Super 8 Languages for Making Movies
(A Functional Pearl)

LEIF ANDERSEN, STEPHEN CHANG, MATTHIAS FELLEISEN, PLT @ Northeastern University

The Racket doctrine tells developers to create languages (as libraries) to narrow the gap between the terminology of a problem domain and general programming constructs. This pearl illustrates this doctrine with the creation of a relatively simple domain-specific language for editing videos. To produce the video proceedings of a conference, for example, video professionals traditionally use "non-linear" GUI editors to manually edit each talk, despite the repetitive nature of the process. As it turns out, the task of video editing naturally splits into a declarative phase and an imperative rendering phase at the end. Hence it is natural to create a functional-declarative language for the first phase, which reduces a lot of manual labor. The implementation of this user-facing DSL, dubbed Video, utilizes a second, internal DSL to implement the second phase, which is an interface to a general, low-level C library. Finally, we inject type checking into our Video language via another DSL that supports programming in the language of type formalisms. In short, the development of the video editing language cleanly demonstrates how the Racket doctrine naturally leads to the creation of language hierarchies, analogous to the hierarchies of modules found in conventional functional languages.

CCS Concepts: •**Software and its engineering** → **Functional languages; Preprocessors; Macro languages; Specification languages;** *Designing software;* •**Information systems** → *Multimedia content creation;* •**Applied computing** → *Media arts;* •**Human-centered computing** → Graphical user interfaces;

Additional Key Words and Phrases: Domain-Specific Language, Declarative Languages, Video Editing, Syntax Elaboration, Language Oriented Design, Movies, Integrated Development Environment

1 BEING IAVOR DIATCHKI

Imagine being Iavor Diatchki. He is the friendly guy who records all the wonderful ICFP presentations, edits them into digestible video clips, and finally creates a YouTube channel for the whole conference. When he creates the video clips, he combines a feed of the presenter with the presenter's screen, the sound feed for the speaker, and yet another one for audience questions. Additionally, Diatchki must add a start and end sequence to each video plus various watermarks throughout.

Once one video is put together, the same process must be repeated for the next conference talk and the next and so on. Worse, even though some editing steps involve creativity, the process becomes so monotonous that it reduces the creative spirit for when it is truly needed.

The problem cries out for a declarative language, especially because the state of the art for video editing suggests (see section 3) that professionals in this domain already think "functionally." To wit, professionals speak of "non-linear video editing" (NLVE) to highlight the idea that the process is

non-destructive. Technically, the editing process separates a descriptive phase—what the eventual video is supposed to look like, given existing tracks—from the rendering phase—which actually creates the video clip from these descriptions.

This pearl presents *Video* (section 4), a scripting language for video production. Video turns video editing upside down. Instead of sitting for hours on end in front of some NLVE GUI, a professional can now spend a few minutes in front of an IDE to create a Video script and, voilà, a video clip pops out. A Video script is just a sequence of expressions, which describe fragments of a video clip, and definitions, which introduce constants for and functions on video clips. Running such a script turns this description into suitable "assembly code" for a video renderer.

Speaking of an IDE, everyone knows that in this day and age a programming language comes with a whole suite of gadgets, that is, the programming environment *around* the language. We therefore throw in an IDE (section 7) and a dependent type system (section 6), not only because these might be useful for producing video channels, but because a real functional pearl deserves this much attention. Thus an Agda-trained programmer may add types to Video modules. And better still, Video code may include an NLVE widget, which may of course come with embedded Video code, which may contain another NLVE widget, ... Did we mention turtles yet?[1]

Abstractly speaking, Video once again demonstrates the power of the Racket doctrine (section 2). Racket hosts Video as an embedded domain-specific language. Implementing (section 5) this language in Racket takes only a small effort because of Racket's powerful language-production language. Indeed, adding an IDE and a type system is also a matter of a few hours of thinking time and coding. In this spirit, the pearl points out a key difference between the construction of embedded DSLs in conventional functional languages and the one true-blue kid on the block, Racket.

2 DR STRANGELOVE: HOW I LEARNED TO STOP WORRYING AND LOVE RACKET

The Racket doctrine (Felleisen et al. 2015) says that developers must feel empowered to easily create and deploy a new language when a software development situation demands it. The emphasis on language reflects the Racket team's deep embrace of the Sapir-Whorf hypothesis. In the world of software, this hypothesis means that language frames the developer's ability to understand a problem and articulate a solution—at every stage in the development process.

Philosophically, Racket achieves this ideal with a radical emphasis on linguistic reuse (Krishnamurthi 2001). Technically, this reuse is enabled via Racket's distinctive feature: a modular syntax system (Flatt 2002). In this system, it is easy to import a linguistic construct as a function; indeed the system blurs the distinction between languages and libraries, e.g. Tobin-Hochstadt et al. (2011). While a library module exports functions with a related purpose, a language module exports the constructs of a programming language.

In Racket, every module must first import its language definition, via a one-line specification. For example, `#lang racket/base`—pronounced "hash lang racket base"—tells Racket and a future reader that the module is written in the `racket/base` language. This specification points to a file containing the language implementation that, approximately speaking, consists of a suite of linguistic features and run-time functions. A developer can thus edit a language L in one buffer of an IDE and an L program in a second one. Any change to the first is immediately visible in the second one, just by switching between tabs. Thus, language development in Racket suffers from no points of friction.

Developing a new language typically starts from a base language close to the desired one. From there, a Racket developer creates a new language with some or all of the following actions:

- adding new linguistic constructs;
- hiding linguistic constructs; and

---

[1] See en.wikipedia.org/wiki/Turtles_all_the_way_down, last visited Feb 20, 2017.

- re-interpreting linguistic constructs.

Video exploits all of the above. Here, linguistic constructs are any functions or syntactic extensions added to a language such as list comprehensions or pattern matching. For the re-interpretation of linguistic constructs, Racket developers heavily rely on linguistic interposition points, that is, anchors in the syntax elaboration process where a program may inject additional syntax transformations.

Due to the ease of developing and installing languages in the Racket ecosystem, the language creation "warhead" is Racket's distinguishing weapon, akin to Haskell's type classes or ML's functors, in its arsenal of software-engineering tools. When developers realize that it is best to express themselves in the language of a domain, they do not hesitate to develop a matching programming language. After all, domain experts have developed this specialized terminology (and ontology) so that they can discuss problems and solutions efficiently.[2]

The domain of video editing is a particularly well-suited domain for illustrating the above points. While the evolution of the language follows the standard path from a veneer for a C library to a full-fledged language (Fowler and Parsons 2010), Racket reduces this path significantly and this pearl demonstrates how. Before we can describe Video and its implementation, however, we need to survey the world of editing videos.

3 PRIMER

People use so-called non-linear video editors (NLVEs) to compose video clips (Dancyger 2010). In the context of film production, *non-linear* means non-destructive, that is, the source videos do not degrade in quality due to editing.[3] A NLVE is a graphical tool with a *time line* of *tracks*. Each track describes a composition of video clips, audio clips, and effects playing in sequence. The NLVE *renders* these tracks by playing them all simultaneously, placing one track on top of the other. Obviously a screen displaying the result can play only a single track for video;[4] by convention this is the last or top track. Video editors use effects to *composite* tracks. That is, effects splice two or more tracks together so that they appear on the screen at the same time. Technically, the renderer uses these effects to combine tracks as they play and either output the result to a file or play it on a screen.

Over time, professionals have developed tools and design patterns to reduce the amount of repetitive manual labor in video editing. They frequently develop so-called macros—a scripted sequence of user interface elements—in languages such as AppleScript (Cook 2007). Some professional tools, such as Adobe Premiere (Jago and Adobe Creative Team 2017), even include an API to create script-style plug-ins directly. Extending tools in this fashion has limits. These kinds of macros are extremely brittle and frequently break, even within a single application, because these macro languages essentially specify dialog box clicks without understanding the underlying tools.

Using a tool's official plug-in interface produces reasonably robust scripts but yields plug-ins that are tightly coupled with its tool. They can be used only when the entire toolchain is present. Blender (Roosendaal and Hess 2007), for example, is only scriptable with a Blender-specific Python interpreter that runs when Blender is launched.

Alternative approaches use general-purpose multimedia frameworks such as GStreamer (Taymans et al. 2013) or the MLT Framework.[5] These frameworks are APIs for C-like languages that provide data types for building and rendering videos. These frameworks are primarily used in two situations. First, they are the back-ends to NLVEs. For example, MLT is the backend for both Shotcut[6] and

---

[2]Not every DSL is a week or month-long project. Turnstile (Chang et al. 2017) took almost a year of development time!

[3]Digital editors achieve this result by operating on references to videos, rather than operating on the videos themselves.

[4]Audio tracks can actually be played simultaneously.

[5]mltframework.org/

[6]shotcutapp.com/

Kdenlive.[7] Second, professionals use these frameworks to batch-process videos, particularly when interactive development is not desired.

The appeal of these frameworks comes from their ability to create abstractions, such as functions, to handle otherwise repetitive tasks. Using these frameworks quickly becomes cumbersome, however, when there is a need to combine interactive and programmatic work flows, as is the case for the creation of conference recordings. Thus, studios tend to stick with NLVEs and use these frameworks only sparingly.

Professionals also use domain-specific languages for video editing. These DSLs primarily fall into two categories: XML-based DSLs and scripting-based DSLs. XML DSLs such as MLT XML and the now-deprecated SMIL (Bulterman et al. 2008) offer declarative languages for processing videos. These languages generally do not have functions or any other type of abstraction, however, and thus professionals tend not to deal with these XML languages directly when video editing. Rather, NLVEs use these languages as a file format to save video projects.

Scripting-based DSLs such as AVISynth[8] are declarative and support functions and other abstractions, but have their own limitations. They typically support only the simplest of tasks such as playing videos in sequence with transitions and minor visual effects. They also tend to lack any formal grammars and use a small script for a parser. Finally, they lack many code reuse features. AVISynth, for example, allows programmers to create simple functions but comes without any control flow constructs such as conditional branching.

4   THE PRODUCERS

The literature survey in the preceding section suggests that non-linear video editing distinctly separates the description of a video clip from the rendering action on it. Specifically, an editor (as in the tool) needs a description of what the final video should look like in terms of the given pieces. The action of rendering this video is a distinct second step. Going from this assessment to a language design requires one more idea: abstraction. For example, a description of a video composition should be able to use a sequence comprehension to apply a watermark to all images. Similarly, a professional may wish to create one module per ICFP presentation in order to make up a complete ICFP channel in a modular fashion. And of course, the language must allow the definition and use of functions because it is the most common form of abstraction.

The Video language gets to the heart of the domain. Each Video program is a complete module that intermingles descriptions of video clips and auxiliary definitions. It denotes a Racket module that exports a playlist description of the complete video. One way to use a Video module is to create a video with a renderer. A different way is to import it into a second Video module and to incorporate its exported playlist into another video.

Figure 1 displays a simple Video script. It consists of five expressions, each describing a piece of a video clip. Right below the third part of the video description, it also contains two definitions, which introduce one name each so that the preceding `multitrack` description does not become too deeply nested. When the renderer turns this script into an actual video, it turns the five pieces into sequence of images, taking into account the transitions between the first and second fragment and the fourth and the fifth.

In essence, the script in figure 1 assembles the visual part of a simple conference video. What is missing is the audio part. Naturally, a Video programmer should abstract over this sequence of expressions, plus the audio processing, and create a suitable library. Figure 2 shows what the same script roughly looks like after a Video programmer has encapsulated an abstraction over the script in

---

[7]kdenlive.org/
[8]avisynth.nl

```
01 #lang video
02
03 (image "splash.png" #:length 100)
04
05 (fade-transition #:length 50)
06
07 (multitrack (blank #f)
08              (composite-transition 0 0 1/4 1/4)
09              slides
10              (composite-transition 1/4 0 3/4 1)
11              presentation
12              (composite-transition 0 1/4 1/4 3/4)
13              (image "logo.png" #:length (producer-length talk)))
14
15 ; where
16 (define slides
17   (clip "slides05.MTS" #:start 2900 #:end 80000))
18
19 (define presentation
20   (playlist (clip "vid01.mp4")
21             (clip "vid02.mp4")
22             #:start 3900 #:end 36850))
23
24 (fade-transition #:length 50)
25
26 (image "splash.png" #:length 100)
```


Figure 1: A first Video script

```
01 #lang video
02
03 (require "conference-lib.vid")
04
05 (conference-talk video slides audio 125)
06 ; where
07 (define slides (clip "slides05.MTS" #:start 2900 #:end 80000))
08 (define video  (playlist (clip "vid01.mp4") (clip "vid02.mp4")
09                          #:start 3900 #:end 36850))
10 (define audio  (playlist (clip "capture01.wav") (clip "capture02.wav")))
```


Figure 2: A Video description of a conference talk

figure 1 as a utility library. This program uses the imported `conference-talk` function to combine a recording of the speaker, a capture of the slides, and the audio. As mentioned, line 1 of the program specifies that this module is written in the Video language. Line 3 imports the library that defines the `conference-talk` function. Line 5 produces the video that this module describes. Finally, the remainder is a sequence of definitions that introduce auxiliary constants.

Figure 3 shows the essence of the utility library, also written as a Video module. Explaining its construction introduces enough of Video's primitives and combinators to get a sense of what the rest of the language looks like. First we explain Video's primary linguistic mechanisms, modules and functions (section 4.1). Then, we describe basic producers (section 4.2): images, clips, colors and so on, following up with a discussion of the basics of how to combine these producers into playlists and multitracks (sections 4.3, 4.5). To make compelling examples, we introduce transitions, filters, and properties (sections 4.4, 4.6, 4.7). Finally, we describe the interface for displaying videos and rendering Video programs (section 4.8).

4.1 Essential Video

Video modules consist of a series of interleaved expressions, definitions, and import/export forms; functions have the same shape as modules but without the import/export forms. Video enforces different scoping rules, and assigns slightly different meaning to these constructs, than Racket does. In both cases, definitions are valid in the entire scope—that is, the entire module or the entire function body. The expressions must describe video playlists. Modules and functions differ in that the former `provide`s a video, while the latter returns one. Furthermore, Video modules are first-order entities that can be compiled separately, while functions are actually first-class values.

Now, take a second look at figure 3. Lines 5 through 7 show the function header. The rest of the code describes the function body (lines 7–28). Functions in Video are declarative; in particular, line 8 is the producer returned by this function. The remaining subsections explain the Video language in sufficient detail to understand the rest of this code. Specifically, we explain individual features of the language and how they improve the video editing process.

4.2 Producers

The *producer* is the most basic building block for a Video program. A producer evaluates to a data structure that denotes some sort of multimedia object: audio clips, video clips, pictures, and so on. Combinations of producers are themselves producers, and they can be further combined into yet more complex producers.

The simplest type of producer, `clip`, incorporates traditional video files. The `clip` producer converts the file into a sequence of frames. Developers use `clip` to import recordings from files, such as a conference talk, into scripts:

```
#lang video

(clip "talk00.mp4")
```


Unlike `clip`, `image` creates an infinite stream of frames. Video's combination forms truncate these streams to fit the length of other producers. Additionally, developers can use the `#:length` keyword when they want a specific number of frames:

```
#lang video

(image "splash.png" #:length 3)
```

```
01 #lang video
02
03 (provide conference-talk)
04
05 ; Describes an edited conference video with appropriate feeds
06 ; Producer Producer Producer Positive-Integer -> Producer
07 (define (conference-talk video slides audio offset)
08   (multitrack clean-video clean-audio)
09   ; where
10   (define clean-audio
11     (playlist (blank offset)
12               (attach-filter audio
13               (envelope-filter 50 #:direction 'in)
14               (envelope-filter 50 #:direction 'out))))
15   (define spliced-video
16     (multitrack (blank #f)
17                 (composite-transition 0 0 1/4 1/4)
18                 slides
19                 (composite-transition 1/4 0 3/4 1)
20                 video
21                 (composite-transition 0 1/4 1/4 3/4)
22                 (image "logo.png" #:length (producer-length talk))))
23   (define clean-video
24     (playlist (image "splash.png" #:length 100)
25               (fade-transition #:length 50)
26               spliced-video
27               (fade-transition #:length 50)
28               (image "splash.png" #:length 100))))
```

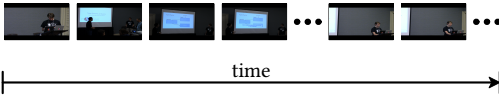Figure 3: A Video function definition

4.3 Playlists

A video is usually a composition of many producers. Video provides two main ways for combining them: *playlists* and *multitracks*. Roughly speaking, playlists play clips in sequence, while multitracks play clips in parallel. Any producer can be put in a `playlist`, including another `playlist`. Each clip in the `playlist` plays in succession. Frequently, video cameras split recordings into multiple files. With `playlist`, developers can easily stitch these files together to form a single producer:

```
#lang video

(playlist (clip "talk00.MTS")
          (clip "talk01.MTS"))
```


Developers cut playlists and other producers to desired lengths with the `#:start` and `#:end` keywords. This capability is included because video recordings frequently start before a talk begins and end after the talk finishes; see figure 4a. Recall that while `define` is located below the video description, its scope includes the preceding expressions.

4.4 Transitions

Jumping from one producer in a playlist to another can be rather jarring. Scripts can reduce this effect with *transitions*: fading, swiping, etc. These transitions merge the two adjacent clips in a playlist and are placed directly inside of (possibly implicit) playlists. Expanding on the example

```video
#lang video

(image "logo.jpg" #:length 100)
talk
(image "logo.jpg" #:length 100)

; where
(define talk
 (playlist (clip "talk00.MTS")
           (clip "talk01.MTS")
           #:start 100
           #:end 8000))
```


Figure 4: (a). Example of cutting a playlist

```video
#lang video

(image "splash.png" #:length 100)
(fade-transition #:length 50)
talk
(fade-transition #:length 50)
(image "splash.jpg" #:length 100)

; where
(define talk ⟨···elided···⟩)
```
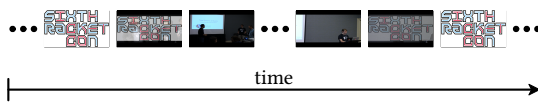

Figure 4 (continued): (b). Example with fading transition

```video
#lang video

(multitrack
 (clip "slides.mp4")
 (composite-transition 0 0 1/4 1/4)
 talk)

; where
(define talk ⟨···elided···⟩)
```


Figure 4 (continued): (c). Example using multitracks

from figure 4a, fade transitions are used to smooth the transition from logo to the talk. Figure 4b illustrates this point.

Every transition in a `playlist` actually shortens the length of its frame sequence, because transitions produce one clip for every two clips they consume. Additionally, a `playlist` may contain multiple transitions. Such a `playlist` still specifies a unique behavior because `playlist` transitions are associative operations. Thus, multiple transitions placed in a single `playlist` describe the desired clip without any surprises.

4.5 Multitracks

Multitracks play producers in parallel. Like playlists, they employ transitions to composite their producers. Syntactically, `multitrack` is similar to `playlist`. The `multitrack` form consists of

```
#lang video

(multitrack
 (blank #f)
 (composite-transition 0 0 1/4 1/4)
 (clip "slides.mp4")
 (composite-transition 1/4 0 3/4 1)
 talk)

; where
(define talk ⟨···elided···⟩)
```


Figure 4 (continued): (d). Example using inlined transitions in a multitracks

a sequence of producers and creates a new `multitrack` producer. Again, transitions are included within the sequence to combine tracks; see figure 4c. This example uses `composite-transition`, which places one producer on top of the other. The four constants specify the coordinates of the top-left corner of the producer and the screen space that the top producer takes. Here, the producer following the transition appears in the top-left hand of the screen and takes up one quarter of the width and height of the screen.

Transitions within a `multitrack` are not associative; instead, `multitrack` interprets transitions in left to right order. Videos that require a different order can embed a `multitrack` inside of another one, because a `multitrack` is itself a producer. Using multiple transitions allow producers to appear side by side rather than just on top of each other. Expanding on the running example in figure 4c, figure 4d describes a conference video where the recording of the presenter goes in the top left while the slides go on the right. This example adds a `composite-transition` to the previous example and places the slides and the recording over a single blank producer. Blank producers are empty slides that act as either a background for a `multitrack` or a filler for a `playlist`. In this case, the blank producer is providing a background that the slides and camera feeds appear on.

4.6 Filters

Filters are similar to transitions, but they modify the behavior of only one producer. In other words, filters are functions from producers to producers. Among other effects, filters can remove the color from a clip or change a producer's aspect ratio. Conference recordings frequently capture audio and video on separate tracks. Before splicing the tracks together, a developer may add an envelope filter to provide a fade effect for audio.

A script may use function application notation to apply filters or attach them to producers with the `#:filters` keyword. Figure 4e shows an example of a filter being attached to an audio track that is itself composited with the video of the composited talk in figure 4d.

4.7 Properties and Dependent Clips

Producers use two types of properties to store information: implicit and explicit properties. Implicit properties are innate to clips, for example, length and dimensions. Explicit properties must be added by the program itself.

The properties API comes with two functions:

- `(set-property` *producer key value*`)` creates an explicit property. It returns a new producer with *key* associated with *value*.

```
#lang video

(multitrack
 composited-talk
 (clip "0000.wav"
  #:filters
    (list
      (envelope-filter 50 #:direction 'in)
      (envelope-filter 50 #:direction 'out))))

; where
(define composited-talk ⟨···elided···⟩)
```
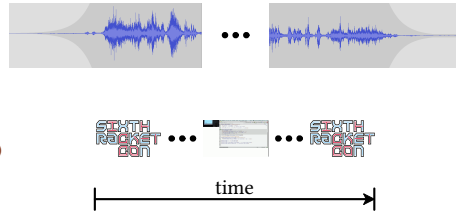

Figure 4 (continued): (e). Example of adding audio tracks

```
#lang video

(multitrack
 (blank #f)
 (composite-transition 0 0 1/4 1/4)
 (clip "slides.mp4")
 (composite-transition 1/4 0 3/4 1)
 talk
 (composite-transition 0 1/4 1/4 3/4)
 (image "logo.png"
  #:length (get-property talk 'length)))

; where
(define talk ⟨···elided···⟩)
```


Figure 4 (continued): (f). Example of adding a watermark

- (get-property *producer key*) returns the value associated with *key*. If the property is set both implicitly and explicitly, the explicit property has priority.

Explicit properties provide a protocol for communicating information from one clip to another. Implicit properties exist for the same purpose except that they store information that is implicitly associated with a producer, such as its length. For example, a conference video may have to come with a watermark that is the same length as the captured conference talk. A script that performs this operation can be found in figure 4f.

4.8 From Programs to Videos

A Video module may be incorporated into a program or understood as a stand-alone program. In the first case, another Video or Racket module may require the Video module and incorporate its export into its own code. In the second case, a user can hand the Video script to a renderer that either plays the video on a screen or saves it to a file.

By default, a Video module implicitly provides a producer. Any module that wants to use this implicitly created video imports it with the external-video form. For an example, consider this two-line module and what it denotes:

```
> (preview                        > (preview-video
    (external-video "talk.vid"))      "talk.vid")
```


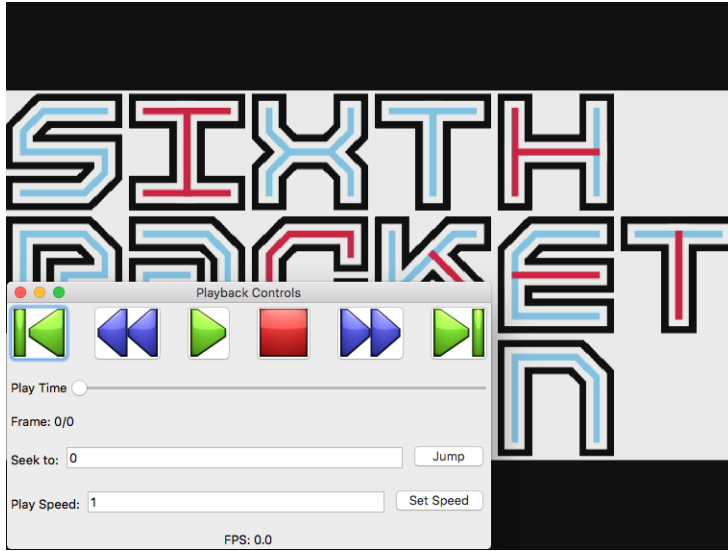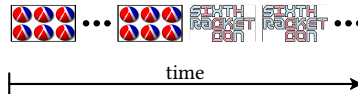
Figure 5: Previewing Videos

```
#lang video

(image "splash.png" #:length 100)
(external-video "talk.vid")
```


The module's first line sets up a splash screen, the second line incorporates the external Video module.

A renderer converts Video scripts to traditional videos. Having a dedicated rendering pass allows users to set various visual properties such as aspect ratio, frame rate, and even output format separately. The simplest renderers, dubbed render and preview, are functions that consume a producer and display it in a separate window. At DrRacket's REPL, developers can apply this function directly (left half of figure 5). While render just displays the video, preview adds playback controls, and even gives developers the ability to preview a video excerpt. Another renderer, called preview-video is a function that consumes a path to a Video script and plays it in a newly opened window (right half of figure 5). This functionality is available outside of the IDE so that non-programmers may view the videos, too.

4.9 Effectiveness

Leif Andersen, the first author, has been involved in the production of a video channel for RacketCon 2016. In Andersen's experience, creating Video and compositing the videos for that conference took less time than manually editing the videos for the previous year's conference (same number of talks, same nature of talks, etc).

```
01 #lang racket/base
02
03 (provide (rename-out [boo:set! set!])
04          (except-out (all-from-out racket/base) set!))
05
06 (define-syntax (boo:set! stx)
07   (syntax-parse stx
08     [(_ id val)
09      #'(begin (log-warning "Boo!!! Reassigning ~a" id)
10               (set! id val))]))
```

Figure 6: Logging assignment statements to expose ill-behaved functional programmers

5   THE FAST AND THE FURIOUS

Using the Racket ecosystem allows developers to implement languages quickly and easily. Further-more, these languages compose so that modules written in one language can easily interact with modules in another. Best of all, the implementation of a language may take advantage of other language technologies, too. The upshot here is that implementing Video is as simple as implementing video-specific pieces, while leaving the general-purpose bits to Racket.

Video's implementation makes heavy use of the Racket ecosystem. It consists of three major components and accounts for approximately 2,400 lines of code: a surface syntax, a run-time library, and a rendering layer. Of the code, about 90 lines are specific to the syntax and 350 lines define the video-specific primitives the language uses. The remaining lines are for the FFI and renderer. The video-specific primitives serve as adapters to imperative actions that work on Video's core data-types; they are implemented using standard functional programming techniques.

This section explains how a developer can implement a DSL in Racket (section 5.1), with Video serving as the running example (section 5.2). Not only is Video a Racket DSL, but part of the implementation of Video is implemented using additional DSLs created specifically for implementing Video (section 5.3).

5.1   Creating Languages, the Racket Way

Recall from section 2, creating Racket DSLs means removing unwanted features from some base language, adding new ones, and altering existing features to match the semantics of the new domain.

Adding and removing features is simple, because a language implementation is a module like any other module in Racket. Removing a feature is simply a matter of not re-providing it from the host language. See figure 6 for an example. Line 4 uses the except-out keyword to remove the definition of set! from racket/base, while all-from-out re-exports all remaining features from racket/base.

In addition to these operations, adding new features is simply a matter of defining the new features and exporting them. Developers do so in the same manner as a programmer who augments the functionality of a library via a wrapper module.

In contrast, modifying existing features requires slightly more work. Specifically the module must define a syntax transformation in terms of the old language and rename this transformation on export.

Let us illustrate this idea with a simple example. Many functional programmers do not like assignment statements; at Clojure conferences, programmers who admit to their use (via Java) are booed on stage.[9] So, imagine creating a language like Racket that logs a boo-warning of any use

---

[9]One of the authors witnessed this booing at a recent Clojure conference. Which has the largest, most successful community of commercial functional programmers.

```
01 #lang racket/base
02
03 (provide (rename-out #%lazy-app #%app)
04          (except-out (all-from-out racket/base) #%app))
05
06 (define-syntax (#%lazy-app stx)
07   (syntax-parse stx
08     [(_ rator rand ...)
09      #'(#%app (force rator) (lazy rand)  ...)]))
```

Figure 7: An essential element of Lazy Racket

of `set!`, Racket's variable assignment form. The `set!` provided by `racket/base` provides the functionality for assignment, while `log-warning` from the same language provides the logging functionality. All we have to do is define a new syntax transformer, say `boo:set!`, that logs its use and performs the assignment. From there, we need to rename `boo:set!` to `set!` in the `provide` specification. This renaming makes the revised `set!` functionality available to programmers who use the new and improved Functions-first variant of Racket. Figure 6 displays the complete solution.

Now recall that Racket's syntax system supports several interposition points that facilitate language creation: `#%app` for function application, `#%module-begin` for module boundaries, `#%datum` for literals, and so on. The purpose of these points is to allow language developers to alter the semantics of the relevant features without changing the surface syntax.

Figure 7 displays a small, illustrative example. Here, a language developer uses a strict `#%app` to construct a lazy form of function application. The `#%app` protocol works because the Racket compiler places the marker in front of every function application. Thus, language developers only need to implement their version of `#%app` in terms of an existing one.[10]

When a programmer uses this new language, the Racket syntax elaborator inserts `#%app` into all regular function applications. The elaborator resolves this reference to the imported version, written as $\#\%\text{app}^{\text{lazy}}$. From there, Racket redirects to `#%lazy-app`, which expands into $\#\%\text{app}^{\text{base}}$, Racket's actual application. Here is what the complete process looks like:

$$(\text{f a b c ...}) \quad \xrightarrow{\text{elaborates to}} \quad (\#\%\text{app}^{\text{lazy}} \text{ f a b c ...})$$
$$\xrightarrow{\text{elaborates to}} \quad (\#\%\text{lazy-app f a b c ...})$$
$$\xrightarrow{\text{elaborates to}} \quad (\#\%\text{app}^{\text{base}} \text{ (force f) (lazy a) (lazy b) (lazy c) ...})$$

The curious reader may wish to step through the elaboration via DrRacket's syntax debugger (Culpepper and Felleisen 2010).

5.2 The Essence of Video's Syntax

The implementation of Video's syntax uses two of Racket's interposition points: `#%module-begin` and `#%plain-lambda`. With these forms, language developers can simultaneously reuse Racket syntax and interpret it in a Video-appropriate manner.

Like `#%app`, `#%module-begin` is inserted at the start of every module and wraps the contents of that module. Hence, a re-interpretation may easily implement context-sensitive transformations. In the case of Video, the re-implementation of `#%module-begin` lifts definitions to the beginning of the module and collects the remaining expressions into a single playlist.

---

[10]Indeed, the Racket family of languages comes with the `lazy` language (Barzilay and Clements 2005), which uses exactly this interposition point to convert `racket` into an otherwise equivalent language with lazy semantics.

```
01 #lang racket/base
02
03 (provide (rename-out [#%video-module-begin module-begin])
04          (except-out #%module-begin racket/base))
05
06 (define-syntax (#%video-module-begin stx)
07   (syntax-parse stx
08     [(#%video-module-begin body ...)
09      #'(#%module-begin (video-begin vid (provide vid) () body ...))]))
10
11 (define-syntax (video-begin stx)
12   (syntax-parse stx
13     [(video-begin vid export (exprs ...))
14      #'(begin
15          (define vid (playlist exprs ...))
16          export)]
17     [(video-begin vid export (exprs ...) b1 body ...)
18      (syntax-parse (local-expand #'b1 'module ⟨···elided···⟩) ; <-- this-syntax
19        [(id*:id rest ...)
20         #:when (matches-one-of? #'id* (list #'provide #'define ⟨···elided···⟩))
21         #'(begin this-syntax (video-begin vid export (exprs ...) body ...))]
22        [else
23         #'(video-begin id export (exprs ... this-syntax) body ...)])]))
```

Figure 8: Video Compilation

Figure 8 shows the essence of Video's `#%module-begin` syntax transformer. It is written in Racket's `syntax-parse` language (Culpepper 2012), a vast improvement over the Scheme macro system (Bawden and Rees 1988; Clinger 1991; Dybvig et al. 1993; Kohlbecker et al. 1986; Kohlbecker and Wand 1987). As before, the transformer is defined with a different name, `#%video-module-begin` (line 6), and is renamed on export (line 3). The implementation of `#%video-module-begin` dispatches to `video-begin` (line 9), the workhorse of the module. This auxiliary syntax transformer consumes four pieces: an identifier (`vid`), a piece of code (`export`) formulated in terms of the identifier, a list of expressions (`e ...`), and the module's body, which is represented as a sequence of expressions (`body ...`). In the case of `#%video-module-begin`, the four pieces are `vid`, `(provide vid)`, `()`, and the module body.

The `video-begin` syntax transformer (lines 11–23) is responsible for lifting definitions to the beginning of the module body and accumulating expressions into a `playlist`. Its definition uses pattern matching again. Lines 13 and 17 specify the two pattern cases, one for when the module body is empty and another one that handles a non-empty sequence of body expressions:

- Once `video-begin` has traversed every piece of syntax (line 13), `exprs` contains all of the original module body's expressions. The generated output (lines 14–16) defines the given `vid` to stand for the list of expressions bundled as a playlist.
- In the second case, the transformer expands the first term up to the point where it can decide whether it is a definition (line 18). Next, the transformer uses `syntax-parse` to check whether the elaborated code is a syntax list (lines 19 and 22) with a recognized identifier in the first position (line 21), in particular, `define` and `provide`.
  - If so, the transformer lifts the first term out of the `video-begin` and recursively calls itself without the newly lifted expression (line 21).
  - Otherwise, it is an expression and gets collected into the `exprs` collection (line 23).

```
#lang video
(image "splash.png" ⟨···elided···⟩)
(conference-talk video ⟨···elided···⟩)
(define video ⟨···elided···⟩)
```
                    — elaborates to →
```
#lang video
(#%module-begin^video
 (image "splash.png" ⟨···elided···⟩)
 (conference-talk video ⟨···elided···⟩)
 (define video ⟨···elided···⟩))
```

                    — elaborates to →
```
#lang racket/base
(#%video-module-begin
 (image "splash.png" ⟨···elided···⟩)
 (conference-talk video ⟨···elided···⟩)
 (define video ⟨···elided···⟩))
```

                    — elaborates to →
```
#lang racket/base
(#%module-begin^base
 (define video ⟨···elided···⟩)
 (video-begin vid
   (image "splash.png" ⟨···elided···⟩)
   (conference-video video ⟨···elided···⟩)))
```

                    — elaborates to →
```
#lang racket/base
(#%module-begin^base
 (define video ⟨···elided···⟩)
 (provide vid)
 (define vid
   (playlist
     (image "splash.png" ⟨···elided···⟩)
     (conference-video video ⟨···elided···⟩))))
```

Figure 9: Compilation for a Video module

The astute reader may wonder about the generated `begin` blocks. As it turns out, Racket's `#%module-begin` flattens `begin` sequences at the module top-level into a simple sequence of terms.

The syntax transformer for function bodies also uses `video-begin`. Instead of handing over `(provide vid)`, the call in the function transformer merely passes along `vid`, because functions *return* the produced `playlist`, they do not export it.

Figure 9 shows the syntax elaboration of a module using the Video specific `#%module-begin` form. The elaborated module describes the running conference talk example. Here, `#%module-begin`^video is Video's variant, while `#%module-begin`^base is the one provided by `racket`.

5.3 Video, Behind the Scenes

Video relies on bindings to a C library to perform the rendering of video descriptions to files or streams. It exploits Racket's FFI language for this purpose (Barzilay and Orlovsky 2004). While Video uses the MLT Multimedia Framework, any suitable multimedia library will do.

As it turns out, the Racket doctrine actually applies to the step of adding bindings too. The task of importing bindings manually is highly repetitive and developers frequently turn to other tools or libraries to construct the FFI bindings for them. Using a DSL to create the bindings has two advantages over using a library for a similar task. First, a DSL separates the task of constructing safe FFI calls with the task of connecting to a specific multimedia library. Using an FFI, developers can connect to a library safely by specifying the contracts. Second, a DSL allows developers to create new binding forms that are relevant to the multimedia library. It turns out that, once again, the effort

of implementing this auxiliary DSL plus writing a single program in it is smaller than the effort of just writing down the FFI bindings directly. In other words, creating the DSL sufficiently reduces the overall effort so much that it offsets the startup cost, even though it is used *only once*.

The auxiliary DSL relies on two key forms: `define-mlt` and `define-constructor`. The first form uses the Racket FFI to import bindings from MLT. The second form, `define-constructor`, defines the core data types for Video and sets up a mapping to data that MLT understands. While we chose to use MLT to support Video, other rendering libraries can be used in its place. For example, early versions of Video used both GStreamer and MLT.

The `define-mlt` form is useful for hardening foreign functions. By using `define-mlt`, programmers must only specify a contract (Findler and Felleisen 2002) that describes the valid inputs and outputs. Consider `mlt_profile_init`, a function that initializes a C library. It takes a string and returns either a profile object or `NULL` if there is an error. Rather than having to manually check the input and output values, the FFI just states input and output types:

```
(define-mlt mlt-profile-init (fun string
                               -> [v : mlt-profile-pointer/null]
                               -> (null-error v)))
```

It errors if a bad input or output type passes through this interface.

The `define-constructor` form introduces both structures to represent video clips in Video and methods for converting these Video-level objects into structures that MLT understands. For its *first* purpose, it generalizes Racket's record-like `struct` definitions with an optional super-struct, additional fields, and their default values. For example, the following is the description of a Video-level producer:

```
(define-constructor producer service ([type #f] [source #f] [start -1] [end -1])
  (define producer* (mlt-factory-producer (current-profile) type source))
  (mlt-producer-set-in-and-out producer* start end)
  (register-mlt-close mlt-producer-close producer*))
```

This definition names the struct `producer`, specifies that it extends `service`, and adds four fields to those it inherits: `type`, `source`, `start`, and `end`.

For its *second* purpose, `define-constructor` introduces the body of a conversion function, which renderers know about. Here the function body consists of three lines. It has access to all of the structure's fields, as well as the parent stucture's fields. The renderer calls this conversion code at the point when it needs to operate over MLT objects rather than Video data structures.

6 THE BODYGUARD

What use is a programming language without a dependent type system? Lots of course, as Video shows. After all, Video is a scripting language, and most descriptions of a conference video are no longer than a few lines. No real programmer needs types for such programs. For some readers, however, the existence of an untyped language might be inconceivable, and we therefore whipped together a dependent type system in a single work day.[11]

After explaining the rationale for such a type system (section 6.1), we present the essential idea: type checking the length of producers, transitions (which are conceptually functions on producers), functions on transitions, and so on (section 6.2). Then we examine a few type-checking rules (section 6.3). Finally, we once again demonstrate the power of Racket's syntax system, which not

---

[11]We do not explore the metatheory of our type system since it is beyond the scope of this pearl.

only modifies the syntax of a base language but also adds a type system—with more or less the familiar notation found in papers on fancy type systems (section 6.4).

6.1 Video Data Types

Video programs primarily manipulate two types of data: producers and transitions. A typical program slices these values and then combines them. Not surprisingly, errors often involve manipulating producers and transitions of improper lengths. Such errors are particularly dangerous because they often manifest themselves at the FFI level during rendering. For example, the following piece of untyped code mistakenly tries to extract 15 frames from a producer that is only 10 frames long:

```
(cut-producer (color "green" #:length 10) #:start 0 #:end 15)
; EXCEPTION: given producer must have length >= end - start = 15
```

The following second example attempts to use producers that are too short for the specified transition:

```
(playlist (blank 10) (fade-transition #:length 15) (color "green" #:length 10))
; EXCEPTION: given producers must have length >= transition length 15
```

While old-fashioned scripting languages rely on dynamic checks to catch these bugs, modern scripting languages use a static type system instead (Gonzalez 2015; Meijer 2000). So does Typed Video.

6.2 Length Indexes

Typed Video adds a lightweight dependent type system to Video, where the types of producers and transitions are indexed by natural-number terms corresponding to their lengths. The rest of the type system resembles a simplified version of Dependent ML (Xi and Pfenning 1998).

Such a type system does not impose too much of a burden in our domain. Indeed, it works well in practice because Video programmers are already accustomed to specifying explicit length information in their programs. For example, the snippets from the preceding section produce static type error messages in Typed Video:

```
(cut-producer (color "green" #:length 10) #:start 0 #:end 15)
; TYPE ERR: cut-producer: expected (Producer 15), given (Producer 10)
(playlist (blank 10) (fade-transition #:length 15) (color "green" #:length 10))
; TYPE ERR: playlist: (fade-transition #:length 15) too long for producer (blank 10)
```

In general, the type system ensures that producer values do not flow into positions where their length is less than expected.

Typed Video also supports writing functions polymorphic in the lengths of videos. Recall the conference-talk example from figure 3. A function add-slides that just combines the video of a speaker with a video of slides from their presentation might look like this:

```
(define (add-slides {n} [vid : (Producer n)] [slides : (Producer n)] -> (Producer n))
  (multitrack vid ⟨···elided···⟩ slides ⟨···elided···⟩))
```

The function's type binds a universally-quantified type index variable n that ranges over natural numbers and uses it to specify that the lengths of the input and output producers must match up.

In addition, a programmer may specify side-conditions involving type index variables. Here is an add-bookend function, which adds an opening and ending sequence to a speaker's video:

```
; Add conference logos to the front and end of a video.
(define (add-bookend {n} [main-talk : (Producer n)] #:when (>= n 400) -> (Producer (+ n 600)))
  (playlist begin-clip
            (fade-transition #:length 200)
            main-talk
            (fade-transition #:length 200)
            end-clip)
  (define begin-clip (image "logo.png" #:length 500))
  (define end-clip (image "logo.png" #:length 500)))
```

The `add-bookend` function specifies, with a `#:when` keyword, that its input must be a producer of at least 400 frames because it uses two 200-frame transitions. The result type says that the output adds 600 frames to the input. The additional frames come from the added beginning and end segments, minus the transition frames.

Programmer-specified side-conditions may propagate to other functions. The `conference-talk` function from section 4 benefits from this propagation:

```
(define (conference-talk {n} [video  : (Producer n)] [slides : (Producer n)]
                             [audio  : (Producer n)] [offset : Int]
                             -> (Producer (+ n 600)))
  ⟨···elided···⟩
  (define p1 (add-slides video slides))
  (define p2 (add-bookend p1))
  ⟨···elided···⟩)
```

Even though `conference-talk` does not specify a side-condition, it inherits the `(>= n 400)` side-condition from `add-bookend`. Thus applying `conference-talk` to a video that is not provably longer than 400 frames results in a type error:

```
(conference-talk (blank 200) (blank 200) (blank 200) 0)
; TYPE ERROR: Failed condition (>= n 400), inferred n = 200
```

6.3   The Type System

While Typed Video utilizes only existing type system ideas, it is nevertheless instructive to inspect a few of its rules before we explain how to turn them into a language implementation.

As previously mentioned, Video programmers already specify explicit video lengths in their programs, and it thus is easy to lift this information to the type level. For example, figure 10 shows a few rules for creating and consuming producers. The COLOR-N rule lifts the specified length to the expression's type. In the absence of a length argument, as in the COLOR rule, the expression has type `Producer`, which is syntactic sugar for `(Producer ∞)`. The CLIP rule says that if the given file f on disk points to a video of length n,[12] then an expression `(clip f)` has type `(Producer n)`.

Typed Video utilizes a standard subtyping relation with a few additions, e.g., for `Producer`s:

$$\frac{m >= n}{(Producer\ m) <: (Producer\ n)}$$

Since Typed Video aims to prevent not-enough-frame errors, it is acceptable to supply a producer that is longer, but not shorter, than expected.

The PLAYLIST rule in figure 10 shows how producer lengths may be combined. Specifically, a `playlist` appends producers together and thus their lengths are summed. If playlists interleave transitions between producers, the lengths of the transitions are subtracted from the total because

---

[12]Obviously, the soundness of our type system is now contingent on the correctness of this system call.

COLOR-N
$$\frac{\Gamma \vdash \texttt{e : String}}{\Gamma \vdash \texttt{(color e \#:length n) : (Producer n)}}$$

COLOR
$$\frac{\Gamma \vdash \texttt{e : String}}{\Gamma \vdash \texttt{(color e) : Producer}}$$

CLIP
$$\frac{\Gamma \vdash \texttt{f : File} \qquad \texttt{|f| = n}}{\Gamma \vdash \texttt{(clip f) : (Producer n)}}$$

PLAYLIST
$$\frac{\forall \texttt{p/t:} \; \Gamma \vdash \texttt{p/t : (Producer n)} \text{ or } \Gamma \vdash \texttt{p/t : (Transition m)}}{\Gamma \vdash \texttt{(playlist p/t ...) : (Producer (- (+ n ...) (+ m ...)))}}$$
where each transition must occur between two producers

Figure 10: A few Producer type rules for Typed Video

each transition results in an overlapping of producers. A type error is signaled if the computed length of a producer is negative, as specified in the kinding rule for `Producers`:

$$\frac{\Gamma \vdash \texttt{n : Int} \text{ and } \texttt{n} \text{ is a well-formed index expression} \qquad \texttt{n >= 0}}{\Gamma \vdash \texttt{(Producer n) : *}}$$

In addition to requiring a non-negative constraint for `n`, this rule also restricts which terms may serve as type indices. Well-formed type index terms include only addition, subtraction, and a few Video primitives. If the `Producer` type constructor is applied to an unsupported term, the type defaults to a `Producer` of infinite length. Despite these restrictions, Typed Video works well in practice and can type check all our example programs, including those for the RacketCon 2016 video proceedings.

The LAM and APP rules in figure 11 roughly illustrate how Typed Video handles constraints. Rather than implement multiple passes, Typed Video interleaves constraint collection and type checking, solving the constraints in a "local" manner.

Specifically, type checking judgements additionally specify a set of constraints ϕ on the right-hand side, as in figure 11, corresponding to the constraints produced while type checking that expression. Constraints in Typed Video are restricted to linear arithmetic inequalities. The LAM rule in figure 11 shows that functions bind a type index variable `n` together with a procedural variable `x`, and must satisfy input constraints ϕ. These functions are assigned a universally quantified type that additionally includes constraints ϕ' collected while type checking the body of the function. When applying such a function, the APP rule first infers a concrete number `i` at which to instantiate the `n` index variable. It then uses `i` to simplify constraints ϕ via a partial-evaluation function $\vec{\mathcal{E}}$. Type checking fails if the resulting constraints are not satisfiable. Otherwise, the unsolved constraints are propagated. Modularly checking constraints at function applications in this manner enables the reporting of errors on a local basis.

6.4 Type Systems as Macros

The implementation of Typed Video relies on linguistic reuse to produce a full-fledged programming language without starting from scratch. Specifically, it reuses Racket's syntax system to implement type checking, following Chang et al.'s (2017) type-systems-as-macros technique. As a result, Typed Video is an extension to, rather than a reimplementation of, the untyped Video language.

Figure 12 shows the implementation of two rules: λ and function application. The `require` at the top of the figure imports and prefixes the identifiers from untyped Video, i.e., the syntactic extensions from section 5, which are used, unmodified, to construct the output of the type-checking pass.

LAM
$$\frac{\Gamma,\texttt{n:Int},\texttt{x}{:}\tau \vdash \texttt{e} \,:\, \tau'; \; \phi'}{\Gamma \vdash \lambda\texttt{n},\texttt{x}{:}\tau,\phi.\texttt{e} \,:\, \Pi\{\texttt{n}|\; \phi \wedge \phi'\}.\tau \to \tau'; \; \texttt{true}}$$

APP
$$\frac{\Gamma \vdash \texttt{f} \,:\, \Pi\{\texttt{n}|\; \phi\}.\tau \to \tau';\phi_1 \qquad \exists i.\Gamma \vdash e : \tau[n/i];\phi_2 \qquad \vec{\mathcal{E}}(\phi[n/i]) = \phi_3, \; \phi_3 \; \texttt{satisfiable}}{\Gamma \vdash \texttt{f} \; \texttt{e} \,:\, \tau'[n/i];\phi_1 \wedge \phi_2 \wedge \phi_3}$$

Figure 11: λ and function application type rules for Typed Video

```
#lang turnstile

01 (require (prefix-in untyped-video: video))
02 (provide λ #%app)
03
04 (define-syntax/typecheck (λ {n ...} ([x : τ] ... #:when φ) e) ≫
05   [(n ...) ([x ≫ x- : τ] ...) ⊢ e ≫ e- ⇒ τ_out]
06   #:with new-φs (get-captured-φs e-)
07   ----------------------------------------------------------
08   [⊢ (untyped-video:λ (x- ...) e-)
09      ⇒ (Π (n ...) #:when (and φ new-φs) (→ τ ... τ_out))])
10
11 (define-syntax/typecheck (#%app e_fn e_arg ...) ≫
12    [⊢ e_fn ≫ e_fn- ⇒ (Π Xs #:when φX (→ τ_inX ... τ_outX))]
13   #:with solved-τs (solve Xs (τ_inX ...) (e_arg ...))
14   #:with (τ_in ... τ_out φ) (inst solved-τs Xs (τ_inX ... τ_outX φX))
15   #:with φ* (type-eval φ)
16   #:fail-unless (not (false? φ*)) "failed side-condition φ at row:col ..."
17   #:unless (boolean? φ*) (add-φ φ*)
18    [⊢ e_arg ≫ e_arg- ⇐ τ_in] ...
19   ----------------------------------------------------------
20    [⊢ (untyped-video:#%app e_fn- e_arg- ...) ⇒ τ_out])
```

Figure 12: Type-checking via syntax transformers

We implement our type checker with Turnstile, a Racket DSL introduced by Chang et al. for creating typed DSLs. This DSL-generating-DSL uses a concise, bidirectional type-judgement-like syntax. In other words, the define-syntax/typecheck definitions in figure 12 resemble their specification counterparts in figure 11. The implementation rules define syntax transformers that incorporate type checking as part of syntax elaboration. Interleaving type checking and elaboration in this manner not only simplifies implementation of the type system, but it also enables creating true abstractions on top of the host language.

Next we briefly explain each line of the λ definition:

line 4: The transformer's input must match (λ {n ...} ([x : τ] ... #:when φ) e), a pattern that binds five pattern variables: n (the type index variables), x (the λ parameters), τ (the type annotations), ϕ (a side-condition), and e (the lambda body).

line 5: Since type checking is interleaved with syntax elaboration, Turnstile type judgements are elaboration judgements as well. Specifically, a judgement of the form [ctx ⊢ e ≫ e- ⇒ τ] is read "in context ctx, e elaborates to e- and has type τ."

Thus the λ transformer elaborates `e` to `e-`, simultaneously computing its type `τ_out`. This elaboration and type checking occurs in the context of the free variables. Instead of propagating a type environment, Turnstile reuses Racket's lexical scoping to implement the type environment. This re-use greatly enhances the compositionality of languages and reduces effort so that a programmer gets away with specifying only new environment bindings. Specifically, the first premise adds type index variables and lambda parameters to the type environment, where the latter may contain references to the former.

line 6: A `#:with` premise binds additional pattern variables. Here, elaborating the lambda body `e` may generate additional side-conditions, `new-φs`, that must be satisfied by the function's inputs. Rather than thread the side-conditions through every subexpression, as in figure 11, Typed Video's implementation utilizes a separate imperative interface for collecting constraints. Specifically, we use the `get-captured-φs` and `add-φ` functions to propagate side-conditions (the definition of these functions are not shown).

line 7: These dashes separate the premises from the conclusion.

line 8: The conclusion specifies the transformer's outputs: `(untyped-video:λ (x- ...) e-)`, an untyped term, along with its type `(Π (n ...) #:when (and φ new-φs) (→ τ ... τ_out))`. In Turnstile, types are represented using the same syntax structures as terms.

The second part of figure 12 presents Typed Video's function application rule implementation. It naturally interposes on Racket's function application hook, `#%app`, via untyped Video's function application definition, to add type checking. Here is a brief description of each line of `#%app`:

line 11: The input syntax is matched against pattern `(#%app e_fn e_arg ...)`, binding the function to `e_fn` and all arguments to `e_arg`.

line 12: The function elaborates to `e_fn-` with type `(Π Xs #:when φX (→ τ_inX ... τ_outX))`, which is universally quantified over type index variables `Xs` and has side-condition `φX`.

line 13: The transformer peforms local type inference, computing the concrete type indices `solved-τs` at which to instantiate the polymorphic function.

line 14: Next, the polymorphic function type is instantiated to concrete types `(τ_in ... τ_out)` and side-condition ϕ.

line 15: The transformer partially evaluates the concrete constraints in ϕ, producing $\phi\star$.

line 16: If $\phi\star$ is `false`, elaboration stops and the transformer reports an error. Though this paper truncates the error message details, this line demonstrates how our DSL creates true abstractions, reporting errors in terms of the surface language rather than the host language.

line 17: If $\phi\star$ is still an expression, propagate it using `add-φ`.

line 18: Check that the function arguments have the instantiated types. This premise uses the "check" left arrow. If a programmer does not explicitly implement a left-arrow version of a rule, Turnstile uses a standard subsumption rule by default.

line 20: The generated code consists of an untyped Video term along with its computed type.

The rest of the implementation of the typing rules is similar. For example, implementing Π is straightforward because Turnstile reuses Racket's knowledge of a program's binding structure to automatically handle naming. Further, the "as macros" approach facilitates implementation of rules
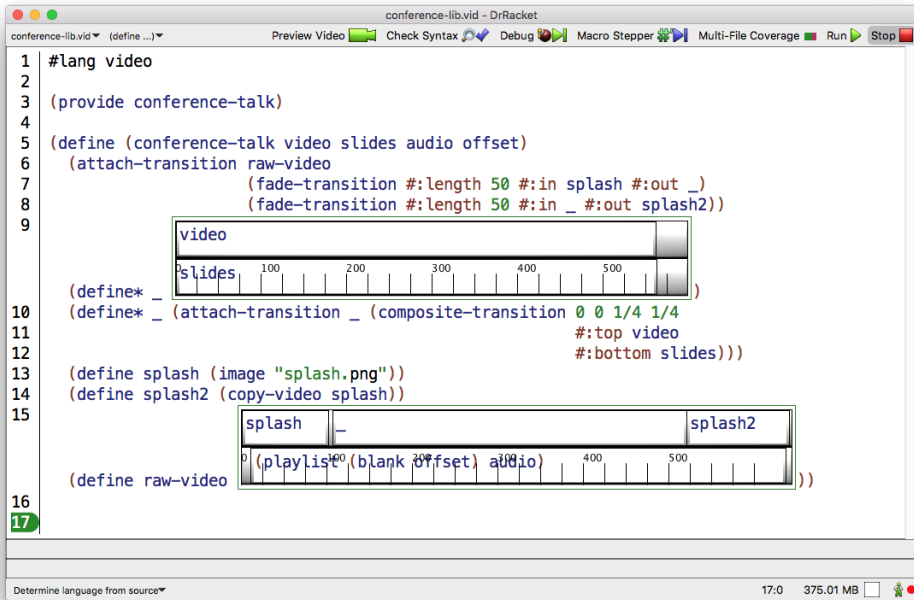
Figure 13: Mingling graphical NLVE widgets inside of Video scripts

for both terms and types, and thus the implementation of type-level computations also resembles the rules in figure 12.

7 TEENAGE MUTANT NINJA TURTLES -ω*

Some videos are best expressed with a graphical NLVE, and the DrRacket extension for Video therefore comes with embedded NLVE widgets. Unlike other NLVEs with scriptable APIs, the NLVE widget is actually part of the language. A developer may place an NLVE directly into a script. Best of all, the embedded NLVE may include code snippets, which in turn can contain yet another NLVE widget, etc. See figure 13 for a screenshot of such nesting.

A reader may wonder why one would want such a "turtles all the way down" approach to a Video IDE. Consider the actual scenario when a hardware failure during a talk prevented the capture of the speaker's screen. Fortunately, the speaker supplied a copy of their slide deck as a PDF document. While the captured video can still be recreated by using the slide deck, a decision has to be made concerning the duration of each slide. If a plain-text Video script were to use this method, it would inevitably contain a bunch of "magic numbers." Embedding NLVE widgets into the code explains these "magic numbers" to any future reader of the code and is thus a cleaner way to solve the problem. Figure 14 illustrates this point with a simplistic example. The module with magic numbers is on the left; the right part of the figure shows how an embedded NLVE explains the numbers directly. In both cases a developer must manually determine the screen time allocated to each slide. However, using the widget gives the author a graphical representation of the layout, thus speeding development

---

*We failed to find the Roman numeral rendering of ω on Google.

```
#lang video

(apply playlist
 (for/list ([slide (directory-list slides)]
            [time (in-list slide-times)])
   (image slide #:length (* 48 time))))

; where
(define slide-times
  (list 75 100 105 120 50 30 30 19 3
        10 50 15 33 250 42 20 65
        13 9 25 37 25 13 30 39 45))
```
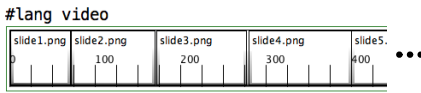
Figure 14: Slide reconstruction using magic numbers (left) and a NLVE widget (right)

time. Additionally, future authors can more easily tweak the times by dragging and resizing clips in the widget.

Graphical NLVEs are producers and first-class objects in Video. They can be bound to a variable, put in a playlist, supplied to a multitrack, and so on. Integrating the graphical and textual program in this manner allows users to edit videos in the style that is relevant for the task at hand. For example, the program in figure 13 shows an implementation of the `conference-talk` function from section 4, now implemented using NLVE widgets with embedded code snippets.

Traditional NLVEs have several advantages over DrRacket widgets.[13] For example, NLVEs such as Premier have a cursor that tracks a notion of "current time," with a preview window that shows a low resolution but real-time sample of the video at that position. This feature enables high-precision editing with quick feedback. As another example, traditional NLVEs do not have a notion of syntax error. While an NLVE project may not be correct, it is guaranteed to describe a Video. Even with widgets, Video programs can still have syntax errors as with other programming languages.

Video relies on the Racket ecosystem and the DrRacket environment to get REPL-style feedback needed for quick video editing. As described in section 4.8 the `preview` function shows a low-resolution (but real time) preview of the video being edited. This function starts the preview moments after it is called; it can additionally be called from both the Racket REPL and any shell environment. Developers using DrRacket can even use its cursor as an indicator for where to preview.

The Racket ecosystem makes it possible to add NLVE support with only a small amount of code. The editor itself plugs into the DrRacket programming environment (Findler et al. 2002). The editor itself is built on top of Racket's graphical framework (Flatt et al. 2010), which greatly facilitates such work. The entire editor is implemented in less than 800 lines of code. Of this, approximately 700 lines are for the graphical editor itself, and 50 are for the integration with Video. These lines are not counted in the 2,400 lines for Video's implementation. The code implementing these NLVE widgets is plain Racket code, and we therefore omit details of the implementation.

8   THE RELATED SUSPECTS

Two main principles guide the Racket Way[14] of creating DSLs (Flatt 2012): **reuse** and **abstraction**. As the old adage goes, "don't build languages from scratch" (Hudak 1996). Reusing existing

---

[13]We conjecture that these limitations are not fundamental to the concept of these widgets. However, testing that claim would require a significant amount of engineering compared to the rest of Video's implementation.

[14]Matthew Flatt. "The Racket Way". Strange Loop Conference, 2012. infoq.com/presentations/Racket

language components greatly helps DSL *creators*, who are unlikely to be programming language researchers. To aid reuse, Racket allows componentwise interposition of every core form.

Like all programmers, DSL *users* deserve proper abstractions, that is, constructs that do not expose the underlying, reused components. In other words, host language details should not leak into uses of the DSL, including in error messages. Racket's syntactic abstraction capabilities builds on the innovations of its Scheme and Lisp roots, but these predecessors never cared about writing true abstractions. At best, Lisp and Scheme programmers write low-level validation code that clutters the implementation; more commonly, validation is omitted, leaving Lisp and Scheme macros that resemble naive rewrite rules that do not distinguish the DSL from the host language. Racket helps the creation of robust linguistic abstractions with a declarative DSL for writing syntax transformers (Culpepper 2012). Instead of low-level validation code, programmers write high-level specifications, which is compiled to produce error messages in terms of the surface language.

Researchers have studied DSLs for a long time and have developed various alternative classifications for DSL construction strategies. In the functional programming language community, Gibbons and Wu (2014) distinguish DSLs along "deep" vs "shallow" lines. Deep embeddings implement a DSL's AST as algebraic datatype constructors. As a result, well-typed DSL terms are also well-typed host terms. While this makes effective reuse of the host language's type checker, this approach falls short on reuse because it requires implementing an evaluator from scratch. In addition, the deep embedding approach requires advanced type system features (Jones et al. 2006; Xi et al. 2003), which often produce obscure error messages that compromise the abstractions of the DSL. The "shallow" approach uses standard functions to build DSLs (Hudak 1996) and thus both reuses the evaluator of the host and leverages the abstraction power of functions and types to hide implementation details. This approach has limited flexibility to deviate from and change features of the host, however, because it does not consider syntactic abstraction nor interposition of language features. In other words, programmers cannot create *new* abstractions. The "finally tagless" technique (Carette et al. 2009) tries to improve on the shallow approach via a clever encoding of types, but it sacrifices reuse and abstraction altogether in the process.

In general, the deep vs shallow distinction seems more of an academic analysis and less focused on the pragmatics of DSL creation. Thus, it might not be so useful for programmers who are weighing tradeoffs and trying to solve domain-specific problems in the most effective manner. Some industrial developers (Fowler and Parsons 2010) view DSLs as either "internal" or "external." Internal DSLs focus on reuse and typically involve extending a host language, which is why they are also commonly called "embedded" DSLs. The Racket approach creates internal DSLs. External DSLs, as exemplified by the UNIX philosophy of "little languages" (Bentley 1986; Raymond 2003), are typically implemented from scratch and require more effort but as a result are not constrained by any particular host language.[15]

Racket may also be considered a "language workbench"[16] (Feltey et al. 2016), which describes a broad spectrum of tools and frameworks for creating DSLs, as well as a large community of researchers and developers. While workbenches tend to utilize GUI tools (Kelly et al. 1996), and commonly create external DSLs (Kats and Visser 2010), some more closely resemble Racket's "reuse and abstraction" approach (Erdweg et al. 2011), but with different design choices and thus trade-offs. See Erdweg et al. (2015)'s recent summary for a detailed survey of the characteristics and design choices in a large world of workbenches.

---

[15]Racket, like most languages, can of course be used to create external DSLs, but they do not represent "the Racket way".
[16]"Language Workbenches: The Killer-App for DSLs?", 2005, martinfowler.com/articles/languageWorkbench.html
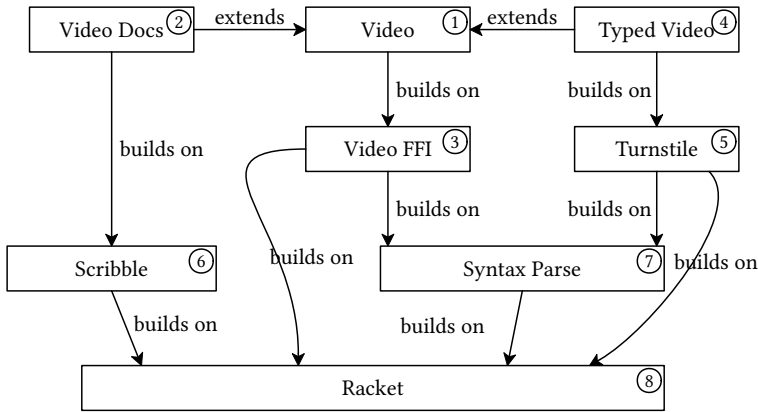
Figure 15: The tower of languages for Video

9 STAR TREK BEYOND

Imagine being Spock on the USS Enterprise. The ship's hyper-light-nano-pulsar sensor has discovered this paper. The discovery reveals a whole new, alternative future—Racket's approach to language-oriented programming (Dmitriev 2004; Ward 1994)—only a few light-years away. Now use extremely rational thinking to reason through the consequences of this insight.

Clearly, this alternative world encourages developers to build languages that are as close as possible to problem domains. In this context, "language" falls into the same class of concepts as library, framework, or module. Software systems will consist of a deep hierarchy of such languages. Some of these languages may sit at the surface of the system, helping domain experts formulate partial solutions to facets of the overall problem. Others may sit below the surface, in the interior of the hierarchy, because the implementation of DSLs also pose domain-specific problems. Each language will narrow the gap between the ordinary constructs of the underlying language (variables, loops, methods) and the concepts found in a problem domain (videos; syllabi; configurations). The hierarchy underneath Racket contains several dozen such languages, each dedicated to a special purpose, but all of them sitting within the core language.

Two critical factors enable this brave new world of language-oriented programming in the Racket ecosystem. The first one is that developing languages in Racket—*real languages*—is a process without friction. A language developer can edit a language implementation in one Emacs buffer, save the file, and immediately run a second Emacs buffer with code written in the language of the first one. Furthermore, the Racket syntax system—with interposition points, advance syntax-transformer facilities, syntax modules and so on—allows an extreme degree of linguistic re-use. Indeed, because of this potential of re-use, developers do not hesitate to create languages for a single use.

The second factor is that Racket acts as a common substrate. Eventually programs in these "little" languages are elaborated into core Racket modules. Developers can link these modules, creating multi-lingual systems within a single host language. While this kind of linking is not without problems, the advantages so far outweigh its disadvantages.

Racket is the host of this multi-lingual paradise, and Video is a poetic illustration of how this paradise works and what it promises. Figure 15 displays an organization diagram that summarizes the small language hierarchy underneath Video and its dependencies. The Video language itself exists because the domain of video editing calls out for a declarative scripting language. As always, Video closes the gap between the domain expert, who wishes to composite video clips

programmatically, and the scripting language, whose core provides nothing but functions and variables and list comprehension and similar linguistic features.

Similarly, a linguistic gap shows up for the implementation of Typed Video, an extension of Video. Here the domain expert is a type-system designer, who uses type-checking rules to design type systems. Once again, the gap between an ordinary functional language and this language of type-system designers is quite large. Hence our implementation uses another DSL to articulate our type system—as attachments to just those linguistic features for which we want types— as much as possible in the language of type-system designers.

The left side of the diagram shows yet another extension of Video, the Video Docs language, which Video programmers can use to create integrated documentation. Like Typed Video, it extends Video and rests on another domain-specific language, Scribble, which is a general-purpose mark-up language for writing API documentation (Flatt et al. 2009). While this extension is less complicated than the one on the right side, it is worth mentioning because documentation is all too often not understood as a domain in its own right.

The language gap also shows up with the implementation of Video. The language's run-time system demands extensive checking of values that flow into the C-level primitives. One way to translate the language from the C documentation into Racket is to add explicit checks inside Racket function definitions. Instead, we designed and implemented a DSL for dealing with just this insertion of checks for this specific library.

Finally, all of these languages make extensive use of yet another DSL, `syntax-parse`. This language closes the gap between Racket's constructs for defining syntax transformers and the language developers have in mind. For example, developers know that one particular element of syntactic form must be an identifier while another must be a definition. Furthermore, they know that the error messages must come from the form itself, not the result of elaborating an instance of the form. The language of `syntax-parse` provides all this and more, once again allowing developers to use the language they have in mind instead of just the underlying, raw core language.

No, Racket by no means solves all problems that come with language-oriented programming; see the note on linking above.[17] But, it sets itself apart from other approaches in the functional world, plus it already has numerous successes to show for. We hope that functional programmers of all stripes recognize the beauty of language-oriented-programming in general and Racket's approach in particular, and we invite them to translate the idea into their world.

Acknowledgments

REFERENCES

Eli Barzilay and John Clements. Laziness Without All the Hard Work. In *Functional and Declarative Programming in Education*, pp. 9–13, 2005.

Eli Barzilay and Dimitry Orlovsky. Foreign Interface for PLT Scheme. In *Scheme and Functional Programming*, pp. 63–74, 2004.

Alan Bawden and Jonathan Rees. Syntactic Closures. In *Lisp and Functional Programming*, pp. 86–95, 1988.

Jon Bentley. Little Languages. *Communications of the ACM* 29(8), pp. 711–21, 1986.

---

[17]Patterson and Ahmed's (2017) linking types might help here.

Dick Bulterman, Jack Jansen, Pablo Cesar, Sjoerd Mullender, Eric Hyche, Marisa DeMeglio, Julien Quint, Hiroshi Kawamura, Daniel Weck, Xabiel García Pañeda, David Melendi, Samuel Cruz-Lara, Marcin Hanclik, Daniel F. Zucker, and Thierry Michel. Synchronized Multimedia Integration Language. World Wide Web Consortium (W3C), 3.0, 2008. https://www.w3.org/TR/2008/REC-SMIL3-20081201/

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally Tagless, Partially Evaluated. *Journal of Functional Programming* 19(5), pp. 509–543, 2009.

Stephen Chang, Alex Knauth, and Ben Greenman. Type Systems as Macros. In *Principles of Programming Languages*, pp. 694–705, 2017.

William Clinger. Hygienic Macros Through Explicit Renaming. *Lisp Pointers* IV(4), pp. 25–28, 1991.

William R. Cook. Applescript. In *History of Programming Languages*, pp. 1-1–1-21, 2007.

Ryan Culpepper. Fortifying Macros. *Journal of Functional Programming* 22(4-5), pp. 439–476, 2012.

Ryan Culpepper and Matthias Felleisen. Debugging Hygienic Macros. *Science of Computer Programming* 75(7), pp. 496–515, 2010.

Ken Dancyger. *The Technique of Film and Video Editing: History, Theory, and Practice*. Fifth edition. Focal Press, 2010.

Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm. *JetBrains onBoard Electronic Magazine* 1(1), 2004.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. *Lisp and Symbolic Computation* 5(4), pp. 295–326, 1993.

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based Syntactic Language Extensibility. In *Object Oriented Programming Systems, Languages, and Applications*, pp. 391–406, 2011.

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Computer Languages, Systems and Structures* 44(Part A), pp. 24–47, 2015.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In *Summit on Advances in Programming Languages*, pp. 113–128, 2015.

Daniel Feltey, Spencer P. Florence, Tim Knutson, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Languages the Racket Way. In *Language Workbench Challenge*, 2016.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming* 12(2), pp. 159–182, 2002.

Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-order Functions. In *International Conference on Functional Programming*, pp. 48–59, 2002.

Matthew Flatt. Composable and Compilable Macros, You Want It When? In *International Conference on Functional Programming*, pp. 72–83, 2002.

Matthew Flatt. Creating Languages in Racket. *Communications of the ACM* 55(1), pp. 48–56, 2012.

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *International Conference on Functional Programming*, pp. 109–120, 2009.

Matthew Flatt, Robert Bruce Findler, and John Clements. GUI: Racket Graphics Toolkit. PLT Design Inc., PLT-TR-2010-3, 2010. https://racket-lang.org/tr3/

Martin Fowler and Rebecca Parsons. *Domain-specific Languages*. Addison-Wesley, 2010.

Jeremy Gibbons and Nicolas Wu. Folding Domain-specific Languages: Deep and Ahallow Embeddings (Functional Pearl). In *International Conference on Functional Programming*, pp. 339–347, 2014.

Gabriel Gonzalez. Light-weight and Type-safe Scripting with Haskell. In *Commercial Users of Functional Programming Tutorials*, 2015.

Paul Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Surveys* 28(4es), 1996.

Maxim Jago and Adobe Creative Team. *Adobe Premiere Pro CC Classroom in a Book*. Adobe Press, 2017.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple Unification-based Type Inference for GADTs. In *International Conference on Functional Programming*, pp. 50–61, 2006.

Lennart C.L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Object Oriented Programming Systems, Languages, and Applications*, pp. 444–463, 2010.

Steven Kelly, Kalle Lyytinen, and Matti Rossi. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In *Conference on Advances Information System Engineering*, pp. 1–21, 1996.

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

Eugene Kohlbecker and Mitchell Wand. Macro-by-example: Deriving Syntactic Transformations From Their Specifications. In *Principles of Programming Languages*, pp. 77–84, 1987.

Shriram Krishnamurthi. *Linguistic Reuse*. PhD dissertation, Rice University, 2001.

Erik Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming* 10(1), pp. 1–18, 2000.

Daniel Patterson and Amal Ahmed. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In *Summit on Advances in Programming Languages*, pp. 12-1–12-15, 2017.

Eric S. Raymond. *The Art of UNIX Programming*. First edition. Addison-Wesley, 2003.

Ton Roosendaal and Roland Hess. *The Essential Blender: Guide to 3D Creation with the Open Source Suite Blender*. No Starch Press, 2007.

Wim Taymans, Steve Baker, Andy Wingo, Rondald S. Bultje, and Kost Stefan. GStreamer Application Development Manual. 2013. https://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/manual.pdf

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages As Libraries. In *Programming Languages Design and Implementation*, pp. 132–141, 2011.

Martin P. Ward. Language Oriented Programming. *Software—Concepts and Tools* 15, pp. 147–161, 1994.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded Recursive Datatype Constructors. In *Principles of Programming Languages*, pp. 224–235, 2003.

Hongwei Xi and Frank Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *Programming Languages Design and Implementation*, pp. 249–257, 1998.