Debugging with Domain-Specific Events via Macros

Xiangqi Li
University of Utah
USA
xiangqi@cs.utah.edu

Matthew Flatt
University of Utah
USA
mflatt@cs.utah.edu

Abstract

Extensible languages enable the convenient construction of many kinds of domain-specific languages (DSLs) by mapping domain-specific surface syntax into the host language's core forms in a layered and composable way. The host language's debugger, however, reports evaluation and data details in ways that reflect the host language, instead of the DSL in its own terms, and closing the gap may require more than correlating host evaluation steps to the original DSL source. In this paper, we describe an approach to DSL construction with macros that pairs the mapping of DSL terms to host terms with a mapping to convert primitive events back to domain-specific concepts. Domain-specific events are then suitable for presenting to a user or wiring into a domain-specific visualization. We present a core model of evaluation and events, and we present a language design—analogous to pattern-based notations for macros, but in the other direction—for describing how events in a DSL's expansion are mapped to events at the DSL's level.

CCS Concepts • **Software and its engineering → Extensible languages; Macro languages; Software testing and debugging**;

Keywords Debugging, domain-specific languages, events

ACM Reference Format:
Xiangqi Li and Matthew Flatt. 2017. Debugging with Domain-Specific Events via Macros. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3136014.3136019

SLE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136019

1 Introduction

Domain-specific languages (DSLs) are designed to improve ease of use and productivity [Mernik et al. 2005; Ward 1994] by offering expressive, domain-specific notation and abstractions [van Deursen et al. 2000]. Although they are specific to a domain, many DSLs are still recognizably programming languages, and a good debugging experience is indispensable to using any programming language. With current DSL-construction tools, however, debugging support for executable DSLs remains unsatisfactory. Sometimes, the problem is that debugging operations are limited to traditional debugging techniques for imperative languages, such as setting breakpoints and stepping [Henriques et al. 2005; Van Den Brand et al. 2005; Wu et al. 2008]. Even in an event-based debugging framework, often the events are insufficiently general [Chis et al. 2014; Lindeman et al. 2011] and expose too much the constructs of the general-purpose language (GPL) that hosts the DSL implementation.

In the realm of debugging for GPLs, a programmer is usually presented with an interface to step through imperative statements and set breakpoints. A more general model, which is supported by many debuggers, views the activity of a program as a generator of events that can be inspected and to some degree controlled [Bates 1995; Marceau et al. 2006; Olsson et al. 1990]. An event-based view enables reuse and extension of a debugger, and we take this view as one of our starting points. In the same way that GPLs can host DSL implementations, an environment that provides GPL debugging events can host DSL debugging events. This approach adapts well to many different kinds of problems, including domain-specific problems where the evaluation rules might not follow a conventional imperative flow. A gap remains, however, for mapping GPL events back to a DSL in a precise, reusable, and extensible way.

We combine an event-oriented view of DSL debugging with a view of DSL construction based on macros, as they are commonly implemented in Lisp environments and especially in Racket [Felleisen et al. 2015; Tobin-Hochstadt et al. 2011]. Macros provide high-level support for converting the constructs of a DSL into lower-level constructs of a host language. Macros compose well, so that multiple DSLs can coexist in a larger application along with the host language. Macros also naturally enable towers of languages [Ward 1994], where terms in a source language are expanded to successively simpler languages, while each intermediate point

serves as a well-defined and reusable language in its own right.

An event-oriented view of debugging meshes well with a macro-expansion view of language implementation. Macros specify the runtime semantics of a DSL through elaboration into lower-level constructs. Meanwhile, debugging events from the lower-level language can be filtered, combined, and transformed to describe debugging events in terms of the DSL. In the same way that static elements of a language can be associated with macros to implement, say, a type system [Chang et al. 2017], a protocol for debugging events can be integrated with macro transformations. As a result, the programmer gets support for DSL debugging on par with support for DSL type systems and runtime evaluation. We further imagine a suite of visualization and interaction tools to allow programmers to view and interact with events to debug programs, where the visualizations and interactions can be tailored to the DSL as needed.

In this paper, we lay a foundation for evaluation and describe events that support our vision. First, we present a core programming-language model that supports debugging events, where the events are sufficient to fully reconstruct the state of an evaluation. We then describe the constructs that DSL implementers use to map events from one language level (either the core language or a derived language) to a new language level; those constructs are ultimately implemented in terms of the core language's event-reporting mechanism, but with conversion layers that aggregate and transform core events into domain-specific events. To help validate the design of our event framework, we have implemented debugging support for three DSLs, and we report our experience with domain-specific event creation and generation for those languages.

2 Motivation

To see the need for domain-specific events to implement a DSL debugger, consider the case of POP-PL [Florence et al. 2015]. POP-PL is a "patient-oriented prescription programming language" that is meant to enable a doctor to describe and automate a course of treatment. The language is message-based, where a message might correspond to adjusting a medical device or calling a nurse to take a specific action. POP-PL is implemented by macro expansion to conventional functional and imperative programming constructs. If we try to rely on the underlying GPL's stepping-based debugger and map POP-PL source terms to those debugging events, the debugger would not match a health-care professional's view of the computation.

A debugger for POP-PL should instead present debugging in terms of the events that capture domain concepts. Concretely, we have a debugging interface as shown in Figure 1. To capture the concepts of an arriving message and an outgoing message from prescription handlers, we should

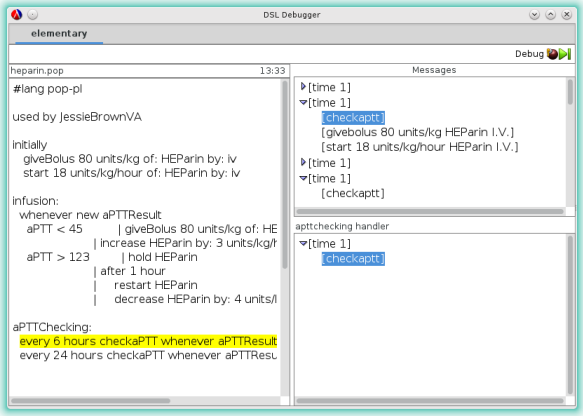


Figure 1. POP-PL debugger. The interface presents a debugging scenario where a click of a message entry in the top-right Messages window triggers a display of the handler information in the bottom-right window. Clicking a message in the bottom-right window navigates us to the source context on the left, which is highlighted in yellow.

rely on domain-specific events such as `receive-msg-e` and `send-msg-e` events.

3 Implementing DSLs with Macros

Our approach of mapping general-purpose events to domain-specific events fits together naturally with an evaluation approach that maps DSL programs into GPL programs via macros. In Racket specifically, building a DSL typically involves a *reader-level extension* to parse a DSL program into parenthesized forms (S-expressions), and then an *expander-level extension* that relies on macros to expand the parenthesized forms. We will review enough of Racket's macro and module system for the rest of the paper to make sense.

A module in Racket is both a unit of compilation and the mechanism for organizing macros and language layers. In the simplest case, a DSL program resides in its own module, while the implementation of the DSL itself resides in another module that is referenced by the DSL program. The DSL implementation is written in some language as defined by module imports, and the DSL's macros expand into the imported constructs—where the imports can be macros that expand to another language, and so on.

Figure 2 illustrates the overall process of expanding a single DSL module via macros and through multiple language layers. (The figure does not show the implementations of the layers, but only the way that the original module expands into each layer.) Box 1 shows a program written in a toy DSL called `point`. The `point` language contains initialization statements for the `x` and `y` properties and operations such as `move x by` and `move y by` to manipulate the two properties.

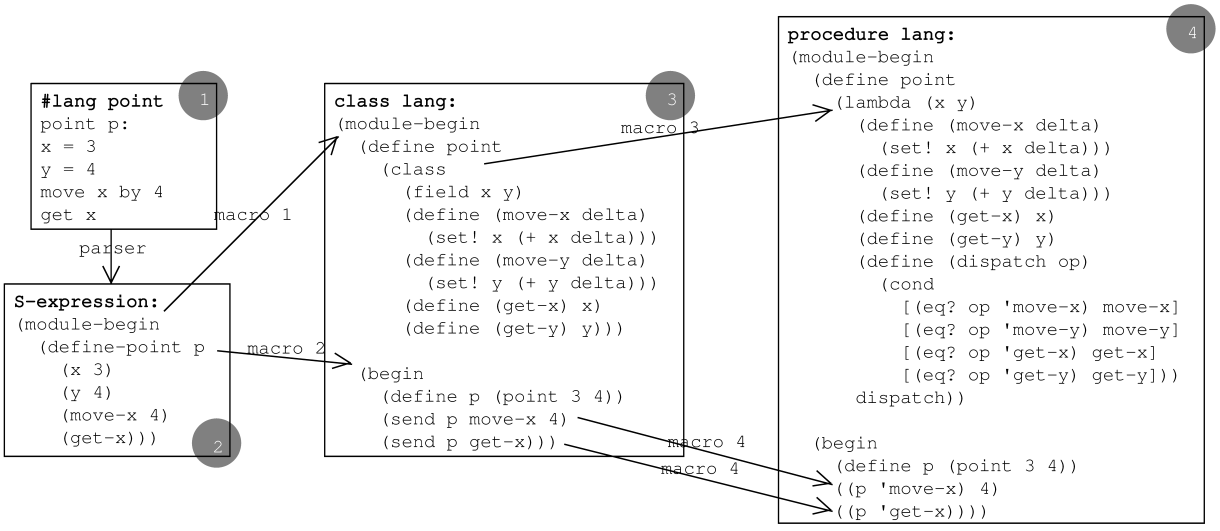


Figure 2. Macro expansion

The `#lang point` declaration in the box selects the reader that parses the program into a `define-point` S-expression, which is sketched in box 2. In addition to that S-expression, box 2 must import a `module-begin` macro (referred to as "macro 1" among the figure's arrows) and a `define-point` macro (referred to as "macro 2"). The `module-begin` macro's job is to add a definition of `point` to the beginning of the module, and the `define-point` macro's job is to make an instance of `point` and call some of its methods.

To a first approximation, each of those macros can be implemented as a simple pattern-based macro, which uses the form

```
(define-syntax-rule pattern template)
```

to indicate that each instance of *`pattern`* should be replaced by an instance of *`template`*. Pattern variables bound in *`pattern`* are replaced as *`template`* is instantiated. So, `module-begin` and `define-point` also can be defined as

```
(define-syntax-rule (module-begin decl ...)
  (base-module-begin
    ; add a point class declaration
    (define point
      (class
        details omitted, but see box 3 in the figure))
    decl ...))

(define-syntax-rule (define-point name (x x-exp) (y y-exp)
                      (op arg ...) ...)
  (begin
    (define name (point x-exp y-exp))
    (send name op arg ...) ...))
```

The pattern `(module-begin decl ...)` matches any term that starts with `module-begin` followed by any number of terms bound to the pattern variable *`decl`*. The `...` after *`decl`* causes *`decl`* to stand for zero or more matches. The expansion of the macro is a `base-module-begin` form that defines the name `point` and continues with all the supplied *`decl`*s. Similarly, the pattern for `define-point` matches that name followed by at least two terms, where *`x-exp`* stands for the second part of the first term, *`y-exp`* stands for the second part of the second term, *`op`* stands for the called method in each subsequent term, and *`arg`* stands for the arguments of each of the called methods (i.e., *`arg`* is effectively a list of lists). Note that the number of `...`s after a pattern variable in a template matches the number of `...`s after the same pattern variable in the pattern. The `base-module-begin`, `define`, `class` and `send` forms used in the macro expansion are all imported into the module that defines the macros.

Although simple pattern-based macros work for many cases, these macros are not quite right for `module-begin` and `define-point`. The `define-point` macro wants `x` and `y` to be literally the identifiers `x` and `y`, instead of pattern variables that match any term. In addition, the macros `module-begin` and `define-point` independently introduce a definition and references to `point`, so macro *hygiene* keeps them separate [Kohlbecker et al. 1986] instead of shared as intended.

To solve these problems, we can rewrite `module-begin` and `define-point` as general compile-time functions using `syntax-case` expression forms that help with pattern matching and template instantiation:

```
(define-syntax id function-expr)

(syntax-case source-expr (literal-id ...)
  [pattern optional-guard-expr template-expr]
```

```
  ...)
```

The `module-begin` definition above can be rewritten as

```
(define-syntax module-begin
  (lambda (stx)
    (syntax-case stx ()
      [(_) #'(base-module-begin)] ; no decls ⇒ no class
      [(_ decl ...)
       #'(base-module-begin
           (define point details omitted)
           decl ...)])))
```

where `define-syntax` binds `module-begin` to a compile-time function that receives a representation `stx` of the macro use. The `syntax-case` form dispatches on that `stx` to match one of the subsequent patterns; we add a new pattern here, as a kind of optimization, to drop the definition of `point` if there are no *`decls`* to use it. After matching a pattern in `syntax-case`, the corresponding clause can perform arbitrary compile-time work, but `#'` produces a compile-time value from a template instantiation, just like a simple pattern-matching macro.

From now on, we'll abbreviate a definition

```
(define-syntax id (lambda (arg-id) body-expr))
⇒
(define-syntax (id arg-id) body-expr)
```

To fix `module-begin`, the `point` identifier in the template needs to be replaced with `point` as if it appeared in the macro-use site. The expression `(syntax-local-introduce #'point)` generates such an identifier,[1] and we can inject it into the pattern world using `with-syntax`, which binds a left-hand-side pattern to the result of a right-hand-side compile-time expression:

```
(define-syntax (module-begin stx)
  (syntax-case stx ()
    [(_) #'(base-module-begin)]
    [(_ decl ...)
     (with-syntax ([point-id
                    (syntax-local-introduce #'point)])
       #'(base-module-begin
           (define point-id details omitted)
           decl ...))]))
```

The repair to `define-point` is similar, but also uses the parentheses that appear after the first argument to `syntax-case`, which hold identifiers to be treated as literals instead of pattern variables:

```
(define-syntax (define-point stx)
  (syntax-case stx (x y)
    [(_ name (x x-expr) (y y-expr) (op arg ...) ...)
     (with-syntax ([point-id
                    (syntax-local-introduce #'point)])
       #'(begin
           (define name (point-id x-expr y-expr))
           (send name op arg ...) ...))]))
```

[1] Using `syntax-local-introduce` is rarely the best strategy, but it suffices for the example here.

The expansion of the `module-begin` and `define-point` macros on the code in box 2 of Figure 2 produces the code in box 3 of the figure. Box 3 shows another `module-begin` in place of `base-module-begin` on the assumption that the form imported by the macro-implementing module as `base-module-begin` is exported from its defining module as `module-begin`.

For the language of box 3, assume that `module-begin` adds nothing to its content, and consider further the `class` and `send` macros that must be imported there. The `class` macro implements a class abstraction in terms of procedures, and the `send` macro accordingly transforms method calls into nested procedure calls.

```
(define-syntax (class stx)
  (syntax-case stx (field define)
    [(class (field f ...)
       (define (method-name arg ...) expr ...) ...)
     (andmap identifier?
             (syntax->list #'(method-name ...
                              arg ... ...)))
     #'(lambda (f ...)
         (define (method-name arg ...) expr ...) ...
         (define (dispatch op)
           (cond
             [(eq? op 'method-name)
              method-name] ...))
         dispatch)]))

(define-syntax (send stx)
  (syntax-case stx ()
    [(send obj method-name arg ...)
     (identifier? #'method-name)
     #'((obj 'method-name) arg ...)]))
```

That is, a `class` form turns into a function that accepts field values for an instance and returns a dispatch function for the instance, and a `send` form turns into a call of that dispatch function passing the method name as a symbol. These macro implementations contain additional compile-time code as guards to check that each method name and method argument is an identifier (as opposed to, say, a number) before generating the expansion, which illustrates another use of compile-time computation.

Using the `class` and `send` forms, the original DSL program is further expanded into the forms in box 4. If the `procedure` language is created through more macro transformations into some other existing language, then the expansion of the DSL program requires additional steps. The general case of a module expansion is depicted on the left-hand side of Figure 3, where a DSL is eventually compiled to a core language, and each dotted line represents the macro transformations in a compilation step.

The right-hand side of Figure 3 is the contribution of this paper. We define a set of events that are produced by the evaluation of a program in the core language, and we show how to enrich and complement the macro-expansion steps

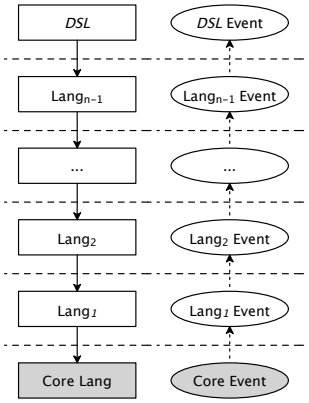


Figure 3. An overview of layered DSL implementation and domain-specific event support

on the left-hand side with event-mapping rules for the right-hand side.

4 Core Events

The core language in the bottom left of Figure 3 can be any simple programming language. For Racket, the core language is a variant of the λ-calculus with primitives and mutable variables, so we can model it with a CESK machine [Felleisen and Friedman 1987]. A CESK machine explicitly manages a lexical environment, continuation, and store, so it serves as a general model of a GPL that exposes features relevant for debugging—but it has no notion of *events*.

Figure 4 defines an extension of the CESK machine that includes a slot for an event trace, and Figure 5 shows the associated grammars. Each step in our *CESKT* machine adds one or more events to the trace component of the machine. The resulting trace models the sequence of events that a debugger can receive to report on the progress of the computation.

An *event* reports some interesting point in the dynamic execution of a program. Core events include `construct-e`, `function-e`, `variable-update-e`, `cont-add-e`, and `cont-rmv-e`. Evaluation of a program generates event instances of these event classes, and each kind of event carries event-specific debugging information. Figure 6 shows the updt-t metafunction, which defines the mechanism of event-specific information encapsulation. Each event instance is a data structure (*evnt* in Figure 8) containing its event class name, source information, an event tag indicating the event's origin in the core language, and an event attribute mapping (A).

Every reduction rule in Figure 4 starts with a `construct-e` event to reflect the occasion of a single-step reduction of an expression. The `function-e` event and the `variable-update-e` event capture the store changes in the [apply] and [assign] rules. To monitor the continuation changes of the CESK machine, we include continuation-related events: `cont-add-e` and `cont-rmv-e`.

$$\langle\langle (M\ N), \mathcal{E}\rangle, \Sigma, K, T\rangle \quad \text{[to-apply]}$$
$$\longrightarrow \langle\langle M, \mathcal{E}\rangle, \Sigma, \langle \texttt{ar}, \langle N, \mathcal{E}\rangle, K\rangle, T_{new}\rangle$$
$$\text{subject to updt-t}[\![T, (M\ N), \langle \texttt{construct-e}, M, \mathcal{E}\rangle, \langle \texttt{cont-add-e}, \langle \texttt{ar}, \langle N, \mathcal{E}\rangle\rangle\rangle]\!] = T_{new}$$

$$\langle\langle (o\ M\ N\ ...), \mathcal{E}\rangle, \Sigma, K, T\rangle \quad \text{[to-prim]}$$
$$\longrightarrow \langle\langle M, \mathcal{E}\rangle, \Sigma, \langle \texttt{op}, \langle o\rangle, \langle\langle N, \mathcal{E}\rangle, ...\rangle, K\rangle, T_{new}\rangle$$
$$\text{subject to updt-t}[\![T, (o\ M\ N\ ...), \langle \texttt{construct-e}, M, \mathcal{E}\rangle, \langle \texttt{cont-add-e}, \langle \texttt{op}, \langle o\rangle, \langle\langle N, \mathcal{E}\rangle, ...\rangle\rangle\rangle]\!] = T_{new}$$

$$\langle\langle V, \mathcal{E}\rangle, \Sigma, \langle \texttt{fn}, \langle (\lambda\ X\ M), \mathcal{E}_1\rangle, K\rangle, T\rangle \quad \text{[apply]}$$
$$\longrightarrow \langle\langle M, \mathcal{E}_1[X \leftarrow \sigma]\rangle, \Sigma[\sigma \leftarrow \langle V, \mathcal{E}\rangle], K, T_{new}\rangle$$
$$\text{subject to } V \notin X, \sigma \notin \texttt{dom}(\Sigma),$$
$$\text{updt-t}[\![T, V, \langle \texttt{construct-e}, M, \mathcal{E}_1\rangle, \langle \texttt{function-e}, X, \langle \sigma, \langle V, \mathcal{E}\rangle\rangle\rangle, \langle \texttt{cont-rmv-e}\rangle]\!] = T_{new}$$

$$\langle\langle V, \mathcal{E}\rangle, \Sigma, \langle \texttt{ar}, \langle N, \mathcal{E}_N\rangle, K\rangle, T\rangle \quad \text{[apply-arg]}$$
$$\longrightarrow \langle\langle N, \mathcal{E}_N\rangle, \Sigma, \langle \texttt{fn}, \langle V, \mathcal{E}\rangle, K\rangle, T_{new}\rangle$$
$$\text{subject to } V \notin X,$$
$$\text{updt-t}[\![T, V, \langle \texttt{construct-e}, N, \mathcal{E}_N\rangle, \langle \texttt{cont-rmv-e}\rangle, \langle \texttt{cont-add-e}, \langle \texttt{fn}, \langle V, \mathcal{E}\rangle\rangle\rangle]\!] = T_{new}$$

$$\langle\langle b_m, \mathcal{E}\rangle, \Sigma, \langle \texttt{op}, \langle\langle b_{rest}, \mathcal{E}_{rest}\rangle, ..., \langle b_1, \mathcal{E}_1\rangle, o\rangle, \langle\ \rangle, K\rangle, T\rangle \quad \text{[prim]}$$
$$\longrightarrow \langle\langle b, \varnothing\rangle, \Sigma, K, T_{new}\rangle$$
$$\text{subject to } \delta(o, b_1, b_{rest}, ..., b_m) = b,$$
$$\text{updt-t}[\![T, b_m, \langle \texttt{construct-e}, b, \varnothing\rangle, \langle \texttt{cont-rmv-e}\rangle]\!] = T_{new}$$

$$\langle\langle V, \mathcal{E}\rangle, \Sigma, \langle \texttt{op}, \langle v_R, ..., o\rangle, \langle\langle N, \mathcal{E}_N\rangle, v_L, ...\rangle, K\rangle, T\rangle \quad \text{[prim-arg]}$$
$$\longrightarrow \langle\langle N, \mathcal{E}_N\rangle, \Sigma, \langle \texttt{op}, \langle\langle V, \mathcal{E}\rangle, v_R, ..., o\rangle, \langle v_L, ...\rangle, K\rangle, T_{new}\rangle$$
$$\text{subject to } V \notin X,$$
$$\text{updt-t}[\![T, V, \langle \texttt{construct-e}, N, \mathcal{E}_N\rangle, \langle \texttt{cont-rmv-e}\rangle, \langle \texttt{cont-add-e}, \langle \texttt{op}, \langle\langle V, \mathcal{E}\rangle, v_R, ..., o\rangle, \langle v_L, ...\rangle\rangle\rangle]\!] = T_{new}$$

$$\langle\langle X, \mathcal{E}\rangle, \Sigma, K, T\rangle \longrightarrow \langle\langle V, \mathcal{E}_V\rangle, \Sigma, K, T_{new}\rangle \quad \text{[var]}$$
$$\text{subject to } \Sigma(\mathcal{E}(X)) = \langle V, \mathcal{E}_V\rangle,$$
$$\text{updt-t}[\![T, X, \langle \texttt{construct-e}, V, \mathcal{E}_V\rangle]\!] = T_{new}$$

$$\langle\langle (\texttt{set}\ X\ M), \mathcal{E}\rangle, \Sigma, K, T\rangle \quad \text{[to-assign]}$$
$$\longrightarrow \langle\langle M, \mathcal{E}\rangle, \Sigma, \langle \texttt{set}, \langle X, \mathcal{E}\rangle, K\rangle, T_{new}\rangle$$
$$\text{subject to updt-t}[\![T, (\texttt{set}\ X\ M), \langle \texttt{construct-e}, M, \mathcal{E}\rangle, \langle \texttt{cont-add-e}, \langle \texttt{set}, \langle X, \mathcal{E}\rangle\rangle\rangle]\!] = T_{new}$$

$$\langle\langle V, \mathcal{E}\rangle, \Sigma, \langle \texttt{set}, \langle X, \mathcal{E}_X\rangle, K\rangle, T\rangle \quad \text{[assign]}$$
$$\longrightarrow \langle\Sigma(\sigma), \Sigma[\sigma \leftarrow \langle V, \mathcal{E}\rangle], K, T_{new}\rangle$$
$$\text{subject to } V \notin X, \mathcal{E}_X(X) = \sigma, \Sigma(\sigma) = \langle V_1, \mathcal{E}_1\rangle,$$
$$\text{updt-t}[\![T, V, \langle \texttt{construct-e}, V_1, \mathcal{E}_X\rangle, \langle \texttt{variable-update-e}, X, \langle \sigma, \langle V, \mathcal{E}\rangle\rangle\rangle, \langle \texttt{cont-rmv-e}\rangle]\!] = T_{new}$$

Figure 4. The CESKT machine

If an implementation of the CESKT machine implements environments and program fragments with sharing, then each event logged to the trace component is a bounded increment in space consumption. Nevertheless, the machine's trace component contains enough information to reconstruct the rest of the machine state after each step. The ability to reconstruct the machine state provides evidence that the trace

$$
\begin{array}{lll}
M, N, L ::= V & & V ::= X \mid \texttt{(}\lambda\ X\ M\texttt{)} \mid b \mid bool \\
\quad \mid \texttt{(}M\ M\texttt{)} & & bool ::= \texttt{true} \mid \texttt{false} \\
\quad \mid \texttt{(}o\ M\ M\ \ldots\texttt{)} & & b ::= number \\
\quad \mid \texttt{(set}\ X\ M\texttt{)} & & v ::= \langle V, \mathcal{E} \rangle \\
o ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{add1} \mid \texttt{sub1} \mid \ldots & & \\
\mathcal{E} ::= \varnothing \mid \texttt{\{(}X\ \sigma\texttt{)}\ \ldots\texttt{\}} & & T ::= \texttt{mt} \mid \langle evnt, T \rangle \\
B ::= \varnothing \mid \{\langle \sigma, v \rangle\ \ldots\} & & \sigma ::= \text{a location} \\
\Sigma ::= \varnothing \mid \{\langle \sigma, v \rangle\ \ldots\} & & store\text{-}e ::= \texttt{function-e} \\
A ::= \varnothing \mid \texttt{\{(}name\ attr\text{-}val\texttt{)}\ \ldots\texttt{\}} & & \quad \mid \texttt{variable-update-e} \\
 & & X ::= variable\text{-}not\text{-}otherwise\text{-}mentioned
\end{array}
$$

Figure 5. Grammars for the extended CESK machine

$$
\begin{array}{l}
\text{updt-t}[\![T, N, \langle \texttt{construct-e}, M, \mathcal{E} \rangle]\!] = \langle evnt, T \rangle \\
\quad \text{subject to } \texttt{\{(expression}\ M\texttt{)\ (bindings}\ \mathcal{E}\texttt{)\}} = A, \\
\qquad \langle \texttt{construct-e}, N, \texttt{core}, A \rangle = evnt \\
\text{updt-t}[\![T, N, \langle store\text{-}e, X, \langle \sigma, \langle V, \mathcal{E} \rangle \rangle \rangle]\!] = \langle evnt, T \rangle \\
\quad \text{subject to } \texttt{\{(name}\ X\texttt{)\ (slot}\ \sigma\texttt{)\ (value}\ \langle V, \mathcal{E} \rangle\texttt{)\}} = A, \\
\qquad \langle store\text{-}e, N, \texttt{core}, A \rangle = evnt \\
\text{updt-t}[\![T, N, \langle \texttt{cont-add-e}, cont \rangle]\!] = \langle evnt, T \rangle \\
\quad \text{subject to } \texttt{\{(continuation}\ cont\texttt{)\}} = A, \\
\qquad \langle \texttt{cont-add-e}, N, \texttt{core}, A \rangle = evnt \\
\text{updt-t}[\![T, N, \langle \texttt{cont-rmv-e} \rangle]\!] = \langle evnt, T \rangle \\
\quad \text{subject to } \langle \texttt{cont-rmv-e}, N, \texttt{core}, \varnothing \rangle = evnt \\
\text{updt-t}[\![T, N, et, et_{rest}, \ldots]\!] = T_{new} \\
\quad \text{subject to updt-t}[\![T, N, et]\!] = T_1, \\
\qquad \text{updt-t}[\![T_1, N, et_{rest}, \ldots]\!] = T_{new}
\end{array}
$$

Figure 6. The updt-t metafunction

component contains all of the information that a debugger will need.

Consider an input program, M, and a reduction sequence, $M = M_0 \mapsto M_1 \mapsto \ldots \mapsto M_m = V$. In the CESK machine, the machine state after i reduction steps is $\langle\langle M_i, \mathcal{E}_i \rangle, \Sigma_i, K_i \rangle$ for $i \in [0, m]$. Events generated in the parallel reduction steps can reconstruct the machine states in the CESK machine. At each reduction step, $M_i \mapsto M_j$, in the CESK machine, events generated by the parallel step, $M_i \mapsto M_j$, can reconstruct the next machine state, $\langle\langle M_j, \mathcal{E}_j \rangle, \Sigma_j, K_j \rangle$ from a transitioning state, $\langle\langle M_i, \mathcal{E}_i \rangle, \Sigma_i, K_i \rangle$.

The trace does not contain all of the information that will be needed to construct a *domain-specific* view of the computation, however. Although domain-specific details can be encoded in aspects of the CESKT machine (analogous to encoding numbers as Church numerals), a more direct and useful approach is to add an extra instruction to the machine to support the logging of arbitrary events:

$$
\begin{array}{l}
\langle\langle \texttt{(core-emit}\ ename\ L\ A\texttt{)}, \mathcal{E} \rangle, \Sigma, K, T \rangle \\
\longrightarrow \langle\langle \texttt{true}, \varnothing \rangle, \Sigma, K, \langle evnt, T \rangle \rangle \\
\quad \text{subject to } \langle ename, L, \texttt{core}, A \rangle = evnt
\end{array}
$$

In fact, this rule makes the logging parts of all other CESKT rules redundant in the sense that a source program can be instrumented with `core-emit` forms to generate exactly the events that the other rules would record in the event trace. A separate report[2] contains an instrumentation metafunction and a proof that an instrumented program without built-in event logging produces the same trace as relying on built-in event logging.

The `core-emit` form, therefore, is our basis of generating events for debugging. Since `core-emit` can simulate built-in events, and since built-in events can reconstruct the CESKT machine's entire state, we know that `core-emit` provides all of the primitive debugging power that we will need.

5 Mapping Events

Using `core-emit` directly to implement DSL events would be as painful as programming DSLs using a pure λ-calculus directly to implement the DSL's evaluation. Our next step is to build a language for conveniently mapping events at one level of a language tower to events at the next level. That is, just as a module can implement a language layer by exporting macros that translate into the forms of a lower language level, a language-implementing module should export events that adapt the ones reporting during evaluation in the lower language level. Besides translating lower-level events to a new level, the macros of a language-implementing module can inject fresh `emit` calls (which are ultimately translated into `core-emit` core forms), and the events generated by those `emit` forms are part of the language's event interface.

5.1 Declaring Events

In the same way that `define-syntax` binds an identifier to a macro, the `define-event` form binds an identifier to an event description:

```
(define-event event-id composition-expr option ...)

composition-expr = event-id
                 | (seq composition-expr ...)
                 | (disj composition-expr ...)
                 | (conj composition-expr ...)
                 | (rep composition-expr n)

          option = #:when expr
                 | #:attributes ([id expr] ...)
                 | #:specific
```

The *`event-id`* in a *`composition-expr`* is a previously defined event, especially one from the language layers that the current module extends. The composition operators in a *`composition-expr`* are similar to the event expression operators in EBBA [Bates 1995] and involve the sequence operator, (`seq`), the disjunction operator, (`disj`), the conjunction operator, (`conj`), and the repetition operator, (`rep`).

The *`option`*s of a `define-event` form further control the generation and content of an event. A `#:when` expression is evaluated each time the event might be generated, and the generation is blocked if the `#:when` expression's result is

[2] http://www.cs.utah.edu/~xiangqi/dsl-events-appendix.pdf

false. An `#:attributes` option adds symbol-keyed information to the event. The `#:specific` option connects an event declaration to the evaluation of an (`emit` *`event-id`*) form.

An *`expr`* within an `#:attributes` option can access fields of the event matching *`composition-expr`* through an `attr` form to propagate or transform attributes values. When the event is declared with `#:specific`, then the expressions can additionally access variables that are in the environment of an associated (`emit` *`event-id`*) form.

To support parameterization over additional values for matching, an event can be defined as

```
(define-event (event-id arg-id ...) composition-expr
  option ...)
```

In that case, the *`arg-id`*s can be used in *`option`* expressions, and all references to *`event-id`* must have the form (`emit` (*`event-id arg-expr ...`*)).

An `emit` expression, which is typically generated by a macro, has either the form (`emit` *`event`*) or (`emit` *`event syntax-expr`*), where *`event`* is either *`event-id`* or (*`event-id arg-expr`*). When *`syntax-expr`* is included, it is used to associate program-source information with the event, and *`syntax-expr`* is typically a `#'` form that produces a template. An `emit` form can have an associated `define-event` form to specify the shape of the emitted event's *`event-id`* and adjust the way the event is reported.

Only events that are declared with `define-event` or `emit` and then exported with `export-event` are part of a module's event interface:

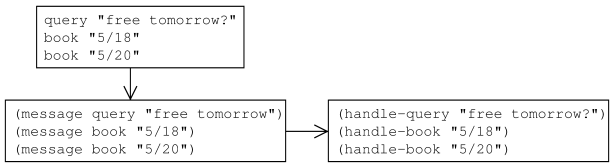
```
(export-event event-id ...)
```

Events generated by lower language levels are not automatically propagated as events from the new language layer. In the unusual case that a module imports from different language modules, the macros and events of all imported languages become visible to the importing module. More typically, however, a module imports a single language module, and so it sees only the syntactic forms (implemented by macros) and events of that language.

5.2 Examples and Kinds of Event Mappings

When performing event mapping from one language over another language, a variety of situations arise. Sometimes, events from the lower-level language can be propagated with small changes to the next layer. In other cases, events in a language correspond to a combination of events from a lower-level language and only when they happen in a particular evaluation context.

Suppose that we have a DSL program expanded to the forms in the right box:



Each statement in the DSL program is parsed as a `message` S-expression, such as `book "5/20"` parsed as

```
(message book "5/20")
```

and the following macro specifies the `message` syntax expansion in terms of the lower-level language's `handle-query` and `handle-book` constructs:

```
(define-syntax (message stx)
  (syntax-case stx (query book)
    [(_ query m)
     #'(handle-query m)]
    [(_ book date)
     #'(handle-book date)]))
```

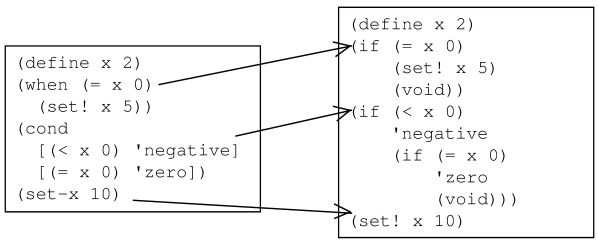
A `message` statement in the DSL translates to a function call in a procedural language. Suppose further that the procedural language generates a `subroutine-e` event when a defined function is called, where `subroutine-e` is parameterized by the name of the function to constrain an event consumption.

If we want a `message-e` event to be associated with the execution of a `message` statement, we can take advantage of the fact that each statement is equal to the call of a `handle-query` or `handle-book` function. We can define the `message-e` event in terms of events associated with `handle-query` and `handle-book`:

```
(define-event message-e
  (disj (subroutine-e #:name 'handle-query)
        (subroutine-e #:name 'handle-book)))
```

We categorize `message-e` as a **generic event**, since the event can just be defined through `define-event` to specify the composition relationship of events.

As another example, suppose we have an L_2 language that just extends a lower-level L_1 language with conditional clauses, such as `when`, `cond`, and a new assignment syntax `set-x`.



The related macros are:

```
(define-syntax (when stx)
  (syntax-case stx ()
    [(_ test then ...)
```

```
     #'(if ....)])) ; details omitted
(define-syntax (cond stx)
  (syntax-case stx ()
    [(_ (test then ...) rest ...)
     #'(if ....)]))
(define-syntax (set-x stx)
  (syntax-case stx ()
    [(_ v)
     #'(set! x v)]))
```

with when and cond expanded into `if` forms and `set-x` expanded into `set!`.

Because L_2 is an embedded language, without instrumentation of event mapping for L_2, the surface syntax in L_2 that belongs to L_1 should automatically have event support defined by L_1. For example, suppose that `set!` is associated with a `variable-update-e` event in L_1 where the event is exported to L_2. The execution of the `(set! x 5)` statement in L_2 will automatically generate a `variable-update-e` event. We call that kind of event a **host event**.

Even though `(set-x 10)` expands to `(set! x 10)`, the `variable-update-e` event associated with `set!` in L_1 will not be lifted to L_2's event stream automatically. The event must be specifically propagated with an `emit` declaration. Avoiding automatic propagation helps hide internal implementation details. For example, a macro for a form that involves no explicit assignment might be transformed into a sequence that involves assignment,

```
....
(set! ....)
....
(set! ....)
....
```

but where the effects are local and not exposed. In the case of `set-x`, however, the effect is exposed and explicit; the `set-x` acts as a syntactic sugar over `set!` and shares the same semantics. Instrumenting the `set-x` macro with an explicit `emit` specification makes it part of the language's interface in the case of a `set-x` expansion.

```
(define-syntax (set-x stx)
  (syntax-case stx ()
    [(_ v)
     (with-syntax ([cur-stx stx])
       #'(begin
           (emit variable-update-e #'cur-stx)
           (set! x v)))]))
```

We call this kind of `variable-update-e` event an **embedded event**.

Finally, if we want to create a `cond-e` event for L_2's `cond` construct, we can express `cond-e` in terms of events associated with the underlying `if` expansion:

```
(define-event cond-e if-e)
```

This declaration registers a listener for the low-level `if-e` event and triggers `cond-e` recognition upon `if-e` event generation. However, the `cond` form can expand into several `if` forms along with the `if` form generated by the `when` macro expansion, which would cause multiple `cond-e` event generation even if our program just contains one `cond` form. In consequence, we need a different kind of event—an **explicit event**—to specify runtime context. The explicit event is declared with a `#:specific` option:

```
(define-event cond-e if-e #:specific)
```

and we need to add an `(emit cond-e)` form inside the macro transformations for `cond` to specify the desired emission point in evaluation:

```
(define-syntax (cond stx)
  (syntax-case stx ()
    [(_ (test then ...) rest ...)
     (with-syntax ([cur-stx stx])
       #'(begin
           (emit cond-e #'cur-stx)
           (if ....)))]))
```

5.3 Environment Information in Events

The `define-event` form supports capturing debugging information in event attributes either by extracting information from constituent events or by obtaining information at `emit` sites by directly using identifiers when the `#:specific` option is used. The `state-e` and `receive-msg-e` events illustrate the uses.

```
(define-event state-e construct-e
  #:attributes ([state
                 (attr construct-e 'bindings)]))
```

By using the `attr` form, the `state-e` event can access the `bindings` attribute value of `construct-e`.

```
(define-event receive-msg-e construct-e
  #:attributes ([message m]
                [type (last (message-tags m))]
                [values (message-values m)]
                [msg-time (message-time m)])
  #:specific)

(define-syntax (module-begin stx)
  (syntax-case stx ()
    [(_ decl ...)
     (with-syntax ([cur-stx stx])
       #'(base-module-begin
           ....
           (define handle-msg
             (lambda (m)
               (emit receive-msg-e #'cur-stx)
               (eval-msg m)))
           ....
           decl ...))]))
```

By declaring `#:specific`, the `receive-msg-e` event is able to access `m`, `message-tags`, `message-values`, and `message-time` identifiers that are available in the environment of the associated `emit`.

In the CESKT model, `construct-e` captures all bindings in the environment at the point that the continuation is extended. Propagating all such bindings in an expanded program would reveal too many implementation details of

expansion. For example, in Figure 2, the expansion of the `point` program into the `procedure` language introduces a new `point` identifier representing a function, which is irrelevant to the DSL program; a user of the DSL should see only that the program creates a `p` binding.

Consequently, the bindings created inside a macro expansion are not automatically collected by `construct-e`. To help programmers declare which bindings should be exposed for a given language layer, our system includes a `new-bindings` form that cooperates with `construct-e`. For the point example, we can add `new-bindings` to the previous `define-point` macro in section 3:

```
(define-syntax (define-point stx)
  (syntax-case stx (x y)
    [(_ name (x x-expr) (y y-expr) (op arg ...) ...)
     (with-syntax ([point-id
                    (syntax-local-introduce #'point)])
       #'(begin
           (new-bindings name #'name)
           (define name (point-id x-expr y-expr))
           (send name op arg ...) ...))]))
```

This `new-bindings` declaration causes `construct-e` emitted by the lower-level language to include the instantiation of `name` in its environment attribute.

5.4 Continuation Information in Events

Similar to environment information in emitted events, language implementations that expose continuation events need to control the emission of events to reflect the language's own continuation points, as opposed to the continuation points of the underlying language. The continuation events of the lower-level language will typically be too fine-grained and expose too much information. For example, using a simple `incr` form defined by

```
(define-syntax (incr stx)
  (syntax-case stx ()
    [(_ v amt)
     #'(set v (+ v amt))]))
```

an expression `(incr v 20)` expands into `(set v (+ v 20))`, which has three subexpressions that generate continuation frames, as opposed to the original expression's single subexpression.

To aid the construction of suitable continuation events, our implementation provides `new-continuation` and `remove-continuation` forms, which expand to emit `cont-add-e` and `cont-rmv-e` events. The following is an example of specifying continuation events for the `incr` construct:

```
(define-syntax (incr stx)
  (syntax-case stx ()
    [(incr v amt)
     (with-syntax ([cur-stx stx])
       #'(begin0
           (set v
             (begin
               (new-continuation "incr" #'cur-stx)
               (+ v amt)))
           (remove-continuation #'cur-stx)))]))
```

The `begin` and `begin0` forms both create sequences, but the `begin0` form returns the result of its first expression instead of its last expression. The form `(new-continuation "incr" #'incr)` emits a `cont-add-e` before the evaluation of `(+ v amt)`, and `(remove-continuation #'cur-stx)` emits a `cont-rmv-e` event afterward.

6 Runtime Event Generation

Our CESKT model with `core-emit` (section 4) shows how the expanded version of a DSL program can generate core-level events, although it leaves abstract how the resulting event stream is consumed. The `define-event` form and associated constructs (section 5), meanwhile, help language implementers specify filters and transformations of events to create a language-specific event interface. Runtime event generation and filtering ties these two pieces together. It consumes the low-level events generated by the core language, and based on the specification of events at each language layer between the core and a DSL, it emits events that are suitable for consumption by a DSL-specific debugger.

Figure 7 illustrates the overall pipeline. A DSL implementation implies both a compiler from the DSL forms to core forms, an instrumentation of the resulting core forms to emit events, and a dependency graph of output events on core events. The events emitted at runtime are triggered and transformed, based on the dependency graph, and a DSL debugger presents them to the user.

6.1 Event Dependency Construction

Figure 8 shows the representation of event-related structures as used by dependency construction. The dependency graph is represented by G. Each event node t has associated node information *cmpt*, and *layer* distinguishes events generated at different language layers. An `export-event` declaration modifies the *layer* tag to reflect each language layer.

The construction process starts with the bottom language and establishes event dependencies in a bottom-up fashion. According to the `define-event` definition for an event, event dependencies are created by connecting the event to the events that it depends on. For every event, `e`, exported by `export-event`, if there exists no event dependency for `e` at this language layer, which means that the `e` event is inherited from the lower-level language, we create a new dependency connecting `e` to the lower-level `e`.

The semantics of `define-event` can be formulated as a register-dep metafunction, which adds an event dependency into the G graph:

$$\begin{aligned}&\text{register-dep}[\![ename, comp, layer, cond\text{-}exp, attr\text{-}exp, bool, G]\!]\\ &= G[t \leftarrow cmpt]\\ &\text{subject to } ename@layer = t, \text{convert}[\![comp, layer]\!] = cnv,\\ &\qquad \text{get-node-type}[\![ename]\!] = ntype,\\ &\qquad \langle cnv, \langle cond\text{-}exp, attr\text{-}exp, ntype\rangle, bool\rangle = cmpt\end{aligned}$$

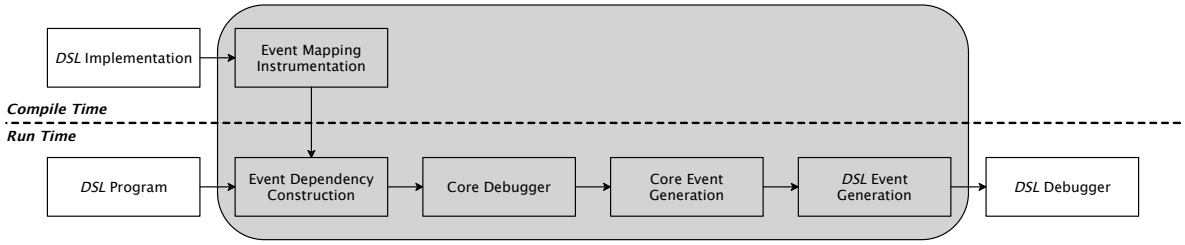


Figure 7. An event framework for domain-specific event creation and generation

$ename ::= etype \mid (etype\ V\ ...)$ $\quad state ::= evnt \mid \textsc{List}(evnt, ...)$
$evnt ::= \langle ename, src, layer, A\rangle$ $\quad mem ::= \textsc{List}(ename, ...)$
$t ::= ename@layer$

$R ::= \{(t\ \textsc{List}(t, ...))\ ...\}$ $\quad G ::= \{(t\ cmpt)\ ...\}$
$F ::= \{(t\ bool)\ ...\}$ $\quad P ::= \{(t\ \langle bool, A, emit\text{-}src\rangle)\ ...\}$
$C ::= \{(t\ \langle ctype, S, mem\rangle)\ ...\}$ $\quad S ::= \{(t\ state)\ ...\}$
$D ::= \{(t\ \langle cond\text{-}exp, attr\text{-}exp, ntype\rangle)\ ...\}$

$cmpt ::= \langle cnv, \langle cond\text{-}exp, attr\text{-}exp, ntype\rangle, bool\rangle$
$cnv ::= t \mid \langle ctype, \textsc{List}(t, ...)\rangle$
$ctype ::= \texttt{disj} \mid \texttt{conj} \mid \texttt{seq} \mid \langle \texttt{rep}, ro\rangle$
$ntype ::= \texttt{internal} \mid \texttt{abstraction} \mid \texttt{false}$

$ro ::= + \mid * \mid integer$ $\quad bool ::= \texttt{true} \mid \texttt{false}$
$emit\text{-}src ::= srcloc$ $\quad layer ::=$ a layer tag
$srcloc ::= \langle src, line, col\rangle$ $\quad comp ::=$ a composition expression

Figure 8. Event-related structure representation

where *comp*, *cond-exp*, *attr-exp*, and *bool* are obtained from the event definition. *comp* is a representation of the event definition's `composition-expr`, and the convert tags every event type in *comp* with the correct layer information. The *cond-exp* and *attr-exp* represent the expressions specified in an event definition's `#:when` and `#:attributes` options, and the *bool* denotes if an event is an explicit event. The *ntype* encodes the type of a node, whether it is an internal node created by the system or another kind of node.

6.2 DSL Event Generation

The preparation step for DSL event generation involves trimming event dependencies and building reverse references. To reduce the size of events, we keep only event dependencies needed by the top language level. As events originate from the core level, we need a push notification mechanism to trigger events at a higher language level. Every dependency, `e1` → `e2`, creates a reverse reference, `e2` → `e1`, and at runtime, the *event processing unit* listens to core events and generates a DSL event according to the reverse mapping of event dependencies and the event definition's constraints.

Formally speaking, after the completion of G construction, the reverse references to event nodes, R, is constructed, and the F, C, and D tables store different aspects of node information. The C table stores a mapping of a high-level tagged event to its recognition constraint and a recognition progress table, S.

The emission of core events initiates the DSL event generation process where the event processing unit looks up the parent nodes of an event in R, a list of event nodes to be triggered, and tries to trigger the parent node, t_p, one-by-one through trigger. Because the `emit` form affects the generation of an explicit event and an embedded event, a P table is used to record the evaluation of `(emit ename stx)` forms along with relevant event information:

$$\text{update-emit}[\![ename, A, stx]\!] = P\big[t \leftarrow \langle \texttt{true}, A, srcloc\rangle\big]$$
$$\text{subject to } \text{get-layer-info}[\![stx]\!] = layer, ename@layer = t,$$
$$\text{get-srcloc}[\![stx]\!] = srcloc$$

A is a set of attribute mappings associated with the *ename* event. The trigger metafunction first checks if a t_p is an explicit event by looking up the F table and then attempts to generate an explicit event instance after the evaluation of `emit`. To illustrate, the case of trigger for a high-level, explicit t_p event is:

$$\begin{aligned}&\text{trigger}[\![evnt, t_p, src_s, P, F, C, D]\!] = \langle ename_p, src_s, layer_p, A_p\rangle\\ &\text{subject to } t_p = ename_p@layer_p,\\ &\quad evnt = \langle ename, src, layer, A\rangle,\\ &\quad F(ename@layer) = \texttt{true},\\ &\quad P(ename@layer) = \langle \texttt{true}, A_{emit}, srcloc\rangle,\\ &\quad C(t_p) = \langle ctype, S, mem\rangle, \text{updt-s}[\![S, evnt]\!] = S_{new},\\ &\quad \text{check-comp-constraint}[\![S_{new}, ctype, mem]\!] = \texttt{true},\\ &\quad D(t_p) = \langle cond\text{-}exp, attr\text{-}exp, ntype\rangle,\\ &\quad \text{get-comp-events}[\![C, D, t_p, mem]\!] = \textsc{List}(evnt_c, ...),\\ &\quad \text{check-cond-constraint}[\![cond\text{-}exp, \textsc{List}(evnt_c, ...)]\!] = \texttt{true},\\ &\quad \text{get-attributes}[\![attr\text{-}exp, \textsc{List}(evnt_c, ...), A_{emit}]\!] = A_p\end{aligned}$$

The check-comp-constraint metafunction checks whether the current recognition state meets its composition requirement on constituent events, and the check-cond-constraint metafunction checks if the *cond-exp* constraint is satisfied by obtaining its current matched constituent event values through get-comp-events. Since `emit` can either obtain attribute values from the `emit` evaluation context or from its event definition's constituent events, get-attributes returns the appropriate attribute values. The get-attributes metafunction also uses the updt-attributes metafunction to enrich the event with a `time` attribute and updates the `loc` attribute value if necessary.

The other cases of trigger generate a higher-level event dispatching on the t_p kind. If t_p is an embedded event, the generation rule is:

$$\begin{aligned}&\text{trigger}[\![evnt, t_p, src_s, P, F, C, D]\!] = \langle ename_p, src_p, layer_p, A_p\rangle\\ &\text{subject to } t_p = ename_p@layer_p,\\ &\qquad evnt = \langle ename, src, layer, A\rangle,\\ &\qquad P(ename@layer) = \langle \texttt{true}, A_{emit}, srcloc\rangle,\\ &\qquad \text{get-source}[\![srcloc, src]\!] = src_p,\\ &\qquad \text{updt-attributes}[\![A, srcloc]\!] = A_p\end{aligned}$$

Depending on the value of *srcloc*, get-source chooses a source value between *srcloc* and *src*.

If t_p is a primitive, generic event and the runtime condition of the event is satisfied, a new event instance is generated by packing appropriate attribute values from the event it depends on:

$$\begin{aligned}&\text{trigger}[\![evnt, t_p, src_s, P, F, C, D]\!] = \langle ename_p, src, layer_p, A_p\rangle\\ &\text{subject to } t_p = ename_p@layer_p,\\ &\qquad evnt = \langle ename, src, layer, A\rangle, C(t_p) = \textsc{None},\\ &\qquad D(t_p) = \langle cond\text{-}exp, attr\text{-}exp, ntype\rangle,\\ &\qquad \text{check-cond-constraint}[\![cond\text{-}exp, evnt]\!] = \texttt{true},\\ &\qquad \text{get-attributes}[\![attr\text{-}exp, evnt]\!] = A_p\end{aligned}$$

Otherwise, if t_p is a high-level, generic event, the event generation rule is similar to the high-level, explicit event case but with a different mechanism for generating A_p:

$$\begin{aligned}&\text{trigger}[\![evnt, t_p, src_s, P, F, C, D]\!] = \langle ename_p, src_s, layer_p, A_p\rangle\\ &\text{subject to } t_p = ename_p@layer_p, C(t_p) = \langle ctype, S, mem\rangle,\\ &\qquad \text{updt-s}[\![S, evnt]\!] = S_{new},\\ &\qquad \text{check-comp-constraint}[\![S_{new}, ctype, mem]\!] = \texttt{true},\\ &\qquad D(t_p) = \langle cond\text{-}exp, attr\text{-}exp, ntype\rangle,\\ &\qquad \text{get-comp-events}[\![C, D, t_p, mem]\!] = \textsc{List}(evnt_c, ...),\\ &\qquad \text{check-cond-constraint}[\![cond\text{-}exp, \textsc{List}(evnt_c, ...)]\!] = \texttt{true},\\ &\qquad \text{get-attributes}[\![attr\text{-}exp, \textsc{List}(evnt_c, ...)]\!] = A_p\end{aligned}$$

In the process of upward event generation, the trigger metafunction updates events with appropriate *src* information so that events belonging to different language layers can be differentiated. In the end, the event processing unit just keeps the event instances at the DSL level and directs these events to event handlers set up for a debugger implementation.

7 Application

We implemented a debugging framework, Ripple, on top of the event framework presented in this paper. Ripple relies on the event framework to instrument DSLs with domain-specific events, where the generated events interact with the debugger front end to enable domain-specific debugging techniques. We worked with three DSLs:

- Scratchy:[3] an imperative language for writing Scratch-like applications.
- POP-PL [Florence et al. 2015]: a declarative, reactive prescription language for automating medical treatment.
- Medic [Li and Flatt 2015]: a metaprogramming language for trace-oriented debugging.

We built three domain-specific debuggers: a Scratchy debugger, a POP-PL debugger (Figure 1), and a Medic debugger. Because Scratchy is imperative, the Scratchy debugger provides a statement-by-statement stepping facility similar to a step-based debugger. The POP-PL debugger records all message logs in the system and enables navigation to message origins, and the Medic debugger allows users to observe Medic program transformation effects on another program.

Instead of mapping DSL concepts to the traditional GPL debugging events, we started with a front-end design to decide what domain concepts to show and mapped domain concepts to domain-specific events for debugger interaction. For example, the domain-specific events we created are: `construct-e` with customized bindings for Scratchy, `send-msg-e` and `receive-msg-e` for POP-PL (explicit events defined in terms of `construct-e`), and `module-entry-e` and `insert-e` for Medic (an explict event defined in terms of `construct-e`). Debugging events serve as back-end representation of program states, and the debugger front-end component operates on the debugging events and implements debugging techniques tailored to domain needs. The debugger interface reacts to debugging events according to specified event handlers.

Domain-specific events free us from a dependence on GPL debugging events, and they have helped us to build domain-specific debuggers with flexible debugging operations. Since our event design allows reusing events from other languages, and since the pattern matching abilities of macros allow central event instrumentation for a category of language constructs, the amount of event instrumentation work is relatively small compared to the debugger interface implementation. For example, Scratchy provides 17 operations for object manipulation, such as `move x by` *`expr`* and `forward by` *`expr`*, but these operations are implemented by just one `define-sprite-method` macro as a single point of control. Event instrumentation with `define-event` or `emit` for Scratchy, POP-PL, and Medic involves 18, 12, and 11 lines of code, respectively, while the interface implementation is mostly over 100 lines of code.

8 Related Work

Event-based debugging for GPLs include Coca [Ducasse 1999], RAIDE [Johnson 1977], Dalek [Olsson et al. 1990], EBBA [Bates 1995], UFO [Auguston et al. 2003], and MzTake [Marceau et al. 2006]. Each system employs a different event model, but the event models in Dalek and EBBA are similar to ours. In addition to primitve events, Dalek provides a machanism for defining high-level events. EBBA views debugging as a process of building models of expected program behavior where the model is based on events. EBBA uses sequential, choice, concurrency, and repetition event expression operators to model the behavior of a program, which inspired our design for high-level events.

Debugging support for DSLs is relatively new compared to debugging support for GPLs. DDF [Wu et al. 2008], LISA [Henriques et al. 2005], and TIDE [Van Den Brand et al. 2005] rely on the DSL grammar or language specification to enable debugger support. In DDF, the DSL grammar is augmented

[3] https://docs.racket-lang.org/scratchy

with additional code to address the abstraction gap from a GPL debugger to a DSL debugger, and the code can also be weaved with an aspect-oriented approach [Wu et al. 2005]. Lindeman et al. [2011] propose implementing a debugger declaratively with event mappings to four fixed events. The moldable debugger [Chis et al. 2014] supports customizing events based on a common set of primitive events. However, we allow event customization on top of flexible events.

In the realm of domain-specific modeling languages, events are also used to observe and control the behavior of a model. BCOoL [Deantoni 2016] relies on domain-specific events to coordinate the behavior of heterogeneous languages. A partly generic omniscient debugging [Bousse et al. 2015], which was proposed for executable domain-specific modeling languages, uses events to capture execution states needed for omniscient debugging. In comparison, our core events aim at capturing whole machine states including information about continuations, and our more general event model offers a means to capture a variety of information. For non-executable models, model simulators and model transformations define the semantics of models, which can also be enabled with debugging support. Simulators can be instrumented with debugging operations [Van Mierlo et al. 2017], and an omniscient debugging technique can also be provided for model transformations [Corley et al. 2017].

Acknowledgments

This work was supported by the National Science Foundation through grant number CNS-1526324.

References

Mikhail Auguston, Clinton Jeffery, and Scott Underwood. A Monitoring Language for Run Time and Post-Mortem Behavior Analysis and Visualization. In *Proc. Fifth Intl. Wksp. on Automated Debugging*, 2003.

Peter C. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. *ACM Transactions on Computer Systems* 13(1), 1995.

Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. Supporting Efficient and Advanced Omniscient Debugging for xDSMLs. In *Proc. Software Language Engineering*, 2015.

Stephen Chang, Alex Knauth, and Ben Greenman. Type Systems as Macros. In *Proc. ACM Sym. Principles of Programming Languages*, 2017.

Andrei Chis, Tudor Girba, and Oscar Nierstrasz. The Moldable Debugger: a Framework for Developing Domain-Specific Debuggers. *Software Language Engineering* 8706, 2014.

Jonathan Corley, Brian P. Eddy, Eugene Syriani, and Jeff Gray. Efficient and Scalable Omniscient Debugging for Model Transformations. *Software Quality Journal* 25(1), 2017.

Julien Deantoni. Modeling the Behavioral Semantics of Heterogeneous Languages and their Coordination. In *Proc. Architecture Centric Virtual Integration*, 2016.

Mireille Ducasse. Coca: An Automated Debugger for C. In *Proc. Intl. Conf. on Software Engineering*, 1999.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In *Proc. 1st Summit on Advances in Programming Languages*, 2015.

Mattias Felleisen and Daniel P. Friedman. A Calculus for Assignments in Higher-Order Languages. In *Proc. ACM Sym. Principles of Programming Languages*, 1987.

Spencer P. Florence, Burke Fetscher, Matthew Flatt, William H. Temps, Tina Kiguradze, Dennis P. West, Charlotte Niznik, Paul R. Yarnold, Robert Bruce Findler, and Steven M. Belknap. POP-PL: A Patient-Oriented Prescription Programming Language. In *Proc. Generative Programming and Component Engineering*, 2015.

Pedro Rangei Henriques, Maria Joao Varanda Pereira, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic Generation of Language-Based Tools Using the LISA System. *IEE Proceedings - Software* 152(2), 2005.

Mark Scott Johnson. The Design of a High-Level, Language-Independent Symbolic Debugging System. In *Proc. ACM '77 Annual Conf.*, 1977.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proc. ACM Sym. on Lisp and Functional Programming*, 1986.

Xiangqi Li and Matthew Flatt. Medic: Metaprogramming and Trace-Oriented Debugging. In *Proc. Wksp. on Future Programming*, 2015.

Ricky T. Lindeman, Lennart C. L. Kats, and Eelco Visser. Declaratively Defining Domain-Specific Language Debuggers. In *Proc. Generative Programming and Component Engineering*, 2011.

Guillaume Marceau, Gregory H. Cooper, Jonathan P. Spiro, Shriram Krishnamurthi, and Steven P. Reiss. The Design and Implementation of a Dataflow Language for Scriptable Debugging. *Automated Software Engineering* 14(1), 2006.

Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys* 37(4), 2005.

Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho. Dalek: A GNU, Improved Programmable Debugger. In *Proc. USENIX Technical Conf.*, 1990.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as Libraries. In *Proc. ACM Conf. Programming Language Design and Implementation*, 2011.

Mark G. J. Van Den Brand, B. Cornelissen, Pieter A. Olivier, and Jurgen J. Vinju. TIDE: A Generic Debugging Framework — Tool Demonstration. *Electronic Notes in Theoretical Computer Science* 141(4), 2005.

Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices* 35(6), 2000.

Simon Van Mierlo, Claudio Gomes, and Hans Vangheluwe. Explicit Modelling and Synthesis of Debuggers for Hybrid Simulation Languages. In *Proc. Sym. on Theory of Modeling and Simulation*, 2017.

Martin P. Ward. Language Oriented Programming. *Software—Concepts and Tools* 15(4), 1994.

Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-Driven Generation of Domain-Specific Language Debuggers. *Software—Practice & Experience* 38(10), 2008.

Hui Wu, Jeff Gray, Suman Roychoudhury, and Marjan Mernik. Weaving a Debugging Aspect into Domain-Specific Language Grammars. In *Proc. Sym. on Applied Computing*, 2005.