Experience Report: Stack Switching in Wasm SpecTec

YALUN LIANG, Shanghai Jiao Tong University, China
SAM LINDLEY, The University of Edinburgh, United Kingdom
ANDREAS ROSSBERG, Independent, Germany

Wasm SpecTec [4] is a domain-specific language (DSL) in which the formal semantics of Wasm can be defined, and a toolchain to generate artefacts necessary to standardise Wasm features. We implemented the stack switching extension [2, 3] in SpecTec. Here we report on our experience.

Stack Switching for Wasm. Stack switching enables a single Wasm execution to manage multiple execution stacks. Its primary use-case is to add direct support for modular compilation of advanced non-local control flow constructs, e.g., coroutines, async/await, generators, lightweight threads, and so forth [3]. The design is based on WasmFX [2], extended with a switch instruction as an optimisation to support direct switching between sibling stacks. A formal but manually written specification exists. The proposal is formalised on paper and implemented both in the Wasm reference interpreter and the Wasmtime [1] engine. It adds six new instructions and one new administrative instruction to Wasm. We added two further administrative instructions in the specification, as we used different but equivalent semantics.

The joys and woes of the SpecTec toolchain. Anecdotally, we found SpecTec to be quite usable. For much of the proposal, the implementation was a direct transliteration of the manually written specification into SpecTec. Some reduction rules were changed to use a different but equivalent formulation, as we used a *bubbling semantics* in place of *contexts* (we will return to this shortly). The meta-level type checking built into SpecTec helped a lot when debugging and refactoring. SpecTec was able to generate the correct formal LaTeX specifications with no additional effort. Generation of the prose specification, however, was not quite as straightforward: initially SpecTec was only able to generate correct pseudocode for one of the six new instructions.

Refining the SpecTec implementation. SpecTec supports a declarative relational specification of reduction rules. In the SpecTec implementation, the source language is referred to as the EL (External Language). SpecTec has two intermediate languages: IL (Internal Language), a simplified version of the declarative EL, and AL (Algorithmic Language), an algorithmic language tuned for generating prose and direct execution. EL is first elaborated into IL, IL is translated into AL, and the prose specification is generated from the AL. The IL to AL translation turned out to be somewhat fragile in our case. The translation makes various assumptions about the semantics of the language. For example, it assumes that a reduction rule with a disjunctive premise may be decomposed into two reduction rules corresponding to the two branches. This assumption appears innocent at first glance, but turns out to be problematic, as a side-effect may occur before the disjunction is evaluated. The translation also hard-codes various aspects of the semantics. For instance, it hard-codes a list of frame types, a list of instructions that are values, and the layout of the program store. It maintains a bidirectional mapping of instructions in SpecTec and instructions in the reference interpreter. It also hard-codes part of the semantics of several instructions. Adjusting the IL to AL translation was a substantial part of our work. A deep understanding of the SpecTec implementation was required to fix these assumptions and update the hard-coded semantics. We generalised some parts of the translation algorithm, removing some assumptions and hard-coded semantics. Still, some remain,

Authors' addresses: Yalun Liang, symb@olic.link, Shanghai Jiao Tong University, Shanghai, China; Sam Lindley, Sam.Lindley@ed.ac.uk, The University of Edinburgh, Edinburgh, United Kingdom; Andreas Rossberg, rossberg@mpi-sws.org, Independent, Munich, Germany.

and we hope to be able to make the translation more robust in order to ease the implementation of future extensions of Wasm.

Evaluation contexts. The rendering of the stack switching specification in SpecTec (and indeed other parts of the core Wasm specification) differs slightly from the way it is officially specified. The key difference is that the reduction rules in the official specification are given in terms of *contexts*, a concept that SpecTec does not currently support. To work around this limitation, SpecTec necessitates explicitly expanding certain reduction rules. For instance, on paper, the reduction rules for traps are defined in terms of *evaluation contexts* of the form.

$$E \quad ::= \quad [\_] \mid val^* \; E \; instr^* \mid \mathsf{label}_n\{instr^*\} \; E \; \mathsf{end}.$$

The following rule allows reduction to proceed inside an evaluation context:

$$\begin{array}{lcl} S; F; E[instr^*] & \hookrightarrow & S'; F'; E[instr'^*] \\ & & (\text{if } E \neq [\_] \;\wedge\; S; F; instr^* \hookrightarrow S'; F'; instr'^*), \end{array}$$

The following rule allows a trap to escape an evaluation context:

$$S; F; E[\mathsf{trap}] \quad \hookrightarrow \quad S; F; \mathsf{trap} \quad (\text{if } E \neq [\_])$$

In SpecTec, four rules are required to specify the same behaviour.

$$\begin{array}{rcl} S; F; val^* \; instr^* \; instr''^* & \hookrightarrow & S'; F'; val^* \; instr'^* \; instr''^* \\ & & (\text{if } S; F; instr^* \hookrightarrow S'; F'; instr'^* \wedge (val^* \neq \epsilon \vee instr^* \neq \epsilon)) \\ S; F; \mathsf{label}_n\{instr''^*\} \; instr \; \mathsf{end} & \hookrightarrow & S'; F'; \mathsf{label}_n\{instr''^*\} \; instr' \; \mathsf{end} \\ & & (\text{if } S; F; instr^* \hookrightarrow S'; F'; instr'^*) \\ S; F; val^* \; \mathsf{trap} \; instr^* & \hookrightarrow & S; F; \mathsf{trap} \quad (\text{if } val^* \neq \epsilon \vee instr^* \neq \epsilon) \\ S; F; \mathsf{label}_n\{instr^*\} \; \mathsf{trap} \; \mathsf{end} & \hookrightarrow & S; F; \mathsf{trap}. \end{array}$$

The first two rules implement the behaviour of the first rule above; the second two implement the behaviour of the second rule above.

Handler contexts and bubbling. The core operations of stack switching are *suspending* and *resuming* continuations (where one can view the switch instruction as a fusion of the two). A suspended continuation may be resumed under a given handler. The handler specifies what to do when the continuation is subsequently resumed. The stack switching reduction rules are defined using *handler contexts*, a refinement of evaluation contexts, which consists of the frames between a suspension and its handler. In order to simulate the presence of such handler contexts in SpecTec, we implement a *bubbling semantics* in which a suspension is bubbled up to its handler one frame at a time. More generally, both handler and evaluation contexts are suspended and resumed one frame at a time. To implement the bubbling semantics we introduce two additional administrative instructions: suspending and resuming. To suspend a frame, we execute the suspending instruction, which stores the current captured continuation, and bubbles up until a matching handler is reached. Resuming a frame is similar but in reverse.

Adding native contexts to SpecTec. We are currently experimenting with introducing a notion of first-class context to SpecTec. This requires a somewhat nontrivial extension to SpecTec and its implementation, as contexts are a new kind of construct: they can be used to perform a pattern match on a sequence of instructions; we can capture them as a value and save them in the store as a part of a continuation; and we can also plug the hole of a context with a sequence of instructions. Contexts would enable us to write reduction rules more concisely as in the current official specification, and to avoid the need for bubbling when specifying features such as stack switching.

REFERENCES

[1] Bytecode Alliance. wasmtime: a standalone runtime for WebAssembly, 2024. https://wasmtime.dev/.

[2] L. Phipps-Costin, A. Rossberg, A. Guha, D. Leijen, D. Hillerström, K. C. Sivaramakrishnan, M. Pretnar, and S. Lindley. Continuing WebAssembly with effect handlers. *Proc. ACM Program. Lang.*, 7(OOPSLA2):460–485, 2023.

[3] WebAssembly Community Group. WebAssembly stack switching extension, 2024. https://github.com/WebAssembly/stack-switching.

[4] D. Youn, W. Shin, J. Lee, S. Ryu, J. Breitner, P. Gardner, S. Lindley, M. Pretnar, X. Rao, C. Watt, and A. Rossberg. Bringing the WebAssembly standard up to speed with SpecTec. *Proc. ACM Program. Lang.*, 8(PLDI), 2024.