracket\ \gamma), \kappa_2 \rangle$$

We prove $\hat{s} = \langle (\textbf{do}\ \#t(\hat{V}))@\ell, \hat{\gamma}, \hat{\kappa}_1^{-t} \cdot h^{+t}(\hat{w}, \mathcal{L}(\ell_2, \hat{\gamma}_2))@\ell_h \cdot \hat{\kappa}' \rangle$ where $\left|\hat{V}\right| = V$, $\kappa_1^{-t} \preccurlyeq (\hat{\gamma}, \hat{\kappa}_1^{-t})$, $\kappa_2 \preccurlyeq (\hat{\gamma}_2, \hat{\kappa}_2)$. We choose $\hat{\gamma}' = (\mathsf{LEAVE} \mapsto \mathcal{L}(\ell_2, \hat{\gamma}_2)) \cdot (res(h_{\#t}) \mapsto (\hat{\kappa}_1^{-t}, h(\hat{w})@\ell_h, \hat{\gamma})) \cdot (dom(h) \mapsto \hat{w}) \cdot (dom(h_{\#t}) \mapsto \llbracket \hat{V} \rrbracket\ \hat{\gamma})$, $\hat{s}' = \langle \hat{M}^h_{\#t}, \hat{\gamma}', \textbf{exit}@\nu(\hat{M}^h_{\#t}) \cdot \hat{\kappa}_2 \rangle$ and we prove $\left|\hat{M}^h_{\#t}\right| = M^h_{\#t}$ (by the definition of the compilation procedure), $\gamma' \preccurlyeq \hat{\gamma}'$, and $\gamma' \preccurlyeq (\hat{\gamma}', \textbf{exit}@\nu(\hat{M}^h_{\#t}) \cdot \hat{\kappa}_2)$ (by applying the appropriate rule in Figure 4). This immediately implies $s' \preccurlyeq \hat{s}'$.

$M = \textbf{resume}\ U(V)$ *and* $\llbracket U \rrbracket\ \gamma = (\kappa_U, h(w))$. In this case, $s' = \langle M', \gamma', \kappa' \rangle = \langle \llbracket V \rrbracket\ \gamma, \gamma, \kappa_U \cdot h(w) \cdot \kappa \rangle$.

We prove $\hat{s} = \langle \textbf{resume}\ \hat{U}(\hat{V})@\ell, \hat{\gamma}, \hat{\kappa} \rangle$ where $\left|\hat{U}\right| = U$, $\left|\hat{V}\right| = V$ $\gamma \preccurlyeq \hat{\gamma}$, and $\kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa})$. Note that in order for $\llbracket U \rrbracket\ \gamma$ to evaluate to a continuation literal, it must be that $U = x_U$, thus $\llbracket U \rrbracket\ \gamma = \gamma(x_U)$. Then $\hat{U} = x_U$ and $\llbracket \hat{U} \rrbracket\ \hat{\gamma} = \hat{\gamma}(x_U)$ and, by applying the hypotheses, we prove $\hat{\gamma}(x_U) = (\hat{\kappa}_U, h(\hat{w})@\ell_U, \hat{\gamma}_U)$ such that $\kappa_U \preccurlyeq (\hat{\gamma}_U, \hat{\kappa}_U)$ and $w \preccurlyeq \hat{w}$.

Choose $\hat{s}' = \langle \llbracket \hat{V} \rrbracket\ \hat{\gamma}, \hat{\gamma}_U, \hat{\kappa}_U \cdot h(\hat{w}, \mathcal{L}(\mu(\hat{\kappa}), \hat{\gamma}))@\ell_U \cdot \hat{\kappa} \rangle$. By Lemma B.3 we prove $\kappa_U \cdot h(w) \cdot \kappa \preccurlyeq (\hat{\gamma}_U, \hat{\kappa}_U \cdot h(\hat{w}, \mathcal{L}(\mu(\hat{\kappa}), \hat{\gamma}))@\ell_U \cdot \hat{\kappa})$.

Furthermore, we note $\llbracket V \rrbracket\ \gamma \preccurlyeq \llbracket \hat{V} \rrbracket\ \hat{\gamma}$. This is all we need to prove $s' \preccurlyeq \hat{s}'$.

$M = U\ \textbf{with}\ h(W)$ *and* $\llbracket U \rrbracket\ \gamma = (\kappa_U, \_)$. . In this case, $s' = \langle M', \gamma', \kappa' \rangle = \langle (\kappa_U, h(\llbracket W \rrbracket\ \gamma)), \gamma, \kappa \rangle$.

We prove $\hat{s} = \langle (\hat{U}\ \textbf{with}\ h(\hat{W}))@\ell, \hat{\gamma}, \hat{\kappa} \rangle$ where $\left|\hat{U}\right| = U$, $\left|\hat{W}\right| = W$, $\gamma \preccurlyeq \hat{\gamma}$, and $\kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa})$. Note that in order for $\llbracket U \rrbracket\ \gamma$ to evaluate to a continuation literal, it must be that $U = x_U$, thus $\llbracket U \rrbracket\ \gamma = \gamma(x_U)$. Then $\hat{U} = x_U$ and $\llbracket \hat{U} \rrbracket\ \hat{\gamma} = \hat{\gamma}(x_U)$ and, by applying the hypotheses, we prove $\hat{\gamma}(x_U) = (\hat{\kappa}_U, h(\hat{w})@\ell_U, \hat{\gamma}_U)$ such that $\kappa_U \preccurlyeq (\hat{\gamma}_U, \hat{\kappa}_U)$ and $w \preccurlyeq \hat{w}$.

Choose $\hat{s}' = \langle (\hat{\kappa}_U, h(\hat{w}), \hat{\gamma}_U), \hat{\gamma}, \hat{\kappa} \rangle$. Clearly, $\hat{s} \hookrightarrow \hat{s}'$; then prove $s' \preccurlyeq \hat{s}'$ by unfolding the definition.

$M = \textbf{newbroom}(f)\ \textbf{with}\ h(V)$ *where* $dom(f_{\textbf{return}}) = x$. In this case, $s' = \langle M', \gamma', \kappa' \rangle = \langle ((((x).f_{\textbf{return}}[]), h(\llbracket V \rrbracket\ \gamma)), \gamma, \kappa \rangle$.

We prove $\hat{s} = \langle \textbf{newbroom@}\ell(f)\ \textbf{with}\ h(\hat{V}), \hat{\gamma}, \hat{\kappa} \rangle$, where $\left|\hat{V}\right| = V$; then we choose $\hat{s}' = \langle (\hat{\kappa}_f, h(\llbracket \hat{V} \rrbracket\ \hat{\gamma})@(\ell + 2), []), \hat{\gamma}', \hat{\kappa}' \rangle$ where

$$\hat{\kappa}_f = \textbf{leave@}\ell \cdot \textbf{jump@}(\ell + 1) \cdot ((x).f_{\textbf{return}})@\mu(f_{\textbf{return}}) \cdot \textbf{exit@}\nu(f_{\textbf{return}})$$

We prove that $((x).f_{\textbf{return}}[]) \cdot h(\llbracket V \rrbracket\ \gamma) \preccurlyeq [], \hat{\kappa}_f \cdot h(\llbracket \hat{V} \rrbracket\ \hat{\gamma})@(\ell + 2)$ by repeated applications of the rules defining $\preccurlyeq$.

$M = \textbf{return}\ V$. In this case, $s' = \langle M', \gamma', \kappa' \rangle = \langle \llbracket V \rrbracket\ \gamma, \gamma, M \rangle$.

We prove $\hat{s} = \langle \textbf{return}\ \hat{V}, \hat{\gamma}, \hat{\kappa} \rangle$ where $\left|\hat{V}\right| = V$, $\gamma \preccurlyeq \hat{\gamma}$, and $\kappa \preccurlyeq (\hat{\gamma}, \hat{\kappa})$. We choose $\hat{s}' = \langle \llbracket \hat{V} \rrbracket\ \hat{\gamma}, \hat{\gamma}, \hat{\kappa} \rangle$ and we obtain $s' \preccurlyeq \hat{s}'$.

$M = v$ *and* $\kappa = (x).M_0[\gamma_0] \cdot \kappa_0$. In this case, $s' = \langle M', \gamma', \kappa' \rangle = \langle M_0, (x \mapsto v) \cdot \gamma_0, \kappa_0 \rangle$.

We prove $\hat{s} = \langle \hat{v}, \hat{\gamma}, \hat{\kappa}_{epi} \cdot ((x).\hat{M}_0)@\ell \cdot \hat{\kappa}_0 \rangle$ such that $v \preccurlyeq \hat{v}, (x).M_0[\gamma_0] \cdot \kappa_0 \preccurlyeq (\hat{\gamma}, \hat{\kappa}_{epi} \cdot ((x).\hat{M}_0)@\ell \cdot \hat{\kappa}_0)$ where $\hat{\kappa}_{epi}$ consists of epilogue frames only.

By multiple applications of Lemma B.4, we prove that the well-scopedness condition on the continuations implies that there exists an environment $\hat{\gamma}_0$ such that $(x).M_0[\gamma_0] \cdot \kappa_0 \preccurlyeq (\hat{\gamma}_0, ((x).\hat{M}_0)@\ell \cdot \hat{\kappa}_0)$

and $\hat{s} \overset{*}{\hookrightarrow} \langle \hat{v}, \hat{\gamma}_0, ((x).\hat{M}_0)@\ell \cdot \hat{\kappa}_0 \rangle$: by taking one ASFX step more, we find ourselves in configuration $\langle \hat{M}_0, (x \mapsto \hat{v}) \cdot \hat{\gamma}_0, \hat{\kappa}_0 \rangle$, which we take be our $\hat{s}'$. By case analysis on the well-scopedness condition for $((x).\hat{M}_0)@\ell \cdot \hat{\kappa}_0$, we obtain $\gamma_0 \preccurlyeq \hat{\gamma}_0$, $\left|\hat{M}_0\right| = M_0$, and $\kappa_0 \preccurlyeq ((x \mapsto \hat{v}) \cdot \gamma_0, \hat{\kappa}_0)$. This is enough to prove that $s' \preccurlyeq \hat{s}'$.

$M = v$ *and* $\kappa = h(w) \cdot \kappa_0$. In this case, $s' = \langle M', \gamma', \kappa' \rangle = \langle M'^h_{\mathbf{return}}, \gamma', \kappa_0 \rangle$, where $\gamma' = (dom(h_{\mathbf{return}}) \mapsto v) \cdot (dom(h) \mapsto w)$.

We prove $\hat{s} = \langle \hat{v}, \hat{\gamma}, \hat{\kappa}_{epi} \cdot h(w, \mathcal{R}_{\mathcal{L}})@\ell \cdot \hat{\kappa}_0 \rangle$ such that $\mathcal{R}_{\mathcal{L}} = \mathcal{L}(\mu(\hat{\kappa}_0), \hat{\gamma}_0)$, $h(w) \cdot \kappa_0 \preccurlyeq (\hat{\gamma}, \hat{\kappa}_{epi} \cdot h(w, \mathcal{R}_{\mathcal{L}})@\ell \cdot \hat{\kappa}_0)$ where $\hat{\kappa}_{epi}$ consists of epilogue frames only, and $v \preccurlyeq \hat{v}$.

By multiple applications of Lemma B.4, we prove that the well-scopedness condition on the continuations implies that there exists an environment $\hat{\gamma}^*$ such that $h(w) \cdot \hat{\kappa}_0 \preccurlyeq (\hat{\gamma}^*, h(w, \mathcal{R}_{\mathcal{L}})@\ell \cdot \hat{\kappa}_0)$ and $\hat{s} \overset{*}{\hookrightarrow} \langle \hat{v}, \hat{\gamma}^*, ((x).\hat{M}_0)@\ell \cdot \hat{\kappa}_0 \rangle$: by taking one ASFX step more, we find ourselves in configuration $\langle \hat{M}^h_{\mathbf{return}}, \hat{\gamma}', \hat{\kappa}_0 \rangle$, where $\hat{\gamma}' = (\mathsf{LEAVE} \mapsto \mathcal{R}_{\mathcal{L}}) \cdot (dom(h_{\mathbf{return}}) \mapsto \hat{v}) \cdot (dom(h) \mapsto \hat{w})$: we take this last ASFX configuration to be $\hat{s}'$. By case analysis on the well-scopedness condition for $h(w, \mathcal{R}_{\mathcal{L}})@\ell \cdot \hat{\kappa}$, we obtain $\kappa_0 \preccurlyeq (\hat{\gamma}_0, \hat{\kappa}_0)$; we also prove $\gamma' \preccurlyeq \hat{\gamma}'$ by unfolding the definitions, and $\left|\hat{M}^h_{\mathbf{return}}\right| = M^h_{\mathbf{return}}$ by the definition of compilation. This is enough to prove that $s' \preccurlyeq \hat{s}'$. □

B.4 Control flow simulation

To complete the proof of correctness of the compiler, we now need to establish a simulation relation between ASFX and AsmFX. We consider AsmFX configurations $a = \langle \Xi, \Theta, C \rangle$, whose elements are memory stores Ξ (containing the program), register files Θ (finite maps from registers to values), and effect contexts C. Since a property of our compiler is that the memory store for a given source program is established at compilation time and stays read-only during execution, we will allow ourselves to omit it from AsmFX configurations and write $\langle \Theta, C \rangle$ for $\langle \Xi, \Theta, C \rangle$.

A source configuration $\hat{s}$ for a program $\hat{P}$ should correspond to an AsmFX configuration a on $\lfloor |P| \rfloor$: actually, for every $\hat{s}$ there are multiple such a, because of the plethora of registers available in the architecture: some of the registers will not have a counterpart in the source configuration and will thus be irrelevant. Therefore, the translation to AsmFX of a source configuration $\hat{s}$, which we denote by $\lfloor \hat{s} \rfloor$, will be the *set* of AsmFX configurations that are a model for $\hat{s}$ up to relevant registers. If we write $a \vDash \hat{s}$ to mean that a models $\hat{s}$, then we can define $\lfloor \hat{s} \rfloor := \{a : a \vDash \hat{s}\}$. We define what it means to be a model separately for the register file and effect context components: the register file must be a model for the whole configuration; on the other hand, the effect context only needs to model the continuation.

$$\langle \Theta, C \rangle \vDash \langle \hat{M}, \hat{\gamma}, \hat{\kappa} \rangle := (\Theta \vDash \langle \hat{M}, \hat{\gamma}, \hat{\kappa} \rangle) \wedge (C \vDash \hat{\kappa})$$

We can define what it means for an effect context to model a continuation by means of a direct translation that only needs to consider the handler frames:

$$C \vDash \hat{\kappa} := C = \lfloor \hat{\kappa} \rfloor$$

$$\lfloor \hat{\kappa} \rfloor := \begin{cases} \lfloor h(w, \mathcal{R}_{\mathcal{L}}))@\ell \rfloor \cdot \lfloor \hat{\kappa}' \rfloor & \text{if } \hat{\kappa} = h(w, \mathcal{R}_{\mathcal{L}})@\ell \cdot \hat{\kappa}' \\ \lfloor \hat{\kappa}' \rfloor & \text{if } \hat{\kappa} = \hat{\varsigma} \cdot \hat{\kappa}' \text{ and } \hat{\varsigma} \neq h(\_) \end{cases}$$

$$\lfloor h(w, \mathcal{R}_{\mathcal{L}})@\ell \rfloor := \mathcal{H}(\lfloor h \rfloor, (\overline{\$\mathsf{s} = \lfloor w \rfloor}), \lfloor \mathcal{R}_{\mathcal{L}} \rfloor)$$

$$\lfloor \mathcal{L}(\ell, \hat{\gamma}) \rfloor := \mathcal{L}(\ell, \lfloor \hat{\gamma} \rfloor)$$

$$\left\lfloor \overline{(x \mapsto v)} \right\rfloor := \overline{(\$\mathsf{x} = \lfloor v \rfloor)}$$

A register file models a source configuration if every variable defined in the source environment is backed by (any number of) registers containing the AsmFX representation of that variable's value; if the program counter matches the address of the beginning of the computation being evaluated or, when the computation has ended and we are returning a value, when it matches the address of the beginning of the continuation. Note that the pseudovariable LEAVE is always mapped to the register `$lr`, and every variable x is mapped to as many $\overline{\texttt{\$x}}$ registers as it is necessary.

$$\Theta \vDash \langle \hat{M}, \hat{\gamma}, \hat{\kappa} \rangle := \Theta(\texttt{\$pc}) = \mu(\hat{M}) \wedge \forall x \in dom(\hat{\gamma}).\overline{\Theta(\texttt{\$x})} = \lfloor \hat{\gamma}(x) \rfloor$$
$$\Theta \vDash \langle v, \hat{\gamma}, \hat{\kappa} \rangle := \Theta(\texttt{\$pc}) = \mu(\hat{\kappa}) \wedge \forall x \in dom(\hat{\gamma}).\overline{\Theta(\texttt{\$x})} = \lfloor \hat{\gamma}(x) \rfloor$$

The soundness property we seek states that the modelling relation is a simulation: whenever we have a source transition $\hat{s} \hookrightarrow \hat{s}'$ and an AsmFX configuration $a \vDash \hat{s}$, then there exists a transition chain $a \overset{*}{\hookrightarrow} a'$ where $a' \vDash \hat{s}'$. However, this property cannot be proved for an arbitrary $\hat{s}$: augmented source configurations contain metadata that encodes part of the execution history; while the compiler produces correct metadata, and the abstract machine ensures that the metadata is propagated correctly, arbitrary configurations could be unsound and not allow the simulation we need for the soundness theorem. We thus need to restrict ourselves to *valid* configurations (denoted by $\hat{s}\checkmark$) and prove, as part of our theorem, that validity is preserved by transitions.

The validity predicate is defined in Figure 5: its role is to check that all the ASFX annotations are consistent with the compiled AsmFX program. A configuration is valid if its environment $\hat{\gamma}$ and continuation $\hat{\kappa}$ are valid and additionally the environment and the continuation are coherent (notation: $\hat{\gamma} \supset \hat{\kappa}$). Furthermore, we require that in a valid configuration whose control is a computation $\hat{M}$, the AsmFX code at location $\mu(\hat{M})$ corresponds to the compiled code for $\hat{M}$, and that if instead the control is a value v, the value itself is valid.

A value is valid if any broom $\hat{k}$ syntactically contained in it is valid; an environment is valid if all the values it assigns to its domain are valid. A broom is valid if, when we resume in any valid environment γ' and continuation κ' such that the two are coherent, we get a valid continuation and, furthermore, the environment in the broom is coherent with the extended continuation.

What it means for a continuation to be valid is expressed by an inductive predicate, with a base rule for the identity continuation and a recursive rule for each type of frame. These rules ensure that the annotation for each frame corresponds to suitable AsmFX code: the annotation for **jump** frames must refer to a jump linking to the start of the rest of the continuation; in the case of **leave**, the annotation points to code loading a new leave record; in the case of **exit**, the annotation must refer to an `exit` instruction; the annotation of pure frames corresponds to an intermediate instruction of the compilation rule for sequencing; the annotation for handler frames refers to the `return` instruction that will invoke that handler's return clause.

B.4.1 Proof of Theorem 7.2. Let P be a source program, and $\Xi = \lfloor P \rfloor$. For all source configurations $\hat{s}, \hat{s}'$ such that $\hat{s}\checkmark$ (valid with respect to Ξ), if $\hat{s} \hookrightarrow \hat{s}'$, then $\hat{s}'\checkmark$ and for all AsmFX configurations a such that $a \vDash \hat{s}$ there exists a' such that $a \overset{*}{\hookrightarrow} a'$ and $a' \vDash \hat{s}'$.

To prove the theorem, we will need a few basic results on validity.

Lemma B.5. *If $\hat{\gamma}\checkmark$, then $[\![\hat{V}]\!]\ \hat{\gamma}\checkmark$.*

Proof. Routine induction on $\hat{V}$. □

Lemma B.6. *Let $\hat{\kappa} = \hat{\kappa}_1 \cdot h(\hat{w}, \mathcal{R}_{\mathcal{L}})@\ell \cdot \hat{\kappa}_0$. Suppose $\hat{\kappa}\checkmark$ and $\hat{\gamma} \supset \hat{\kappa}$. Then we have:*

- $\hat{w}\checkmark$
- $\hat{\kappa}_0\checkmark$
- $(\hat{\kappa}_1, h(\hat{w})@\ell, \hat{\gamma})\checkmark$

- $\mathcal{R}_{\mathcal{L}} = \mathcal{L}(\mu(\hat{\kappa}_0), \hat{\gamma}_0)$ *such that* $\hat{\gamma}_0 \supset \hat{\kappa}_0$

Proof. By induction on $\hat{\kappa}_1$ with a case analysis on $\hat{\kappa}\checkmark$. □

Lemma B.7. *If* $(\hat{\kappa}, h(\hat{w})@\ell, \hat{\gamma})\checkmark$ *then* $(\hat{\kappa}, h'(\hat{w}')@\ell, \hat{\gamma})\checkmark$.

Proof. The definition of $\hat{k}\checkmark$ is based on coherence, which uses operations *laddr* and *lcont*: the values of these operations are the same for both continuations (the proof is by induction on $\hat{\kappa}$). □

Lemma B.8. *Let* $\hat{k} = (\hat{\kappa}, h(\hat{w})@\ell, \hat{\gamma})$. *If* $\hat{k}\checkmark, \hat{\kappa}_0\checkmark$ *and* $\hat{\gamma}_0 \supset \hat{\kappa}_0$, *then* $(\hat{\kappa} \cdot h(w, \mathcal{L}(\mu(\hat{\kappa}_0), \hat{\gamma}_0))@\ell \cdot \hat{\kappa}_0)\checkmark$.

Proof. By the definition of $k\checkmark$. □

Proof of Theorem 7.2. The proof proceeds by cases on the possible reductions $\langle \hat{M}, \hat{\gamma}, \hat{\kappa}\rangle \hookrightarrow \langle \hat{M}', \hat{\gamma}', \hat{\kappa}'\rangle$. First we show that the transition preserves validity, and then we are able to show that all AsmFX models of the source configuration deterministically transition (in multiple steps) to a model of the target configuration.

Case $\hat{M} = \oplus(\hat{V})$. In this case $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}'\rangle = \langle [\![\oplus]\!]\,([\![\hat{V}]\!]\,\hat{\gamma}), \hat{\gamma}, \hat{\kappa}\rangle$. Since the environment and the continuation do not change, we need to check that the result of the evaluation is valid: this is trivial because we have assumed that $[\![\oplus]\!]$ only outputs valid values.

If $a \vDash \hat{s}$, we know that the AsmFX machine is about to execute $\lfloor \oplus(\hat{V}) \rfloor$. By direct inspection of the code, we see that execution until $\nu(\hat{M})$ leads us to a state a' where the $\overline{\$\mathsf{a}}$ registers have been loaded with the value $[\![\oplus]\!]\,([\![\hat{V}]\!]\,\hat{\gamma})$ and everything else is unchanged. Then $a' \vDash \hat{s}'$ and $a \overset{*}{\hookrightarrow} a'$.

Case $\hat{M} = \mathbf{if}(\hat{V}, \hat{M}_1, \hat{M}_2)$. In this case we perform different operations depending on whether $[\![\hat{V}]\!]\,\hat{\gamma}$ is **true** or **false**.

If it is **true**, then $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}'\rangle = \langle \hat{M}_1, \hat{\gamma}, \mathbf{jump}@\nu(\hat{M}_1) \cdot \hat{\kappa}\rangle$. To prove $\hat{\kappa}'\checkmark$, we show that $\hat{\kappa}\checkmark$ and $\Xi(\nu(\hat{M}_1)) = \mathsf{j}\ \mu(\hat{\kappa})$ (by the validity of $\hat{s}$ and a direct inspection of $\lfloor \hat{M} \rfloor$).

It is then trivial to prove that $\Xi(\mu(\hat{M}_1)) = \lfloor \hat{M}_1 \rfloor$, $\nu(\hat{M}_1) = \mu(\mathbf{jump}@\nu(\hat{M}_1) \cdot \hat{\kappa})$, and $\hat{\gamma} \supset \hat{\kappa}'$, therefore $\hat{s}'\checkmark$.

If $a \vDash \hat{s}$, we can execute the AsmFX code deterministically up to $\lfloor \hat{M}_1 \rfloor$ (since the if condition evaluated to **true**, the branch-if-zero instruction bz will not branch) and take the resulting configuration as a': we can easily show that $a' \vDash \hat{s}'$, and $a \overset{*}{\hookrightarrow} a'$ holds trivially.

If instead $[\![\hat{V}]\!]\,\hat{\gamma}$ evaluates to **false**, then $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}'\rangle = \langle \hat{M}_2, \hat{\gamma}, \hat{\kappa}\rangle$. The continuation and environment do not change therefore, to prove that $\hat{s}'\checkmark$, we only need to show that $\Xi(\mu(\hat{M}_2)) = \lfloor \hat{M}_2 \rfloor$ and $\nu(\hat{M}_2) = \mu(\hat{\kappa})$: these are immediate consequences of $s\checkmark$ after unfolding the definition of $\hat{M}$.

If $a \vDash \hat{s}$, we can execute the AsmFX code deterministically up to $\lfloor \hat{M}_2 \rfloor$ (since the if condition evaluated to **false**, the branch-if-zero instruction will branch to l_{else}) and take the resulting configuration as a': we can easily show that $a' \vDash \hat{s}'$, and $a \overset{*}{\hookrightarrow} a'$ holds trivially.

Case $\hat{M} = \mathbf{let}\ x \leftarrow \hat{M}_1\ \mathbf{in}\ \hat{M}_2$. In this case $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}'\rangle = \langle \hat{M}_1, \hat{\gamma}, ((x).\hat{M}_2)@\nu(\hat{M}_1) \cdot \hat{\kappa}\rangle$. To prove $\hat{\kappa}'\checkmark$, we show that $\hat{\kappa}\checkmark$, $\Xi(\nu(\hat{M}_1)) = \overline{\$\mathsf{x}} \leftarrow \overline{\$\mathsf{a}}; \lfloor \hat{M}_2 \rfloor$ and $\nu(\hat{M}_2) = \mu(\hat{\kappa})$ (all by the validity of $\hat{s}$; the second and last property also require a direct inspection of $\lfloor \hat{M} \rfloor$).

It is then trivial to prove that $\Xi(\mu(\hat{M}_1)) = \lfloor \hat{M}_1 \rfloor$, $\nu(\hat{M}_1) = \mu(((x).\hat{M}_2)@\nu(\hat{M}_1)) \cdot \hat{\kappa}$, and $\hat{\gamma} \supset \hat{\kappa}'$, therefore $\hat{s}'\checkmark$.

If $a \vDash \hat{s}$, we can choose $a' = a$: we can easily show that $a \vDash \hat{s}'$, and $a \overset{*}{\hookrightarrow} a$ holds trivially.

Case $M = \textbf{unpack}\ \overline{x_n} \leftarrow \langle \overline{\hat{V}_n} \rangle\ \textbf{in}\ \hat{M}_0$. In this case $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = \langle \hat{M}_0, \overline{(x_n \mapsto [\![\hat{V}_n]\!]\ \hat{\gamma})} \cdot \hat{\gamma}, \hat{\kappa} \rangle$. The continuation has not changed, and it is easy to show that, since $\hat{\gamma} \supset \hat{\kappa}$, then $\hat{\gamma}' \supset \hat{\kappa}$ too (because the value of LEAVE, if any, has not changed with the transition).

It is then trivial to prove that $\Xi(\mu(\hat{M}_0)) = \lfloor \hat{M}_0 \rfloor$ and $\nu(\hat{M}_0) = \mu(\hat{\kappa})$ (by direct inspection of $\lfloor \hat{M} \rfloor$); therefore $\hat{s}'\checkmark$.

If $a \vDash \hat{s}$, we execute deterministically the code for $\lfloor \hat{M} \rfloor$ up to and excluding $\lfloor \hat{M}_0 \rfloor$, obtaining a configuration a' where the registers corresponding to variables $\overline{x_n}$ have been loaded with the subvalues from the evaluation of $\hat{V}$. We can then easily show that $a' \vDash \hat{s}'$, and $a \overset{*}{\hookrightarrow} a'$ holds trivially.

Case $\hat{M} = \textbf{handle}\ \hat{M}_0\ \textbf{with}\ h(\hat{V})$. In this case

$$\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = \langle \hat{M}_0, \hat{\gamma} \setminus \mathsf{LEAVE}, h([\![\hat{V}]\!]\ \hat{\gamma}, \mathcal{L}(\mu(\hat{\kappa}), \hat{\gamma}))@\nu(\hat{M}_0) \cdot \hat{\gamma} \rangle$$

To prove $\hat{\kappa}'\checkmark$, we show that $\hat{\kappa}\checkmark$, $\Xi(\nu(\hat{M}_0)) = \mathtt{return}$, $\hat{\gamma} \supset \hat{\kappa}$ (all by the validity of $\hat{s}$, the second property also requires a direct inspection of $\lfloor \hat{M} \rfloor$).

It is then trivial to prove that $\Xi(\mu(\hat{M}_0)) = \lfloor \hat{M}_0 \rfloor$, $\nu(\hat{M}_0) = \mu(h([\![\hat{V}]\!]\ \hat{\gamma}, \mathcal{L}(\mu(\hat{\kappa}), \hat{\gamma}))@\nu(\hat{M}_0) \cdot \hat{\kappa})$, and $\hat{\gamma} \setminus \mathsf{LEAVE} \supset \hat{\kappa}'$, therefore $\hat{s}'\checkmark$.

If $a \vDash \hat{s}$, we perform as many steps as we need to reach the recursive compilation $\lfloor \hat{M}_0 \rfloor$. It is easy to show that the resulting configuration a' is a model of $\hat{s}'$.

Case $\hat{M} = \textbf{do}\ \#t(\hat{V})$ *and* $\hat{\kappa} = \hat{\kappa}^{-t} \cdot h^{+t}(\hat{w}, \mathcal{R}_{\mathcal{L}})@\ell \cdot \hat{\kappa}_0$. In this case

$$\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = \left\langle \begin{array}{l} \hat{M}_{h_{\#t}} \\ (\mathsf{LEAVE} \mapsto \mathcal{R}_{\mathcal{L}}) \cdot (res(\#t) \mapsto (\hat{\kappa}^{-t}, h^{+t}(\hat{w})@\ell, \hat{\gamma})) \\ \quad \cdot (dom(h_{\#t}) \mapsto [\![\hat{V}]\!]\ \hat{\gamma}) \cdot (dom(h) \mapsto \hat{w}) \\ \textbf{exit}@\nu(\hat{M}_{h_{\#t}}) \cdot \hat{\kappa}_0 \end{array} \right\rangle$$

To prove $\hat{\kappa}'\checkmark$, we show that $\hat{\kappa}\checkmark$ and $\Theta(\nu(\hat{M}_{h_{\#t}})) = \mathtt{exit}$ (by the validity of $\hat{s}$ and, for the second property, by direct inspection of $\lfloor h_{\#t} \rfloor$).

It is then trivial to prove that $\Xi(\mu(\hat{M}_{h_{\#t}})) = \lfloor \hat{M}_{h_{\#t}} \rfloor$, $\nu(\hat{M}_{\#t}) = \mu(\textbf{exit}@\nu(\hat{M}_{h_{\#t}}) \cdot \hat{\kappa})$. To prove $\hat{\gamma}' \supset \hat{\kappa}'$ we show:

- $\mathcal{R}_{\mathcal{L}} = (\mu(\hat{\kappa}_0), \hat{\gamma}_0)$ such that $\hat{\gamma}_0 \supset \hat{\kappa}_0$ (by $\hat{s}\checkmark$ and Lemma B.6)
- $(\hat{\kappa}^{-t}, h^{+t}(\hat{w})@\ell, \hat{\gamma})\checkmark$ (by $\hat{s}\checkmark$ and Lemma B.6)
- $[\![\hat{V}]\!]\ \hat{\gamma}\checkmark$ (by $\hat{s}\checkmark$ and Lemma B.5)
- $\hat{w}\checkmark$ (by $\hat{s}\checkmark$ and Lemma B.6).

Then it follows that $s'\checkmark$.

If $a \vDash \hat{s}$, we perform as many steps as we need to reach the recursive compilation $\lfloor \hat{M}_{\#t} \rfloor$. It is easy to show that the resulting configuration a' is a model of $\hat{s}'$.

Case $\hat{M} = \textbf{resume}\ \hat{U}(\hat{V})$. Assume $\hat{k} := [\![\hat{U}]\!]\ \hat{\gamma} = (\hat{\kappa}_U, h_U(\hat{w})@\ell, \hat{\gamma}_U)$. Then:

$$\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = ([\![\hat{V}]\!]\ \hat{\gamma}, \hat{\gamma}_U, \hat{\kappa}_U \cdot h_U(\hat{w}, \mathcal{L}(\mu(\hat{\kappa}, \hat{\gamma})))@\ell \cdot \hat{\kappa})$$

To prove $\hat{s}'\checkmark$, we first show that $[\![\hat{V}]\!]\ \hat{\gamma}\checkmark$ (by Lemma B.5). Then we note $\hat{\gamma} \supset \hat{\kappa}$, $\hat{\kappa}\checkmark$ by the validity hypothesis on the source configuration. Furthermore, in order for $[\![\hat{U}]\!]\ \hat{\gamma}$ to evaluate to the broom $\hat{k}$, we must have $\hat{U} = x_U$ for some variable x_U: therefore $\hat{k} = \hat{\gamma}(x_U)$, from which we prove $\hat{k}\checkmark$. Using these results, we apply Lemma B.8 to prove $\hat{\gamma}_U \supset \hat{\kappa}'$ and $\hat{\kappa}'\checkmark$.

If $a \vDash \hat{s}$, we execute deterministically the entire compiled code for **resume** $\hat{U}(\hat{V})$: it is easy to show that the resulting AsmFX state a' is such that $a' \vDash \hat{s}'$. In particular, we can verify that just

before executing the `resume` instruction, the register file holds the following values:

$$\begin{array}{lcl} \$\mathtt{lr} & \mapsto & \mathcal{L}(\mu(\hat{\kappa}), \lfloor \hat{\gamma} \rfloor) \\ \$\mathtt{kr} & \mapsto & \mathcal{K}(\ell_k, \lfloor \hat{\gamma}_U \rfloor, \lfloor \hat{\kappa}_U \rfloor) \\ \$\mathtt{hr} & \mapsto & \mathcal{H}(\lfloor h \rfloor, (\overline{\$\mathtt{s}} = \lfloor \hat{w} \rfloor)) \end{array}$$

where $\ell_k = \mu(\hat{\kappa}_U \cdot h(\hat{w})@\ell)$. By executing the next instruction, we therefore move into a configuration a' where the register $pc holds the address ℓ_k, the register file has been updated to match the environment $\hat{\gamma}_U$, and the effect context has been extended with $\lfloor \hat{\kappa}_U \rfloor \cdot \mathcal{H}(\lfloor h \rfloor, (\overline{\$\mathtt{s}} = \lfloor \hat{w} \rfloor), \mathcal{L}(\mu(\hat{\gamma}_0), \lfloor \hat{\gamma} \rfloor)$. We can thus easily show that $a' \vDash \hat{s}'$.

Case $\hat{M} = \hat{U}$ **with** $h(\hat{W})$. Assume $\hat{k} := [\![\hat{U}]\!]\ \hat{\gamma} = (\hat{\kappa}_U, \_@\ell, \hat{\gamma}_U)$. Then:

$$\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = \langle (\hat{\kappa}_U, h([\![\hat{W}]\!]\ \hat{\gamma}), \hat{\gamma}_U), \hat{\gamma}, \hat{\kappa} \rangle$$

To prove $\hat{s}'\checkmark$, we unfold the definition and use Lemma B.8.

If $a \vDash \hat{s}$, we execute deterministically the entire code for $\lfloor \hat{M} \rfloor$ It is easy to show that the resulting configuration a' is a model of $\hat{s}'$.

Case $\hat{M} =$ **newbroom**$@\ell(f)$ **with** $h(\hat{V})$. We have $\hat{s'} = \langle (\hat{\kappa}_f, h([\![\hat{V}]\!]\ \hat{\gamma})@(\ell + 2), []), \hat{\gamma}', \hat{\kappa}' \rangle$ where

$$\hat{\kappa}_f = \mathbf{leave}@\ell \cdot \mathbf{jump}@(\ell + 1) \cdot ((x).f_{\mathbf{return}})@\mu(f_{\mathbf{return}}) \cdot \mathbf{exit}@\nu(f_{\mathbf{return}})$$

To prove $\hat{s}'\checkmark$, we unfold the definition and in particular we verify that $(\hat{\kappa}_f \cdot h([\![\hat{V}]\!]\ \hat{\gamma})@(\ell + 2)\checkmark$ by repeated applications of the rules defining augmented source validity (this requires us to verify that the locations annotating the continuation contain the expected instructions).

If $a \vDash \hat{s}$, we execute deterministically the entire code for $\lfloor \hat{M} \rfloor$ It is easy to show that the resulting configuration a' is a model of $\hat{s}'$.

Case $\hat{M} =$ **return** $\hat{V}$. In this case $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = ([\![\hat{V}]\!]\ \hat{\gamma}, \hat{\gamma}, \hat{\kappa})$. Knowing by hypothesis that $\hat{s}\checkmark$, to prove $\hat{s}'\checkmark$ we only need to show, by Lemma B.5, that $[\![\hat{V}]\!]\ \hat{\gamma}\checkmark$.

If $a \vDash \hat{s}$, we perform the code $\lfloor$**return** $\hat{V}\rfloor = \lfloor \hat{V} \rfloor$, which loads an initial segment $\overline{\$\mathtt{a}}$ of the argument registers with the evaluation of $\hat{V}$. We can then show that the resulting state a' is a model of $\hat{s}'$.

Case $\hat{M} = \hat{v}$ *and* $\hat{\kappa} =$ **jump**$@\ell \cdot \hat{\kappa}_0$. In this case $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = (\hat{v}, \hat{\gamma}, \hat{\kappa}_0)$. To prove $\hat{s}'\checkmark$ knowing $\hat{s}\checkmark$, it is enough to show that $\hat{\kappa}_0\checkmark$ by inversion on $\hat{\kappa}\checkmark$.

If $a \vDash \hat{s}$, we know the next instruction is `j` $\mu(\hat{\kappa}_0)$. By executing that instruction, we move into a configuration a' that is the same as a except for the register $pc holding the address $\mu(\hat{\kappa}_0)$: then we can easily prove $a' \vDash \hat{s}'$.

Case $\hat{M} = \hat{v}$ *and* $\hat{\kappa} =$ **exit**$@\ell \cdot \hat{\kappa}_0$. In this case $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = (\hat{v}, \hat{\gamma}(\mathsf{LEAVE}).env, \hat{\kappa}_0)$. Knowing by hypothesis that $\hat{s}\checkmark$, we show that $\hat{\kappa}_0\checkmark$ by inversion on $\hat{\kappa}\checkmark$.

To show that $\hat{\gamma}(\mathsf{LEAVE}).env \supset \hat{\kappa}_0$, we note $\hat{\gamma} \supset \hat{\kappa}$ which, by its definition, implies the thesis.

Then, knowing $\hat{v}\checkmark$, we have immediately $\hat{s}'\checkmark$.

If $a \vdash \hat{s}$, we know the next instruction is an `exit` and the current value of register `$lr` is $\mathcal{L}(\mu(\hat{\gamma}_0), \lfloor \hat{\gamma}(\mathsf{LEAVE}).env \rfloor)$. Then, by executing that instruction, we move into a configuration a' where the register $pc holds the address $\mu(\hat{\gamma}_0)$, and the register file has been updated to match the environment $\hat{\gamma}(\mathsf{LEAVE}).env$. We can thus easily show that $a' \vDash \hat{s}'$.

Case $\hat{M} = \hat{v}$ *and* $\hat{\kappa} = \mathbf{leave}@\ell \cdot \hat{\kappa}_0$. In this case $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = (\hat{v}, (\mathsf{LEAVE} \mapsto \mathcal{L}(\ell{+}2, [])) \cdot \hat{\gamma}, \hat{\kappa}_0)$. Knowing by hypothesis that $\hat{s}\checkmark$, we show that $\hat{\kappa}_0\checkmark$ by inversion on $\hat{\kappa}\checkmark$.

To show that $(\mathsf{LEAVE} \mapsto \mathcal{L}(\ell + 2, [])) \cdot \hat{\gamma} \supset \hat{\kappa}_0$, we note, by inversion on validity, that $\hat{\kappa}_0 = \mathbf{jump}@(\ell + 1) \cdot ((x).\hat{N})@\mu(\hat{N}) \cdot \mathbf{exit}@\nu(\hat{N}) \cdot h(\hat{w})@(\ell + 2) \cdot \hat{\kappa}_1$. We can thus see that $lcont(\hat{\kappa}_0) = \ell{+}2$, which is all we need.

Then, knowing $\hat{v}\checkmark$, we have immediately $\hat{s}'\checkmark$.

If $a \vdash \hat{s}$, we know the next instruction is `loadl` $\ell + 2$. Then, by executing that instruction, we move into a configuration a' where the register `$lr` holds the record $\mathcal{L}(\ell + 2, [])$ and all the other registers match the environment $\hat{\gamma}$. We can thus easily show that $a' \vDash \hat{s}'$.

Case $\hat{M} = \hat{v}$ *and* $\hat{\kappa} = ((x).\hat{M}_0)@\ell \cdot \hat{\kappa}_0$. In this case $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = (\hat{M}_0, (x \mapsto \hat{v}) \cdot \hat{\gamma}, \hat{\kappa}_0)$.

To prove $\hat{s}'\checkmark$, knowing $\hat{s}\checkmark$, we trivially show $\hat{\kappa}_0\checkmark$ (by inversion on $\hat{\kappa}\checkmark$) and then proceed to show that $\hat{\gamma} \supset \hat{\kappa}$ implies $((x \mapsto \hat{v}) \cdot \hat{\gamma}) \supset \hat{\kappa}_0$ (LEAVE is defined in one environment if and only if it is defined in the other, and if it is its value is the same; furthermore, the conditions it has to satisfy in one continuation are exactly the same as those needed in the other). Therefore, $\hat{s}'\checkmark$ holds.

If $a \vDash \hat{s}$, we know that the register `$pc` points to code that will deterministically move data from registers $\overline{\texttt{\$a}}$ (holding the value $\lfloor \hat{v} \rfloor$) into registers $\overline{\texttt{\$x}}$, then proceed with the code for $\lfloor \hat{M}_0 \rfloor$. We can then easily show that by executing the first few instructions until we reach code address $\mu(\hat{M}_0)$, we obtain a configuration a' where the registers $\overline{\texttt{\$x}}$ have been loaded with $\lfloor \hat{v} \rfloor$. Then, if we note that $\lfloor \hat{\kappa} \rfloor = \lfloor \hat{\kappa}' \rfloor$ (since pure frames are ignored when compiling a source continuation), we prove $a' \vDash \hat{s}'$.

Case $\hat{M} = \hat{v}$ *and* $\hat{\kappa} = h(\hat{w}, \mathcal{R}_{\mathcal{L}})@\ell \cdot \hat{\kappa}_0$. In this case $\hat{s}' = \langle \hat{M}', \hat{\gamma}', \hat{\kappa}' \rangle = (h_{\mathbf{return}}, (\mathsf{LEAVE} \mapsto \mathcal{R}_{\mathcal{L}}) \cdot (dom(h_{\mathbf{return}}) \mapsto \hat{v}) \cdot (dom(h) \mapsto \hat{w}), \hat{\kappa}_0)$.

Knowing $\hat{s}\checkmark$, to prove $\hat{s}'\checkmark$ we start by proving that $(\mathbf{exit}@\nu(h_{\mathbf{return}}) \cdot \hat{\kappa}_0)\checkmark$ which requires us to show that $\Xi(\nu(h_{\mathbf{return}})) = \texttt{exit}$ (trivial by inspection of $\lfloor h_{\mathbf{return}} \rfloor$)).

To prove $\hat{\gamma}' \supset \hat{\kappa}'$, it is sufficient to show that $\mathcal{R}_{\mathcal{L}} = \mathcal{L}(\mu(\hat{\kappa}_0), \hat{\gamma}_0)$ such that $\hat{\gamma}_0 \supset \hat{\kappa}_0$. This is proved by inversion on $\hat{\kappa}\checkmark$.

Noting that $\Xi(\mu(h_{\mathbf{return}})) = \lfloor h_{\mathbf{return}} \rfloor$ (by the definition of Ξ), we have $\hat{s}'\checkmark$.

If $a \vDash \hat{s}$, we know that the next instruction is a `return`, that the initial segment of argument registers $\overline{\texttt{\$a}}$ holds $\lfloor \hat{v} \rfloor$, and that the effect context is $\mathcal{H}(\lfloor h \rfloor, (\overline{\texttt{\$s}} = \lfloor \hat{w} \rfloor), \lfloor \mathcal{R}_{\mathcal{L}} \rfloor) \cdot \lfloor \hat{\kappa}_0 \rfloor$. Then by executing `return` we load the encoding of $\hat{w}$ into registers $\overline{\texttt{\$s}}$, the encoding of $\mathcal{R}_{\mathcal{L}}$ into register `$lr`, and we jump to the address $\mu h_{\mathbf{return}}$. We execute the first few instructions of $\lfloor h_{\mathbf{return}} \rfloor$ until we reach $\hat{M}_{\mathbf{return}}$ — these instructions move data between registers and ensure that the register file corresponds to $\hat{\gamma}'$. It is thus easy to show that the configuration a' we have reached is such that $a' \vDash \hat{s}'$. □