Practical compilation of fexprs using partial evaluation

Fexprs can performantly replace macros in purely-functional Lisp

NATHAN BRASWELL, Georgia Institute of Technology, United States
SHARJEEL KHAN, Georgia Institute of Technology, United States
SANTOSH PANDE, Georgia Institute of Technology, United States

Macros are a common part of Lisp languages, and one of their most lauded features. Much research has gone into making macros both safer and more powerful resulting in developments in multiple areas, including maintaining hygiene, and typed program staging [Rompf et al. 2013]. However, macros do suffer from various downsides, including being second-class. Particularly egregious for eager functional programming, they are unable to be passed to higher-order functions or freely composed.

Fexprs, as reformulated by John Shutt [Shutt 2010], provide a first-class and more powerful alternative to macros that meshes well with pure functional programming. Unfortunately, naive execution of fexprs is much slower than macros due to re-executing unoptimized operative combiner code at runtime that, in a macro-based language, would have been expanded and then optimized at compile time. To show that fexprs can be practical replacements for macros, we formulate a small purely functional fexpr based Lisp, *Kraken*, with an online partial evaluation and compilation framework that supports first-class, partially-static-data environments and can completely optimize away fexprs that are used and written in the style of macros. We show our partial evaluation and compilation framework produces code that is more than 70,000 times faster than naive interpretation due to the elimination of repeated work and exposure of static information enabling additional optimization. In addition, our *Kraken* compiler performs better compared to existing interpreted languages that support fexprs, including improving on NewLisp's [Mueller 2018] fexpr performance by 233x on one benchmark.

CCS Concepts: • **Software and its engineering → Functional languages**; **Compilers**.

Additional Key Words and Phrases: partial evaluation, Vau, F-exprs, fexprs, WebAssembly

1 INTRODUCTION

Lisps languages [Flatt and PLT 2010; Hickey 2008; Winston and Horn 1986] generally have two different abstraction methods: functions and macros. These two abstractions differ in their characteristics and semantics. Functions operate at run-time and always evaluate their parameters, while macros operate solely at expansion time and do not evaluate their parameters. Functions can sometimes (depending on the implementation) be used within macros but with restrictions. These restrictions split the language into two, and do not exhibit a few of the key tenets of functional programming - namely, being higher-order and supporting composition.

Macro systems generally attempt to be hygienic by either preventing or making it difficult to manipulate the code environment of the macro expansion [Clinger and Rees 1991]. However, manipulation of the code environment is often needed in special cases. The solution is typically various escape hatches that complicate the macro system. For these reasons, practical macro systems are often fairly complex and different from the language they are embedded within. In addition, because macros are expanded away, debuggability suffers as programmers must mentally expand macros to determine the cause of an error in code they did not explicitly write, but was generated by macro expansion.

Authors' addresses: Nathan Braswell, School of Computer Science, Georgia Institute of Technology, Atlanta, GA, United States, nathan.braswell@gtri.gatech.edu; Sharjeel Khan, School of Computer Science, Georgia Institute of Technology, Atlanta, GA, United States, smkhan@gatech.edu; Santosh Pande, School of Computer Science, Georgia Institute of Technology, Atlanta, GA, United States, santosh.pande@cc.gatech.edu.

A possible solution to these problems, fexprs, were created as a first-class and more powerful alternative in the 1960s and reformulated in 2010 [Shutt 2010]. Following Shutt's terminology, fexprs unify functions, macros, and built-in language forms into a single concept called a combiner. Where *lambda* introduces functions, *vau* introduces combiners. A combiner is able to evaluate its arguments 0 or more times in the calling environment, and additionally receives that environment as a parameter, allowing it to dynamically evaluate code in the same scope from which it was called. Combiners that evaluate their parameters 0 times are called operatives, while those that evaluate their parameters 1 or more times are called applicatives. Applicatives that evaluate their parameters one time and do not use the calling environment are simply normal functions. Operatives can subsume macros, built in special forms, and even control flow, like *if* and *lambda* itself.

A programming language based on combiners, using operatives instead of macros, provides a range of benefits:

- **Simpler Language**: While not having a separate macro system already makes for a simpler language, supporting operative combiners as first-class values further simplifies the definition of the language because there are no special forms. For instance, *if* and *vau* are just built-in operative combiners with support from the language definition or hypothetical interpreter loop. Other language features often implemented via macros like *let*, *and*, *or*, *cond* can of course be implemented as derived operatives, and going further, even *lambda* can be derived instead of built-in.
- **Simpler Mental Model and Better Debugging**: By unifying functions and macros (and formerly special forms) into one concept, debugging and mentally following macro-like operatives can be the same as for normal functions. When encountering an error, one can look at a stack trace and use a debugger to inspect stack variables to figure out what went wrong. The prototype debugger we developed in *Kraken* can show this information even when it would otherwise be optimized away by re-evaluating the side-effect-free code necessary to reconstruct the missing information. In a language based on macros one would need to print out different expansions of the macro and then try to figure out which one failed and why, without debugger support.
- **Greater Flexibility and Expressivity**: All combiners, including operatives, are first class and can thus be named, passed to higher-order combiners, composed, or put into data-structures. While many languages have first class functions, first class macros have not enjoyed the same success. Combiners are both in one. For example, in Scheme *and* is often a macro that expands to conditional control flow, meaning that it cannot be passed to a higher-order function such as *fold* without first being wrapped in a *lambda*, as seen in Listing 1. When *and* is an operative, it can be freely passed to higher-order combiners as is, as demonstrated in Listing 2.

```
> (fold and #t (list #t #t))
Exception: invalid syntax and
> (fold (lambda (a b) (and a b)) #t (list #t #t))
#t
```

Listing 1. Scheme's version of *and* example

```
> (foldl and true (array true true))
true
```

Listing 2. *Kraken*'s version of *and* example

The power of fexprs is reminiscent of advanced macro systems like that of Racket [Flatt and PLT 2010], which advocates for the definition of entirely new languages using its impressive,

but complex macro system. A language using fexprs can have similar expressive power, but with simpler semantics. Listing 3 shows a sample fexpr implementation of *let1* (a simple version of a let binding supporting only one variable) and *lambda* (using *let1*), demonstrating the early stages of bootstrapping a full language out of an extremely spartan base language where *vau* is the only abstraction operator. The details of how this works will be explained later in the paper. This is only to give the flavor of how features most languages would consider primitive and built-in can instead be defined inside the language itself:

```
((wrap (vau (let1)
; Definition of lambda
(let1 lambda (vau se (p b1)
      (wrap (eval (array vau p b1) se)))
  ; a simple function that multiplies its argument by two
  (lambda (n) (* n 2))
)
; Definition of let1
)) (vau de (s v b)
      (eval (array (array vau (array s) b) (eval v de))
            de)))
```

Listing 3. Fexpr implementation of *let1* and *lambda*

Despite all these benefits, naive execution of a pure language based on fexprs is exceedingly slow. *During its evaluation, the body of the of the called combiner is re-executed every time it is invoked, not just for function-like calls to applicatives, but for all macro-like calls to operatives too.* This re-execution happens for ever combiner call in the definition of the called operative as well. As a result, the re-executions will cause slowdowns likely to be exponential in the depth of the chain of definitions of macro-like operatives using other macro-like operatives in their definition. On the other hand, a macro in a macro system would have been executed once at expansion time and never during runtime. In the case of macros used in the definition of other macros, they will all be completely expanded before compilation. Because it is impossible to tell from syntax alone whether an argument to a combiner is evaluated or passed unevaluated (because it is impossible to tell if the call is going to be to an applicative or operative), code with fexprs is difficult to compile and optimize. As a result, typical implementations of fexprs leave them unoptimized and entirely interpreted, augmenting the slowness issue.

Some languages like NewLisp and PicoLisp [Burger 2013; Mueller 2018] that implemented some part of fexprs generally limit the number of layered fexprs to avoid compounding slowdowns. They chose to implement many combiners directly in the interpreter for speed instead of writing them as derived fexprs. Any code using fexprs in these languages still incur performance penalties or crash after hitting a limit. Other works [Kearsley [n.d.]; Shutt 2010] have described fexpr languages and shown their usefulness but the few implementations based on them have been extremely slow. As a result, a practical (fast) language based primarily on fexprs does not exist.

One solution to the performance issues of fexprs is partial evaluation. Partial evaluators have been developed for numerous languages [Alpuente et al. 1998; Andersen 1992; Elphick et al. 2003; Komorowski 1982; Lloyd and Shepherdson 1991; Meyer 1991] and for different domains [Berlin 1990; Berlin and Weise 1990; Futamura 1971]. The purpose of partial evaluation is *to specialize code based on values known at partial-evaluation time in order to do less work at execution time*, hopefully improving performance [Jones 1996; Würthinger et al. 2017]. Partial evaluation can be broken into online and offline techniques with [Jones 1996] explaining the differences the best. In addition, John Shutt [Shutt 2010] also suggested partial evaluation might be a solution to fexpr

performance but did not provide any specific details or implementation. Despite well-known online and offline partial evaluation techniques [Consel and Danvy 1993; Jones 1996], there is still no partial evaluation solution for fexprs, and so fexprs remain slow in all existing implementations.

In this work, we propose the first practical (fast) purely functional Lisp language based entirely on fexprs, *Kraken*. *Kraken* utilizes an online partial evaluator specifically created to evaluate away all calls to macro-esque operatives paired with a compiler backend that takes advantage of the static information exposed by the partial evaluator to produce reasonably performant WebAssembly binaries. Using it, we show that *a functional Lisp based on fexprs can be approximately as efficient and at least as expressive as one based on macros.*

***Kraken* Language** (§3): Our first contribution is our purely functional fexpr-based Lisp called *Kraken*. The language is based on John Shutt's definition of pure Vau calculus [Shutt 2010] (using fexprs and based on *vau* instead of *lambda*) but augmented with explicit primitive data and operations to demonstrate a more practical language.

Partial Evaluation and Compiler Optimizations (§4 - §5): Our second contribution is the tailored partial evaluation algorithm and compiler optimizations that enable an fexpr-based functional language to have competitive performance. Partial evaluation will evaluate away any user-defined operatives that behave like macros, and the compiler will inline any primitive operatives (if, etc), leaving only the non-macro applicatives for runtime. Compiler optimizations (type-inference-informed primitive inlining, single-use-closure-inlining, and lazy-environment-creation) remove many of the remaining inefficiencies in our language. This combined partial-evaluation and compilation technique shows that macro-esque operatives can perform as well as macros due to being compiled away statically, similar to how macros would be expanded away.

Evaluation of the Language (§6): Our final contribution is the evaluation of the language and compiler on a few benchmarks to demonstrate its practicality. Firstly, we show partial evaluation with compiler optimizations improves the runtime performance by over **70,000x** compared to our baseline interpreter. Secondly, we show *Kraken*'s optimized and compiled code performs significantly better than NewLisp's interpreted fexpr [Mueller 2018] implementation, by **233x** in one benchmark. Lastly, we compare our runtime performance against NewLisp's macro implementation and end up faster than it as well.

The paper begins by discussing our general compilation flow in Section 2 before defining a simplified version of our *Kraken* language in Section 3. Section 4 contains our core contribution, the partial evaluation algorithm focused on evaluating away macro-esque operatives. This is followed by a discussion of the major optimizations that work hand-in-hand with the partial evaluation algorithm in Section 5. Section 6 presents our benchmarks demonstrating the dramatic speedups our algorithm achieves over the naive interpretation of fexprs before Section 7 lists related work and Section 8 concludes.

2 GENERAL COMPILATION FRAMEWORK

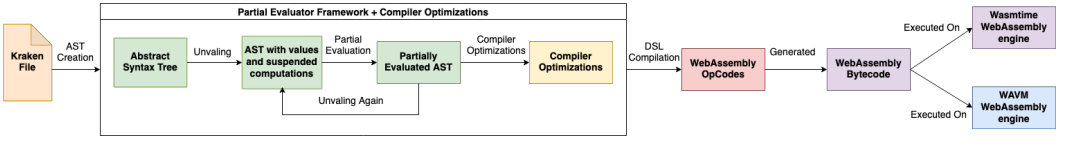


Fig. 1. The Compilation Framework (including partial evaluation and compiler optimizations) compiles *Kraken* code into a WebAssembly binary that (for our benchmarks) executes on either the WAVM WebAssembly engine or the Wasmtime WebAssembly engine

The *Kraken* language gets compiled into WebAssembly before it is executed on either the Wasmtime WebAssembly engine or WAVM WebAssembly engine as seen in Fig. 1. Initially, the *Kraken* code is parsed to create an abstract syntax tree (AST). The AST gets 'marked', adding bookkeeping information needed for partial evaluation to the AST, and is then 'unvaled', annotating the sections of the AST which will be evaluated as suspended computations. Once the AST is unvaled, the AST is partially evaluated by checking each suspended computation to see if it can be evaluated based on the current (generally partially-static-data) environment. While the compiler backend walks the AST and emits WebAssembly bytecode, it performs optimizations like type-inference-based primitive inlining, single-use-closure inlining, lazy-environment-creation, and basic tail call elimination to further improve code quality for better performance. This bytecode can then be executed on any compliant WebAssembly engine.

3 *KRAKEN* LANGUAGE

As mentioned before, the proposed *Kraken* language follows quite closely to John Shutt's definition of the pure Vau calculus ([Shutt 2010]) augmented with explicit primitive data types and operations to create a more practical language. Basing our language on the pure Vau calculus instead of the more complex calculi that form the basis of John Shutt's Kernel language makes the partial evaluation and optimization simpler and more feasible. The main missing features from his complex calculi are mutation and continuations. In order to keep our formalization understandable, we present a calculus simplified further from our real language that eliminates boolean values, varaidac parameters, and prunes down the builtin primitive combiners.

We start defining the *Kraken* language by presenting its syntax, contexts, and small-step semantics before expounding upon the difference between its surface and internal syntax and semantics.

3.1 Syntax

$$
\begin{array}{rcll}
n & \in & \mathbb{N} & \text{(Integers)} \\
s & \in & \textit{Symbols} & \\
o & \in & \{\langle 1\ \textbf{eval}\rangle, \langle 0\ \textbf{vau}\rangle, \langle 1\ \textbf{wrap}\rangle, \langle 1\ \textbf{unwrap}\rangle, & \\
 & & \langle 0\ \textbf{if0}\rangle, \langle 0\ \textbf{vif0}\rangle, \langle 1\ \textbf{int-to-symbol}\rangle, & \\
 & & \langle 1\ \textbf{symbol?}\rangle, \langle 1\ \textbf{int?}\rangle, \langle 1\ \textbf{combiner?}\rangle, \langle 1\ \textbf{env?}\rangle, & \\
 & & \langle 1\ \textbf{array?}\rangle, \langle 1\ \textbf{len}\rangle, \langle 1\ \textbf{idx}\rangle, \langle 1\ \textbf{concat}\rangle, & \\
 & & \langle 1\ \texttt{+}\rangle, \langle 1\ \texttt{<=}\rangle\} & \text{(Primitive Operations)} \\
E & := & \langle\langle (s \leftarrow T) \ldots \mid, E\rangle\rangle \mid \langle\langle (s \leftarrow T) \ldots \mid s' \leftarrow E, E\rangle\rangle & \text{(Environments)} \\
A & := & (T \ldots) & \text{(Arrays)} \\
C & := & \langle \textbf{comb}\ n\ s'\ E\ (s \ldots)\ T\rangle & \text{(Combiners)} \\
S & := & n \mid o \mid E \mid C & \text{(Self evaluating terms)} \\
V & := & S \mid s \mid A & \text{(Values)} \\
T & := & V \mid AT & \text{(Terms)} \\
AT & := & [\text{eval}\ T\ E] \mid [\text{combine}\ T\ (T \ldots)\ E] & \text{(Active terms)}
\end{array}
$$

Fig. 2. Syntax of the Base Language

The surface syntax of the language (Fig. 2) consists of arrays of symbols and integers. The internal syntax includes both primitive and derived combiners, environments, and actively-executing terms. It is this split between surface syntax and internal syntax that permits a non-trivial theory [Shutt 2010].

3.2 Contexts

$$\begin{aligned}\mathcal{E} \quad := \quad & \square \mid [\text{combine } \mathcal{E}\ (T \ldots)\ E] \mid [\text{combine } T\ (\mathcal{E}, T \ldots)\ E] \\ & \mid [\text{combine } T\ (T \ldots, \mathcal{E}, T \ldots)\ E] \mid [\text{combine } T\ (T \ldots, \mathcal{E})\ E]\end{aligned}$$

Fig. 3. Contexts of the Base Language

The contexts for evaluation in our base calculus (Fig. 3) define the holes where current evaluation must take place - namely, either evaluating the head of an array to find the combiner to call, or evaluating parameters before calling combiner.

3.3 Small-Step Semantics

$$\begin{aligned}\mathcal{E}[E] &\rightarrow \mathcal{E}[E'] \ (\text{if } E \rightarrow E') \\ [\text{eval } S\ E] &\rightarrow S \\ [\text{eval } s\ E] &\rightarrow \mathit{lookup}(s, E) \\ [\text{eval } (T_1\ T_2 \ldots)\ E] &\rightarrow [\text{combine } [\text{eval } T_1\ E]\ (T_2 \ldots)\ E] \\ \\ [\text{combine } \langle \textbf{comb } (S\ n)\ s'\ E'\ (s \ldots)\ Tb \rangle\ (V \ldots)\ E] &\rightarrow [\text{combine } \langle \textbf{comb } n\ s'\ E'\ s\ Tb \rangle \\ &\qquad [\text{eval } V\ E] \ldots\ E] \\ [\text{combine } \langle \textbf{comb } 0\ s'\ E'\ (s \ldots)\ Tb \rangle\ (V \ldots)\ E] &\rightarrow [\text{eval } Tb\ \langle\langle (s \leftarrow V) \ldots \mid s' \leftarrow E,\ E' \rangle\rangle] \\ \\ [\text{combine } \langle (S\ n)\ \textbf{o} \rangle\ (V \ldots)\ E] &\rightarrow [\text{combine } \langle n\ \textbf{o} \rangle\ ([\text{eval } V\ E] \ldots)\ E]\end{aligned}$$

Fig. 4. Semantics of the Base Language

Our small-step semantics (Fig. 4) provides the evaluation steps for calls, symbols, and values. *Kraken* evaluates self-evaluating values to themselves and symbols to the lookup of the symbol within the current environment. The most interesting case is calls because it requires the first element of the array to be evaluated first resulting in a combiner.

How the call proceeds after obtaining the combiner to be called depends on the "wrap level" and type of combiner. This "wrap level" designates how many times the arguments to the combiner should be evaluated, and the semantics encodes evaluating the arguments that many times, enforcing that a combiner will have a "wrap level" of 0 (decremented every time the parameters are evaluated) before being called. As we mentioned earlier, a combiner with a wrap level of 1 is an applicative, equivalent to functions in other languages. The combiner can be either primitive (built into the language) or derived (created by the user with a call to *vau*). A derived combiner's body is evaluated in a new environment that maps the parameter symbols to the actual arguments, including mapping a special symbol to the current dynamic call environment, and chains up to the existing static environment stored within the derived combiner. The primitive combiners are explained more in the next section.

3.4 Small-Step Semantics (selected primitives)

The semantics of the most important primitive combiners are given in Fg. 5, but are additionally described here in English for clarity:

- $\langle 0\ \textbf{eval} \rangle$: evaluates its argument in the given environment.

$$
\begin{array}{rcl}
[\text{combine } \langle 0\ \textbf{eval}\rangle\ (V\ E')\ E] & \rightarrow & [\text{eval } V\ E'] \\
[\text{combine } \langle 0\ \textbf{vau}\rangle\ (s'\ (s\ldots)\ V)\ E] & \rightarrow & \langle \textbf{comb}\ 0\ s'\ E\ (s\ldots)\ V\rangle \\
[\text{combine } \langle 0\ \textbf{wrap}\rangle\ \langle \textbf{comb}\ 0\ s'\ E'\ (s\ldots)\ V\rangle\ E] & \rightarrow & \langle \textbf{comb}\ 1\ s'\ E'\ (s\ldots)\ V\rangle \\
[\text{combine } \langle 1\ \textbf{unwrap}\rangle\ \langle \textbf{comb}\ 1\ s'\ E'\ (s\ldots)\ V\rangle\ E] & \rightarrow & \langle \textbf{comb}\ 0\ s'\ E'\ (s\ldots)\ V\rangle \\
[\text{combine } \langle 0\ \textbf{if0}\rangle\ (V_c\ V_t\ V_e)\ E] & \rightarrow & [\text{combine } \langle 0\ \textbf{vif0}\rangle \\
 & & ([\text{eval } V_c\ E]\ V_t\ V_e)\ E] \\
[\text{combine } \langle 0\ \textbf{vif0}\rangle\ (0\ V_t\ V_e)\ E] & \rightarrow & [\text{eval } V_t\ E] \\
[\text{combine } \langle 0\ \textbf{vif0}\rangle\ (n\ V_t\ V_e)\ E] & \rightarrow & [\text{eval } V_e\ E]\ (\text{n != 0}) \\
[\text{combine } \langle 0\ \textbf{int-to-symbol}\rangle\ (n)\ E] & \rightarrow & 'sn\ (\text{symbol made out of the number n}) \\
[\text{combine } \langle 0\ \textbf{array}\rangle\ (V\ldots)\ E] & \rightarrow & (V\ldots)
\end{array}
$$

Fig. 5. Semantics of Base Language Primitives

- $\langle 0\ \textbf{vau}\rangle$: creates a new combiner and is analogous to lambda in other languages, but with a "wrap level" of 0, meaning the created combiner does not evaluate its arguments.
- $\langle 0\ \textbf{wrap}\rangle$: increments the wrap level of its argument. Specifically, we are "wrapping" a "wrap level" n combiner (possibly "wrap level" 0, created by *vau*) to create a "wrap level" n+1 combiner. A wrap level 1 combiner is analogous to regular functions in other languages.
- $\langle 0\ \textbf{unwrap}\rangle$: decrements the "wrap level" of the passed combiner, the inverse of *wrap*.
- $\langle 0\ \textbf{if}\rangle$: evaluates only its condition and converts to the $\langle 0\ \textbf{vif}\rangle$ primitive for the next step. It cannot evaluate both branches due to the risk of non-termination.
- $\langle 0\ \textbf{vif}\rangle$: evaluates and returns one of the two branches based on if the condition is non-zero.
- $\langle 0\ \textbf{int-to-symbol}\rangle$: creates a symbol out of an integer.
- $\langle 0\ \textbf{array}\rangle$: returns an array made out of its parameter list.

The less interesting primitives we just describe here:

- [combine $\langle 0\ \textbf{type-test?}\rangle$ (A) E]: *array?*, *comb?*, *int?*, and *symbol?*, each return 0 if the single argument is of that type, otherwise they return 1.
- [combine $\langle 0\ \textbf{len}\rangle$ (A) E]: returns the length of the single array argument.
- [combine $\langle 0\ \textbf{idx}\rangle$ $(A\ n)$ E]: returns the nth item array A.
- [combine $\langle 0\ \textbf{concat}\rangle$ $(A\ B)$ E]: combines both array arguments into a single concatenated array.
- [combine $\langle 0\ \textbf{+}\rangle$ $(A\ A)$ E]: adds its arguments
- [combine $\langle 0\ \textbf{<=}\rangle$ $(A\ A)$ E]: returns 0 if its arguments are in increasing order, and 1 otherwise.

3.5 Base Language Summary

This base calculus defined above is not only capable of normal lambda-calculus computations with primitives and derived user applicatives, but also supports a superset of macro-like behaviors via its support for operatives. All of the advantages listed in the introduction apply to this calculus, as do the performance drawbacks, at least if implemented naively. Our partial evaluation and compilation framework will demonstrate how to compile this base language into reasonably performant binaries (WebAssembly bytecode, for our prototype).

4 PARTIAL EVALUATION

Partial Evaluation dates back at least to Lombardi's partial evaluator for Lisp [Lombardi 1964]. In this work, we devise a partial evaluator specifically focused on partially evaluating away fexprs that behave like macros that we call **macro-like operatives**. By partially evaluating and inlining

calls to these fexprs in a way analogous to macro expansion, we show a language based on fexprs instead of macros can be approximately as efficient and at least as powerful. Our partial evaluation algorithm can eliminate all uses of macro-like operatives, where macro-like means the uses are static, and the combiner definition looks something like Listing 4.

```
(let (helper (lambda (...) <generate code like macro>))
     (vau dynamic_env (parameters) (eval (helper parameters) dynamic_env)))
```

Listing 4. A Macro-like Combiner Definition

In these cases, the combiner takes in its arguments unevaluated (as it is an operative), passes them to a helper function that generates new code based on the passed code parameters, and then immediately evaluates the code generated by the macro-like function. Any combiner written following this template will be partially evaluated away by our algorithm. Our partial evaluator and compiler can handle more cases than just this exact "macro-like" template, but we focus on this case. By showing that it is possible to port anything written as a macro (which will be expanded away) to an analogous operative combiner (that will be partially evaluated away), we establish the practicality of using fexprs instead of macros as the fundamental building blocks of a purely functional Lisp-like language. When creating a new language, one loses neither expressivity nor significant performance by choosing fexprs as the base construct over the combination of functions and macros. Any use of fexprs that is not partially evaluated away is something that *could not be expressed via macros*, and thus paying a performance penalty in these cases is not onerous.

We implemented an online partial evaluation strategy for *Kraken*. Based on [Jones 1996]'s description, offline partial evaluation relies on a Binding Time Analysis to have been done so it can determine which pieces of code are dependent upon dynamic runtime values of the system and which can be computed statically. On the other hand, online partial evaluation actually executes the program using real values if they are statically known, and symbolic values otherwise. Computations that use only static values result in more statically known values, whereas the computations involving dynamic values generate residual code to be included in the final program.

In our case, it is impossible to even determine what symbols will be used as variables without performing at least some execution, much less determining if those variables will contain static or dynamic values, so online partial evaluation is the solution for us. By being an online partial evaluation algorithm, we do not mean it is happening at program run-time rather it runs at compile-time. In other words, "online" only refers to when the binding analysis is done relative to partial evaluation, not when the partial evaluation itself is done.

Sophisticated partial evaluation algorithms can handle partially-static data structures, in which only some of the data and structure is known [Sperber and Thiemann 1996]. A classic example of partially-static data is a Lisp cons cell, one side of which contains a static integer and the other residual code. Some advanced partial evaluators maintain additional information about values and residual code, such as its type [Ruf 1993].

The following is the key difficulty in compiling away macro-esque operatives - the call site environment must be reified and passed to the operative, and that reified environment will be a partially-static data structure. That is, each frame in the environment is either static data, mapping symbols to static values, or it is a static description of dynamic data, containing symbols and the ID of the combiner that created it (by being called). Either of these types of frames may then chain upwards to a parent frame, which may be of either type as well. Correctly handling these partially-static-data environments while ensuring that all macro-esque operatives are partially evaluated away while preventing both non-termination and exponential runtime was the key challenge in this work. A thorny complication is that that a combiner definition may have to be partially evaluated multiple times in environments with different amounts of static data before it has

enough information to reduce as far as it should according to our criteria of eliminating all statically known operative calls. This means if partial evaluation for a form fails, we can't turn it into residual code right away, as it might be evaluated again later. If this is done naively, it is very easy to incur exponential runtime as at every call first the combiner is evaluated, then the parameters, followed by the combiner again. Since this compounds at every call inside a combiner (and compounds via environments containing combiners containing environments without memorization or similar), a mitigating technique is needed. Our solution to this issue is the needed-for-progress-IDs system, which will be explained below.

As bits of the partial evaluation, especially as relating to calls, get complex, we broke down the algorithm into sections to make it easier to understand. In each section, we provide examples along the way, and at the end provide some final examples to show the process. Our partial evaluation roadmap:

- **Section 4.1**: Partial Evaluation Contexts, Marking Syntax, and Unval Relation
- **Section 4.2**: Small Step Semantics
- **Section 4.3**: Helper Relations
- **Section 4.4**: Partial-Eval Versions of Primitives
- **Section 4.5**: Total Effect of Partial Evaluation
- **Section 4.6**: Examples

4.1 Evaluation Contexts, Marking Syntax, and Unval Relation

For partial evaluation, we added a new active term, *under*, replace *eval* with *peval* (partial eval), and add new values to *combine*, as shown in Fig. 6.

$$
\begin{array}{rcll}
AT & := & [\text{peval } T\ /i_x/E\ ES\ FS] & \\
 & | & [\text{under } T\ (T\ldots)\ /i_x/E\ ES\ FS] & \\
 & | & [\text{combine } T\ (T\ldots)\ E] & \text{(Active terms)}
\end{array}
$$

Fig. 6. New Terms for Partial Evaluation

We track two critical types of information throughout the partial evaluation process: IDs of environments that contain values needed for progress, and a set of the currently evaluating terms. This allows us to massively prune the execution space and guide the path of partial evaluation.

This bookkeeping information is added via a "mark" pass. Partial evaluation will operate on this "mark"'d representation, as will the compiler. The extra information that bookkeeping adds for each type of form is as follows:

- **Combiner**: A unique ID (i) that indicates environments created by calling this combiner
  Example: $/i/\langle\textbf{comb}\ n\ s'\ /i'_{f'}/E'\ (s\ldots)\ Tb\rangle$
- **Env**: A unique ID (i_r or i_f) matching this environment to the combiner whose call created it. The r or f subscript indicating whether the environment is "fake", a static description of the dynamic environment which maps symbols to placeholder values, or "real", a fully static map from symbols to (almost entirely) static values. "Almost entirely" because a "real" environment can map a symbol to a "fake" environment, as happens when partially evaluating a call to an operative that takes in its call site environment.
  Example: $/i_r/\langle\langle(s \leftarrow V)\ldots\ |s' \leftarrow /i_x/E,\ /i'_{x'}/E'\rangle\rangle$
  Example: $/i_f/\langle\langle(s \leftarrow V)\ldots\ |s' \leftarrow /i_x/E,\ /i'_{x'}/E'\rangle\rangle$
- **Symbol**: ID of environment that the symbol will be resolved in, if applicable. True means the symbol has yet to be partially evaluated, so the ID of the environment will resolve to

unknown. $\emptyset$ means the symbol is a value instead of a suspended lookup.
Example: /true/s
Example: /7/s
Example: /$\emptyset$/s

- **Array**: Arrays can be marked by one of three options: *val*, *freshCall*, or *attemptedCall*. *val* indicates the array is a value. *freshCall* indicates the array is a call that hasn't been partially evaluated yet. *attemptedCall* indicates the suspended call was partially evaluated, but couldn't proceed. *attemptedCall* contains two values - the ID of the dynamic calling environment if the called combiner needs it (otherwise $\emptyset$) and/or the (form, environment) pair that caught and prevented infinite recursion.
Example: /val/(V_1 V_2)
Example: /freshCall/(V_1 V_2)
Example: /atmdCall $\emptyset$ $\emptyset$/(V_1 V_2)

In our implementation, we store additional data for efficiency, such as all for-progress-IDs found inside an array, but we will define them in Appendix A as stand-alone functions for ease of presentation. Our implementation essentially pre-computes and stores the results of these functions as additional bookmarking data, reducing exponential lookups to constant factors. This could also be achieved via memoization.

The mark relation in Fig. 7 only needs to add bookkeeping to the surface syntax, so it is quite simple.

$$\begin{array}{rcl} \text{mark}(n) & = & n \\ \text{mark}(s) & = & /\emptyset/s \\ \text{mark}((T \ldots)) & = & /\text{val}/(\text{mark}(T) \ldots) \end{array}$$

Fig. 7. Mark Relation

We must also split evaluation into two pieces: "unval"-ing and "partial-eval"-ing. Unvaling takes a value and turns it into a suspended computation. Partial-evaling takes a suspended computation (or a value containing one) and tries to perform at least some evaluation in order to reduce work at runtime. Now, evaluation is just unvaling composed with partial evaluation.

$$\begin{array}{rcl} \text{unval}(n) & = & n \\ \text{unval}(\langle n\ \mathbf{o}\rangle) & = & \langle n\ \mathbf{o}\rangle \\ \text{unval}(/i''/\langle \mathbf{comb}\ n\ s'\ /i'_{f'}/E'\ (s \ldots)\ Tb\rangle) & = & /i''/\langle \mathbf{comb}\ n\ s'\ /i'_{f'}/E'\ (s \ldots)\ Tb\rangle \\ \text{unval}(/i/E) & = & /i/E \\ \text{unval}(/\emptyset/s) & = & /\text{true}/s \\ \text{unval}(/\text{val}/(T_1\ T_2 \ldots\ T_n)) & = & /\text{freshCall}/(\text{unval}(T_1)\ T_2 \ldots\ T_n) \end{array}$$

Fig. 8. Unval Relation

Unvaling (Fig. 8) a self-evaluating value is just the value itself, as there is no computation to suspend. The two forms that change when unvaling are symbols (which go from values to suspended lookups of that symbol in an environment) and arrays (which turn into suspended calls).

Contexts in Fig. 9 are defined like before, but augmented with our new "peval" (partial evaluate), combiner bodies, and multiple positions in the new "under" form.

$$
\begin{array}{rcl}
\mathcal{E} & := & \Box \mid [\text{peval } \mathcal{E}\ /i_x/E\ ES\ FS] \mid \langle \textbf{comb } n\ s'\ /i_x/E\ (s \ldots)\ \mathcal{E}\rangle \\
& & \mid [\text{under } \mathcal{E}\ (T \ldots)\ /i_x/E\ ES\ FS] \mid [\text{under } T\ (\mathcal{E}\ T \ldots)\ /i_x/E\ ES\ FS] \\
& & \mid [\text{under } T\ (T\ \mathcal{E}\ T \ldots)\ /i_x/E\ ES\ FS] \mid [\text{under } T\ (T \ldots\ \mathcal{E})\ /i_x/E\ ES\ FS] \\
& & \mid [\text{combine } \mathcal{E}\ (T \ldots)\ E] \mid [\text{combine } T\ (\mathcal{E}, T \ldots)\ E] \\
& & \mid [\text{combine } T\ (T \ldots, \mathcal{E}, T \ldots)\ E] \mid [\text{combine } T\ (T \ldots, \mathcal{E})\ E]
\end{array}
$$

Fig. 9. Contexts for Partial Evaluation

4.2 Partial-Eval Small-Step Semantics

In Appendix B we present some simplified pseudocode to give an additional reference and roadmap for the relations presented here.

For simplicity, we split the semantics into two subsections. The first one will talk about partial evaluation of everything but calls. The second one will just be the calls due to their complexity.

4.2.1 Partial Evaluation of Non-Calls. : The partial evaluation of non-calls is easier to understand, and it helps to get a feel for the basic semantics of partial evaluation.

$$
\begin{array}{rcl}
\mathcal{E}[E] & \rightarrow & \mathcal{E}[E'] \text{ (if } E \rightarrow E') \\
\\
[\text{peval } x\ /i_x/E\ ES\ FS] & \rightarrow & x \text{ if true} \notin \text{neededIDs}(x) \wedge \\
& & \text{neededIDs}(x) \cap ES = \emptyset \wedge \\
& & \text{resumeForms}(x) \cap ES = \text{resumeForms}(x) \\
& & \textbf{else:} \text{ continue below} \\
\\
[\text{peval } /x/s\ /i_x/E\ ES\ FS] & \rightarrow & \mathit{lookup}(s, E) \\
[\text{peval } /i'_{x'}/E'\ /i_x/E\ ES\ FS] & \rightarrow & /i'_{x'}/E'' \textbf{ if } /i'_{x''}/E'' \in ES \textbf{ else } /i'_{x'}/E' \\
\\
[\text{peval} & & \\
/i''/\langle \textbf{comb } n\ s'\ /i'_r/E'\ (s \ldots)\ Tb\rangle & & \\
/i_x/E\ ES\ FS] & \rightarrow & /i''/\langle \textbf{comb } n\ s'\ /i'_r/E'\ (s \ldots)\ Tb\rangle \\
\\
[\text{peval} & & \\
/i''/\langle \textbf{comb } n\ s'\ /i'_{f'}/E'\ (s \ldots)\ Tb\rangle & & \\
/i_x/E\ ES\ FS] & \rightarrow & \text{let } E'' = \langle\langle (s \leftarrow /i''/s) \ldots \mid s' \leftarrow /i''/s',\ /i_x/E\rangle\rangle \text{ in} \\
& & /i''/\langle \textbf{comb } n\ s'\ /i_x/E\ (s \ldots) \\
& & \quad [\text{peval } Tb\ /i''_f/E''\ \{/i''_f/E''\} \cup ES\ FS]\rangle
\end{array}
$$

Fig. 10. Non-Call Partial-Eval Semantics

In Fig. 10, we see the formal relations for partial evaluation of all forms except calls. The components of an actively partially evaluating form $[\text{peval } x\ /i_x/E\ ES\ FS]$ is as follows:

- The form that needs to be partially evaluated (x in the figure).
- The environment to partially evaluate it in ($/i_x/E$ in the figure)
- The "call stack" (a set in this case) of environments for evaluating the forms above this one (ES in the figure)

- A set of currently evaluating forms (*FS* in the figure) used to check for and prevent infinite recursion

Our first relation is the same as from the small-step semantics of the base language - that a larger expression can step if a sub-component of it as indicated by the Context form can step.

The next relation checks to see if partially evaluating the form will make any progress. There are three conditions which, if true, mean that the form cannot make progress:

- the form has already been partially evaluated ($\texttt{true} \notin \text{neededIDs}(x)$)
- none of the environments the form needs to progress are currently real or in our environment stack ($\text{neededIDs}(x) \cap ES = \emptyset$)
- the form hadn't previously stopped evaluating to prevent infinite recursion or we are no longer under the form that began the loop ($\text{resumeForms}(x) \cap ES = \text{resumeForms}(x)$)

If all three of these are true, partially evaluating the form won't have any affect and instead the form can be returned immediately.

On the other hand, if any of these are false the form might be able to make progress and it falls through to the remaining relations. Note that integers, symbol values, and array values will all already have been returned by the progress-check condition. We are only left with suspended symbol lookup, environment value, or derived combiner. For suspended symbol lookup, we just lookup the symbol in the current environment. If this environment is fake, it will return the same symbol marked with the ID of the environment that it would be found in if it were real. When partially evaluating an environment value we check the current environment_stack to see if the environment has a newer, real version (identified by ID) - if so we return it, else we return the environment unchanged.

A derived combiner is the most complicated case in this section. First, we check to see if its static environment is real or not. If it is, the fifth rule of Fig. 10 (as noted by the r subscript in the static environment $/i'_r/E'$ in the combiner value), we return it immediately, since this combiner has already been evaluated in a real environment. As a closure, it captured the environment from its original evaluation during its creation, and since it might have since moved this environment should not be replaced. On the other hand, a fake static environment (as noted by the f subscript in the static environment $/i'_f/E'$ in the combiner value) means the closure's initial evaluation has not yet happened and the closure is still at its original defining place. In this case, we replace the old static environment with the current one, and then create a new fake environment that maps the function's parameters to placeholder symbols marked with the ID of the combiner/fake environment. We further partially evaluate the combiner's body in this fake environment. The final result is a new derived combiner form that wraps up the new static environment and the partially evaluated body form.

4.2.2 Partial Evaluation of Calls. The partial evaluation of calls is more complex due to the high amount of bookkeeping that needs to be maintained. This complexity can be seen in Fig. 11.

First, we partially evaluate the first item in the array, which will be the combiner, transitioning to a *combine* form to indicate we are in the process of evaluating a call, as seen before in the base language semantics in Section 3. The next rule handles the case where the thing to be called isn't a combiner but instead a suspended symbol lookup ($/x/s$). In this case, we simply return the call as-is, marked as attempted, because we can't make progress. We do the same if the thing to be called is another suspended call ($/\texttt{atmdCall}\ x\ y/(T_1 \ldots)$). Otherwise, we have a combiner, either primitive or derived, and we extract the wrap_level from it. We then partial eval, unval, and partial eval again the arguments wrap_level number of times. If the combiner is a normal applicative, this would mean 1 time, like function calls in most other languages. The other common option is 0 times when

$$
\begin{array}{rcl}
[\text{peval } /\mathsf{freshCall}/(T_1\ T_2 \ldots)\ /i_x/E\ ES\ FS] & \rightarrow & [\text{combine} \\
 & & \quad [\text{peval } T_1\ /i_x/E\ ES\ FS] \\
 & & \quad (T_2 \ldots)\ /i_x/E\ ES\ FS] \\
[\text{peval } /\mathsf{atmdCall}\ x\ y/(T_1\ T_2 \ldots)\ /i_x/E\ ES\ FS] & \rightarrow & [\text{combine} \\
 & & \quad [\text{peval } T_1\ /i_x/E\ ES\ FS] \\
 & & \quad (T_2 \ldots)\ /i_x/E\ ES\ FS] \\
 & & \\
[\text{combine } /x/s\ (T_2 \ldots)\ /i_x/E\ ES\ FS] & \rightarrow & /\mathsf{atmdCall}\ \emptyset\ \emptyset/(/x/s\ T_2 \ldots) \\
[\text{combine } /\mathsf{atmdCall}\ x\ y/(T_1 \ldots) & & \\
\quad (T_2 \ldots)\ /i_x/E\ ES\ FS] & \rightarrow & /\mathsf{atmdCall}\ \emptyset\ \emptyset/ \\
 & & \quad (/\mathsf{atmdCall}\ x\ y/(T_1 \ldots)\ T_2 \ldots) \\
 & & \\
[\text{combine } \langle (S\ n)\ \mathbf{o} \rangle\ (V \ldots)\ /i_x/E\ ES\ FS] & \rightarrow & [\text{combine } \langle n\ \mathbf{o} \rangle \\
 & & \quad [\text{peval unval}([\text{peval } V\ /i_x/E\ ES\ FS]) \\
 & & \quad\ /i_x/E\ ES\ FS] \ldots\ /i_x/E\ ES\ FS] \\
 & & \\
[\text{combine} & & \\
/i''/\langle \mathbf{comb}\ (S\ n)\ s'\ /i'_{x'}/E'\ (s \ldots)\ Tb \rangle & & \\
\quad (V \ldots)\ /i_x/E\ ES\ FS] & \rightarrow & [\text{combine } /i''/\langle \mathbf{comb}\ n\ s'\ /i'_{x'}/E'\ s\ Tb \rangle \\
 & & \quad [\text{peval unval}([\text{peval } V\ /i_x/E\ ES\ FS]) \\
 & & \quad\ /i_x/E\ ES\ FS] \ldots\ /i_x/E\ ES\ FS] \\
 & & \\
[\text{combine} & & \\
/i''/\langle \mathbf{comb}\ 0\ s'\ /i'_{x'}/E'\ (s \ldots)\ Tb \rangle & & \\
\quad (V \ldots)\ /i_x/E\ ES\ FS] & \rightarrow & \mathbf{let}\ E'' = \langle \langle (s \leftarrow V) \ldots\ | s' \leftarrow /i_x/E,\ /i'_{x'}/E' \rangle \\
 & & \mathbf{let}\ C = /i''/\langle \mathbf{comb}\ 0\ s'\ /i'_{x'}/E'\ (s \ldots)\ Tb \rangle\ \mathbf{in} \\
 & & \mathbf{let}\ F = (C, E'')\ \mathbf{in} \\
 & & /\mathsf{atmdCall}\ \emptyset\ F/(C\ V \ldots)\ \mathbf{if}\ F \in FS \\
 & & \mathbf{else}\ [\text{under} \\
 & & [\text{peval } Tb\ /i''_r/E''\ (/i''_r/E'' \cup ES)\ \{F\} \cup FS] \\
 & & (/i''/\langle \mathbf{comb}\ 0\ s'\ /i'_{x'}/E'\ (s \ldots)\ Tb \rangle\ V \ldots) \\
 & & /i_x/E\ ES\ FS] \\
 & & \\
[\text{under } z & & \\
(/i''/\langle \mathbf{comb}\ 0\ s'\ /i'_{x'}/E'\ (s \ldots)\ Tb \rangle\ V \ldots) & & \\
\quad /i_x/E\ ES\ FS] & \rightarrow & \mathbf{let}\ o = returnOk(z, i')\ \mathbf{in} \\
 & & dropRV(z, /i_x/E, ES, FS)\ \mathbf{if}\ o = \mathsf{true}\ \mathbf{else} \\
 & & \mathbf{let}\ i_? = i_x\ \mathbf{if}\ s' \neq \emptyset\ \mathbf{else}\ \emptyset\ \mathbf{in} \\
 & & /\mathsf{atmdCall}\ i_?\ \emptyset/(/i''/ \\
 & & \quad \langle \mathbf{comb}\ 0\ s'\ /i'_{x'}/E'\ (s \ldots)\ Tb \rangle V \ldots)
\end{array}
$$

Fig. 11. Call Partial-Eval Semantics

the combiner is an operative, in order to take the code for the parameters in unevaluated. This might be so the operative can behave like a macro, or implement special control flow. Numbers greater than 1 for wrap_level are also possible, but rarely useful. We retain the ability for wrap_levels greater than 1 for uniformity, but have not used them in practice.

Now, we are ready to execute the actual call due to the parameters being evaluated the proper number of times according to the wrap_level in the combiner. The semantics for executing calls to primitive combiners are given as relations in Fig. 9 in Appendix A. For derived combiners, we create a new inner environment (E'') that maps the parameter symbols to the argument values, the special dynamic environment parameter symbol (s') to the current dynamic environment ($/i_x/E$), and chains up to the indicated static environment ($/i'_{x'}/E'$) from the combiner value. We first check to see if this (form, environment) pair is currently executing ($(C, E'') \in FS$), and return early if so to prevent infinite recursion. If it does return early, the returned call notes the (form, environment) pair that caused it to stop executing in its *atmdCall* form: $/\texttt{atmdCall}\ \emptyset\ F/(C\ V\ \ldots)$. Otherwise, the combiner body is evaluated in this new environment, with the call site marked by the *under* marker form. The *under* form delineates where a currently executing call is. This allows us to check whether the value/code resulting from partially evaluating a call site is legal to return whenever the form stops evaluating. To do this, it contains both the partial evaluation of the combiner body to execute the call as well as a fallback term. The fallback term, which is an updated suspended call in this case, is to be used if the call fails. The fallback term additionally carries the current dynamic environment, environment stack, and executing form set to be used in the next rule. The result is checked by the *returnOk* auxiliary function to see if this value is safe to return (defined in Appendix A, overview given in Section 4.3). If it is safe, it passes through *dropRV* before being returned (Fig. 8 in Appendix A).

The case where it is not legal to return a value is if there is, or could be, a reference that must be resolved in the inner environment created by this call. If the result is legal to return, the dropRedundentVeval (or *dropRV*) ensures no useless call to *veval* (a modified *eval* used in the primitive semantics (Fig. 9 in Appendix A)) to wrap any part of the result. This is important because these unnecessary wrapper calls to *veval* can block partial evaluation progress in some cases.

When the result form is not legal to return, we return a call form with the partially evaluated combiner (with updated wrap_level) and partially evaluated arguments based on the fallback value in the *under* form. If the combiner does take in a dynamic environment, we note the current dynamic environment's ID on the suspended call form ($i_?$).

4.3 Helper Relations

Next we quickly describe the various helper relations (fully defined in Appendix A) used in our definition of the partial evaluation relations.

- **Needed-For-Progress**: The needed-for-progress relation shows the set of IDs for which at least one needs to correspond to a real environment (static data) with values in order for the form to make progress. true means that it can make progress no matter what.
- **Needed-For-Progress-Upper**: An "upper" version of the needed-for-progress relation (in Appendix A) that tracks IDs of environments that are real themselves, but chain upwards to fake environment IDs.
- **Needed-For-Progress-Infinite**: This relation extracts the forms that have previously stopped executing to prevent infinite recursion. This is important when we are not currently executing in the call stack of one of these forms. If we are not, we should keep executing this form, as it has more evaluation to go before it hits another opportunity for infinite recursion. This is part of the mechanism that allows us to use the Y-combinator to implement recursion without either having infinite recursion or un-evaluated recursive calls with finite inputs.
- **Lookup**: As its names signifies, it finds the value associated with a symbol in an environment.

- **returnOk**: It determines if it is legal to return a particular result out of a particular environment. For example, a term can be returned out of a call if the ID of the inner environment/-combiner is not present in the term either explicitly or implicitly through a suspended call to a combiner that takes in its dynamic environment.
- **IDin**: It determines if this ID appears in this value without being under a combiner that introduces this ID.
- **takesDE**: It determines if this combiner takes in the dynamic environment.
- **dropRV** / dropRedundentVeval: Our final helper function removes extraneous calls to *veval* that can get in the way of further partial evaluation. *veval* is a version of the applicative *eval*, but with both parameters (the term to evaluate and the environment to evaluate it in) already unvaled and partially evaluated. It requires special handling (which it receives via the -1 wrap_level), because its term argument should not be partially evaluated via the normal machinery, which would use the wrong environment (the current dynamic environment, instead of the explicit environment passed to *veval*/*eval*). Calls to *eval* are common in macro-like operatives, where the normal final call of a macro-like operative is to *eval* the code constructed during the body of the combiner. This call to *eval* will partially evaluate to a call to *veval*, which will then return successfully to the call site. In a normal macro-like operative call, this call site's dynamic environment is the explicit environment passed to *veval*, and thus the call to *veval* is extraneous and the term will be inlined directly, completing the partial evaluation dance that fully expands macro-like operative calls. This removal is the responsibility of *dropRV*. A "macro-like operative combiner call" example that demonstrates this happening is located below in section 4.6.

4.4 Partial Eval Primitives Small-Step Semantics

Finally, we come to the (selected) semantics of the primitive combiners. For space, their formal definition is relegated to Appendix A. In general, they perform the same function as the simpler base primitives, but operate on the marked terms. The main combiners of interest are *if0* and *vau* which perform additional partial evaluation on their branches and body, respectively. We have already heard about *eval*'s half-life as *veval* and function in carrying along the proper environment to evaluate a suspended piece of code in until it can be unwrapped by *dropRV* and spliced into its final location. The formal definition of *eval*, *veval*, and *dropRV* are in Appendix A. The primitive implementations omitted are the trivial ones - they only evaluate if their parameters are fully evaluated values and they evaluate to the same value that they would under the base semantics.

4.5 Combined Effect of Partial Evaluation

The end result of all of this interconnected machinery is the removal of all statically called operatives where the operative was written in a macro-esque style. Note that the invariants maintained by partial evaluation ensure that derived combiners are only executed whose arguments are all values. Additionally, the result of a call to a combiner is either a value, a suspended computation where all calls either don't take in the dynamic environment (which would be the combiner's environment that it's being returned from), or an explicit call to veval providing its own environment. These are the cases that can be returned by a macro-like combiner! In addition, the partial evaluation process strips redundant calls to veval. Once the suspended code is returned from the call to the macro-like combiner, its call to veval will become redundant and the inner suspended computation will be inlined into the parent, just like how a macro would be expanded to code spliced into its calling location.

4.6 Examples

To illustrate the algorithm, we have some examples of marking, unvaling, and then partial-evaling in stages. The full step-by-step examples are too long for any paper, so these have been somewhat abbreviated.

4.6.1 Addition Example. We'll walk through the full partial evaluation steps for (+ 1 2) which is 3.

(+ 1 2) /val/(/∅/+ 1 2) /freshCall/(/true/+ 1 2)	The initial code Marked Then unvaled
[peval /freshCall/(/true/+ 1 2) /i_r/*E ES FS*]	We can't show the entire env, but for illustration say that E maps "+" to the primitive + combiner
[combine [peval /true/+ /i_r/*E ES FS*](1 2) /i_r/*E ES FS*] [combine ⟨1 +⟩ (1 2) /i_r/*E ES FS*]	Begin call, PV combiner Lookup replaces the symbol + with the primitive combiner
[combine ⟨0 +⟩ ([peval unval(1) /i_r/*E ES FS*] [peval unval(2) /i_r/*E ES FS*]) /i_r/*E ES FS*]	Unval+PartialEval to evaluate parameters, but integers stay the same
[combine ⟨0 +⟩ (1 2) /i_r/*E ES FS*] +(1 2) 3	And then the call Primitive does the calculation result is 3, as expected, which is legal to return

4.6.2 Constant Combiner Example. Now that we've done addition, we'll get slightly more complex by introducing the creation of a combiner with a body that can be partially evaluated. We take larger steps because we've gone over the details with the simple addition. In this case, we have (*vau* (*x*) (+ 1 2 *x*)) meaning "1+2+x" for some input x. We use just a pinch of syntactic sugar to have a 2-argument vau that is equivalent to the 3-argument vau that ignores its special dynamic environment parameter.

(*vau* (*x*) (+ 1 2 *x*)) /val/(/∅/*vau* /val/(*x*) /val/(/∅/+ 1 2 *x*)) /freshCall/(/true/*vau* /val/(*x*) /val/(/∅/+ 1 2 *x*))	The initial code Parsed and marked syntax Unvaled
[peval /freshCall/ (/true/*vau* /val/(*x*) /val/(/∅/+ 1 2 *x*)) /i_r/*E ES FS*]	We can't show the entire env. For illustration, E maps "vau" to the primitive vau combiner
[combine [peval /true/*vau* /i_r/*E ES FS*] (/val/(*x*) /val/(/∅/+ 1 2 *x*))) /i_r/*E ES FS*]	Begin call, PV combiner
[peval /7/⟨**comb** 0 *s′* /i_f/*E* (*x*) /freshCall/(/true/+ 1 2 /∅/*x*)⟩	The symbol Vau maps to its combiner value that will now

$/i_r/E\ ES\ FS]$ be partially evaluated

$/7/\langle \textbf{comb}\ 0\ s'\ /i_f/E\ (x)$ Partial evaluating the body
$[\text{peval}\ /\text{freshCall}/(/\text{true}/+\ 1\ 2\ /\emptyset/x)$ with fake environment. Notice, we
$/7_f/\langle\langle(x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle\ ES\ FS]\rangle$ are almost back to our first example

$/7/\langle \textbf{comb}\ 0\ s'\ /i_r/E\ (x)\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ +\rangle\ 3\ /7/x)\rangle$ We'll fast forward through the process from our first example

We moved quickly through the part mostly shared with the previous example. The only difference being the addition of the parameter reference x to the addition. The only thing to note is that $/\emptyset/x$ is unvaled to $/\text{true}/x$ (not shown) and then was partially evaluated to $/7/x$ (7 being the ID of the combiner/fake environment). Furthermore, the partial evaluation version of addition had to return a new partially evaluated call, since it could not yet evaluate x. We are left with a partially-evaluated combiner, but haven't yet seen this new technique do anything that previous partial evaluation techniques couldn't. For that, let's see a high-level view of how a macro-like operative call would be partially evaluated away.

4.6.3 Macro-like Operative Call Example. Due to the length of a step-by-step evaluation of this code, we will take larger jumps than before, omitting that which could be inferred from our two previous examples.

```
(let ( (double_parameter (vau de (x) (eval (array + x x) de))) )
     (vau (x) (double_parameter (+ 1 2 x))))
```

We'll skip over how "let" works for now (it's an operative combiner too) and focus on the partial evaluation of a macro-like operative part. This piece of code defines a macro-like operative called "double_parameter" that takes in a piece of code "x" unevaluated along with the dynamic calling environment "de" and then constructs the code "(+ x x)" as an array and evaluates it in de, the calling environment. This code is intentionally simplistic and will evaluate its argument twice (though the downsides of doing so are considerably reduced in a purely functional language where evaluating x cannot have side effects). This should bring to mind the classic macro example from C, "#define double(x) (x+x)", and indeed we chose it for its familiarity. Let's start by seeing what the macro-like f-expression itself looks like partially evaluated, and then we'll jump right into its application. This code:

```
(vau de (x) (eval (array + x x) de))
```

becomes

$/6/\langle \textbf{comb}\ 0\ de\ /i_r/E\ (x)\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ \textbf{eval}\rangle\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ \textbf{array}\rangle\ \langle 1\ +\rangle\ /6/x\ /6/x)\ /6/de)\rangle$

in a way very similar to our earlier examples of partially evaluated combiners. We have a combiner with two suspended calls, nested, with some suspended symbol lookups.
Now let's look at its use, when partially evaluating the body:

```
(vau (x) (double_parameter (+ 1 2 x)))
```

becomes, skipping forwards to evaluating the body

$/7/\langle \textbf{comb}\ 0\ \emptyset\ /i_f/E\ (x)\ [\text{peval}$
$/\text{freshCall}/(/\text{true}/\textit{double\_parameter}\ /\text{val}/(/\emptyset/+\ 1\ 2\ /\emptyset/x))$
$/i_r/\langle\langle(x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle\ ES\ FS]\rangle$ forwards to call

$$/7/\langle \textbf{comb}\ 0\ \emptyset\ /i_f/E\ (x)\ [\text{peval}\ /\text{freshCall}/(\ /6/\langle \textbf{comb}\ 0\ de\ /i_r/E\ (x)\ /\text{atmdCall}\ \emptyset\ \emptyset/ (\langle 0\ \textbf{eval}\rangle\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ \textbf{array}\rangle\ \langle 1\ +\rangle\ /6/x\ /6/x)\ /6/de)\rangle\ /\text{val}/(/\emptyset/+\ 1\ 2\ /\emptyset/x))\ /7_r/\langle\langle (x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle\ ES\ FS]\rangle$$ substituting in definition

We'll use a symbol for this nested environment, as it is quite unwieldy.

$$E'' = /6_r/\langle\langle (x \leftarrow /\text{val}/(/\emptyset/+\ 1\ 2\ /\emptyset/x))\ |/7_r/\langle\langle (x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle,\ /i_r/E\rangle\rangle$$

Let's continue along with the example.

$$[\text{peval}\ /\text{atmdCall}\ \emptyset\ \emptyset/ (\langle 0\ \textbf{eval}\rangle\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ \textbf{array}\rangle\ \langle 1\ +\rangle\ /6/x\ /6/x)\ /6/de)\ E''\ ES\ FS]$$ evaluating body with generated env

$$[\text{peval}\ /\text{atmdCall}\ \emptyset\ \emptyset/ (\langle 0\ \textbf{eval}\rangle\ /\text{val}/(\langle 1\ +\rangle\ (/\emptyset/+\ 1\ 2\ /\emptyset/x)(/\emptyset/+\ 1\ 2\ /\emptyset/x)) /7_f/\langle\langle (x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle)\ E''\ ES\ FS]$$ evaluating array call and its symbol parameters

$$[\text{combine}\ /i''/\langle -1\ \textbf{veval}\rangle (\text{unval}(/\text{val}/(\langle 1\ +\rangle\ (/\emptyset/+\ 1\ 2\ /\emptyset/x)(/\emptyset/+\ 1\ 2\ /\emptyset/x))) /7_f/\langle\langle (x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle) E''\ ES\ FS]$$ evaluating eval

$$[\text{combine}\ /i''/\langle -1\ \textbf{veval}\rangle (/\text{freshCall}/(\langle 1\ +\rangle\ (/\emptyset/+\ 1\ 2\ /\emptyset/x)(/\emptyset/+\ 1\ 2\ /\emptyset/x)) /7_f/\langle\langle (x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle) E''\ ES\ FS]$$ unval

$$\text{let}\ /7_f/E' = /7_f/\langle\langle (x \leftarrow /7/x)\ |,\ /i_r/E\rangle\rangle\ \text{in}\ \text{let}\ V' = [\text{peval}\ /\text{freshCall}/(\langle 1\ +\rangle\ (/\emptyset/+\ 1\ 2\ /\emptyset/x)(/\emptyset/+\ 1\ 2\ /\emptyset/x)) /7_f/E'\ \{/7_f/E'\} \cup ES\ FS]\ \text{in}\ [\text{under}\ V'\ (\langle -1\ \textbf{veval}\rangle\ V'\ /7_f/E')\ /i_x/E\ ES\ FS]$$ we also abbriviate E' stepping veval

V' will step to:

$$\text{let}\ V' = /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ +\rangle\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ +\rangle\ 3\ /7/x)\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ +\rangle\ 3\ /7/x))\ \text{in}$$

Now, the under call will fail to complete, as *returnOk* will return false, finding 7 in the result. Thus, the fallback will be returned and will be the final result of the call to *double_parameter*. This time, *returnOk* will be true, as 6, *double_parameter*'s ID, is nowhere to be found in the result. Thus, we will transition to a call to *dropRV* with the ID 7.

$$dropRV(/\text{atmdCall}\ \emptyset\ \emptyset/(\langle -1\ \textbf{veval}\rangle\ V'\ /7_f/E'),\ /7_f/E',\ ES,\ FS)$$

Of course, this is a redundant veval, so the final result of partially evaluating the call to *double_parameter* is:

$$V'$$

that is,

$$/\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ +\rangle\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ +\rangle\ 3\ /7/x)\ /\text{atmdCall}\ \emptyset\ \emptyset/(\langle 0\ +\rangle\ 3\ /7/x))$$

Note that this suspended computation is returned and replaces the call to the macro-like operative call. This code is the equivalent of:

```
(vau (x) (+ (+ 3 x) (+ 3 x)))
```

This is what we would have gotten if we had used a macro and basic constant propagation.

Note that this evaluation of the example did not depend on the exact values, the code generated by the macro-like operative, or the parameters to the call. In just this way, static calls to macro-like operatives will be partially evaluated away by our algorithm in a way congruent to macro-expansion and then constant propagation in a more standard optimizing Scheme implementation.

5 COMPILER BACKEND AND OPTIMIZATIONS

Partial evaluation can eliminate all inefficiencies from macro-like operatives but there are other inefficiencies left that require backend optimizations. One major remaining inefficiency is dynamic combiner call sites. In such cases, no local information is available ahead of time to determine if the combiner that will be called is an applicative or an operative - that is, whether the arguments will be evaluated or not and whether the combiner needs to access its calling dynamic environment. This would normally mean that code in the parameter position of dynamic calls cannot even be partially evaluated. To overcome this inefficiency, we tag the compiled combiner closure values with bits indicating their wrap level and need of the calling environment. Each dynamic call site branches on these bits. Inside the "wrap_level=0" side of the dynamic branch, a reference to the unevaluated arguments written out in static memory is emitted. Inside the "wrap_level=1" side of the dynamic branch, the compiler re-invokes unval and the partial evaluator recursively on each argument, resulting in code that is again as efficient as a language without fexprs (plus the overhead of the dynamic branch). Note that this requires that both the partial evaluation algorithm above as well as the compilation algorithm be extended to support failure. Failure during partial evaluation does not necessarily mean failure during run-time. For instance, a dynamic combiner might always be an operative with a wrap level of 0, and so some erroring parameter code is actually data, and will never be evaluated (and thus never lead to an error).

Garbage is collected via reference counting for simplicity. Since this is a purely functional language, there are no cycles to worry about. In order to remove the rest of the inefficiencies after partial evaluation, we have implemented various compiler optimizations. The following sections cover the other key optimizations implemented in our compiler.

5.1 Lazy Environment Instantiation

We delay the allocation and initialization of dynamic environment values until they are actually needed. Combiner calls that do take in the dynamic environment check a dedicated register to see if the environment value has already been created. If not, it creates it. This means the dynamic execution traces of combiner calls where there is no call that takes in the dynamic environment never reifies it and incurs only a single (predictable) branch of overhead. For static combiner calls, this information (if arguments are evaluated, if it takes in the surrounding environment, etc) is known at compile time, and no runtime branches are generated.

5.2 Type-Inference-Based Primitive Inlining

In order to reduce the overhead of every built-in operation being a combiner call with dynamic types, we implemented type-inference guided inlining of primitive operations. An analysis pass infers types based on branch predicates, which works quite well with the code generated by our match operative. For instance, the combiner *len* can be inlined to just a few bit-twiddling opcodes by determining that a particular variable must contain an array in *cond*.

For instance, consider the following code:

```
(cond (and (array? a) (= 3 (len a)))    (idx a 2)
      true                              nil)
```

Listing 5. Type Inference Example

The call to *idx* can be fully inlined without type or bounds checking because it resides in a block only reachable if the variable 'a' does contain an array of length 3. No type information is needed to inline type predicates, as they only need to look at the tag bits. Equality checks can be inlined as a simple word/ptr compare if any of its parameters are of a type that can be word/ptr compared (ints, bools, and symbols). When type inference and primitive inlining is combined together, it means that every primitive call in most match expressions can be fully inlined into a handful of opcodes apiece. In the above example, every single primitive listed will be inlined: the *cond* to WebAssembly if blocks, the predicate functions to bit-twiddling and branches, the idx to bit-twiddling and a load with a constant offset, etc.

5.3 Immediately-Called Closure Inlining

Inlining calls to closure values that are allocated and then immediately used helps incur no overhead for implementing some operatives. The main macro-like operative reaping the benefit is "let". As seen below, Listing 6 is partially evaluated to the equivalent of Listing 7, then inlined. As a result, the only overhead is the creation of a new environment, which is further made lazy and eliminated in the common case by Lazy Environment Instantiation. In this way, "let" is actually syntactic sugar for the definition and immediate call of a closure, like in many lambda calculi, but no efficiency is lost by doing so.

```
    (let (a (+ 1 2))
         (+ a 3))
```

Listing 6. Let Inlining Example

```
    ((wrap (vau (a) (+ a 3))) (+ 1 2))
```

Listing 7. Let Inlining Example - Expanded

5.4 Y-Combinator Elimination

Continuing the theme of making the classic lambda-calculi implementations of concepts as efficient as standard implementations, the final set of optimizations ensures no overhead from using the Y-Combinator to implement recursion. In Kraken, the Y-Combinator looks like Listing 8 with a tiny example of its use shown in Listing 9.

```
(let Y (lambda (f)
           ((lambda (x) (x x))
            (lambda (x) (f (wrap (vau app_env (& y) (lapply (x x) y app_env))))))))
)
```

Listing 8. The Y Combinator, as defined in Kraken

```
(Y (lambda (recurse) (lambda (n) (if (= 0 n) 1
                                     (* n (recurse (- n 1)))))))
```

Listing 9. A Factorial function explicitly using the Y Combinator

Normally, one does not manually use the Y-Combinator. In Kraken, there is a *rec-lambda* derived operative that is easy to use and evaluates Y-Combinator behind the scenes. Y-Combinator is actually always used to implement recursion in Kraken whether explicitly stated or not. This allows us to keep our language pure, and in agreement with the calculus.

This optimization actually falls out naturally from our architecture, with just a little bit of care taken while bookkeeping. When compiling a combiner, the compiler first inserts what the combiner index will be into a memoization dictionary before re-executing partial evaluation on the body of the combiner. Any static recursive calls will have the exact form of the combiner currently being compiled, and so the compiler can emit a static reference to the correct combiner index. All of this works because the re-executed partial evaluation of the body before compilation made sure to normalize the form of the combiner of a recursive call to be identical to that of the combiner being compiled before this partial-evaluation. Since this is an eager language, the definition of the Y-Combinator in our language has an extra closure to prevent infinite recursion inside the Y-Combinator itself. We, thus, additionally implement eta-conversion in the compiler to remove this extra level of indirection. Since the expression inside is now a constant instead of a call, there is no risk of infinite recursion. Combined with the normalization above, we achieve fully-efficient static recursive calls when using the Y-Combinator to define recursive functions.

Finally, as a purely functional Lisp, we use recursion instead of iteration. While we wait for the tail_call instruction in WebAssembly to be merged and implemented, we implemented a more limited form of Tail Call Elimination where auto-recursive calls in tail position are transformed into branches to the head of a loop that encloses the combiner's body. The combination of Tail Call Elimination with Y-Combinator Elimination above means that a recursive function defined using the Y-Combinator can be as efficient as an imperative loop in other languages. When WebAssembly finishes implementing the tail_call instruction, it can easily be emitted to gain full proper tail calls.

6 BENCHMARKS AND EVALUATION

We evaluate *Kraken* to answer the following set of questions:

- How much improvement does partial evaluation and our implemented compiler optimizations give *Kraken*?
- How much faster is our purely functional f-expr language, *Kraken* , compared to other implementations of fexprs?
- How does *Kraken*'s performance, with its fexprs, compare to macros?
- How do the different partial evaluation mechanisms/optimizations in *Kraken* contribute towards reduction in overall runtime?

Experimental Setup: We ran these experiments in a reproducible Nix environment on a NixOS install [Dolstra and Löh 2008] (Kernel 6.0.0) on a laptop with 8 cores / 16 threads and 64 GB of RAM. Our code contains the scripts and Nix Flakes needed to reproduce the exact set of dependencies to run our tests.

The Kraken benchmarks were run using both the Wasmtime and WAVM WebAssembly engines for most benchmarks. The Wasmtime WebAssembly engine is one of the most popular, developed by the Bytecode Alliance itself, and uses the CraneLift code generation backend. The WAVM WebAssembly engine is interesting for its use of LLVM, and it often produces the fastest code on benchmarks but has a higher startup time. We eliminated the Cfold Wasmtime benchmark due to problems running out of stack space (a known property of the Cfold benchmark).

Benchmarks: To showcase the capability of Kraken, we created benchmarks that are commonly implemented in functional languages and have been used as benchmarks in other papers [Reinking et al. 2021; Westrick et al. 2022]. The benchmarks are

- Fib - Calculating the nth Fibonacci number
- RB-Tree - Inserting n items into a red-black tree, then traversing the tree to sum its values
- Deriv - Computing a symbolic derivative of a large expression
- Cfold - Constant-folding a large expression
- NQueens - Placing n number of queens on the board such that no two queens are diagonal, vertical, or horizontal from each other

All benchmarks besides Fibonacci use the fexpr version of match for pattern matching in *Kraken*, which is equivalent to the macro version in NewLisp. We also RB-Tree using NewLisp's [Mueller 2018] version of fexpr match. We modified the sizes of the problems presented to the benchmark to account for the longer running times of some of the less-optimized implementations. The code for Kraken and NewLisp is very similar, and we should note that it is very unidiomatic NewLisp. Our goal was not to compare Kraken and NewLisp as implementation languages for Red-Black Trees, but to stress test a single reasonably complex fexpr/macro, namely pattern matching.

6.1 The Effect of Partial Evaluation on Eval Calls

Table 5. Number of eval calls with no partial evaluation for Fexprs

	Evals	Eval w1 Calls	Eval w0 Calls	Comp Dyn w1 Calls	Comp Dyn w0 Calls
Cfold 5	10897376	2784275	879066	1	0
Deriv 2	11708558	2990090	946500	1	0
NQueens 7	13530241	3429161	1108393	1	0
Fib 30	119107888	30450112	10770217	1	0
RB-Tree 10	5032297	1291489	398104	1	0

As mentioned before, using fexprs without partial evaluation will prelude optimization and cause a massive amount of repeated work. Table 5 and Table 6 show the number of calls to the *Kraken* runtime's eval function, the number of times the runtime's eval function executed a call to an applicative with wrap_level=1, the number of times the runtime's eval function executed a call to an operative with wrap_level=0, the number of compiled dynamic calls to applicatives with wrap_level=1, and the number of compiled dynamic calls to operatives with wrap_level=0. These are shown for *Kraken* test cases with partial evaluation turned off and turned on.

Table 6. Number of eval calls in Partially Evaluated Fexprs

	Evals	Eval w1 Calls	Eval w0 Calls	Comp Dyn w1 Calls	Comp Dyn w0 Calls
Cfold 5	0	0	0	0	0
Deriv 2	0	0	0	2	0
NQueens 7	0	0	0	0	0
Fib 30	0	0	0	0	0
RB-Tree 10	0	0	0	10	0

Table 7. Number of calls to the runtime's eval function for RB-Tree. The table shows the non-partial evaluation numbers -> partial evaluation numbers.

	Evals	Eval w1 Calls	Eval w0 Calls	Comp Dyn w1 Calls	Comp Dyn w0 Calls
RB-Tree 7	2952848 -> 0	757932 -> 0	233513 -> 0	1 -> 7	0 -> 0
RB-Tree 8	3532131 -> 0	906548 -> 0	279379 -> 0	1 -> 8	0 -> 0
RB-Tree 9	4278001 -> 0	1097965 -> 0	3383831 -> 0	1 -> 9	0 -> 0

Without partial evaluation, no compilation can be done because it is impossible to tell if arguments to calls will be evaluated. In all benchmarks, partial evaluation removed all calls to the runtime's eval function, resulting in a completely compiled program. Looking at RB-Tree, there are over a million calls to combiners with wrap level 1 (normal functions), and 398,000 calls to combiners with wrap level 0 (operatives replacing macros). This massive blowup in the number of calls is due to the repeated and exponential re-execution of macro-like-combiners in the definition of other macro-like-combiners, as discussed in the Introduction.

The non-partially-evaluated benchmarks show 1 compiled dynamic call to an applicative (its the first call into eval) and 0 compiled dynamic calls to operatives, because there is no compilation at all. For the partially evaluated benchmarks, there are a few compiled dynamic calls to applicatives due to higher-order function use in the benchmarks, and there are no compiled dynamic calls to operatives, as all operative use has been eliminated. We also varied the inputs for RB-Tree shown in Table 7 to give a sense for how the number scale with respect to input size.

The incredible slowdown implied by these tables comes to full fruition in our RB-Tree test in Fig. 13. We kept this run shorter because Kraken's non-partial-evaluating interpreter takes an incredibly long time even for 100 insertions (40 minutes). The compounding layers of repeated macro-like operative calls in the non-partially-evaluated Kraken version cause a 70,000x slowdown relative to the partial evaluated, optimized, and compiled version. For the remaining benchmarks, we remove the naive interpreted *Kraken* version, as in each case its performance is so bad as to blow out the graph and make it impossible to do any comparison. In our optimized Kraken, our partial evaluation algorithm is able to fully collapse these levels of inefficiency, evaluate and inline the results, and give the backend more specialized code to optimize, emitting a compiled version that handily beats not only the NewLisp-fexpr implementation but even the NewLisp-macro implementation, as can be seen in Fig. 14. We kept the benchmark sizes small in this test because the stack limits of NewLisp prevent sizes larger then 880, while the Tail Call Elimination performed by the *Kraken* compiler allows us to run much larger benchmarks, including the run of 4,800,000 inserts to the RB-Tree. This result shows the dramatic effect of partial evaluation and compiler optimizations on runtime for *Kraken*. Our technique takes the performance of a fully fexpr based language from being completely infeasible to being faster than a macro-based dynamic scripting language currently in use.

6.2 Comparison between Kraken Versions

Beyond the massive speedup from partial-evaluation, Fig. 12 and 13 show the effect of the various compiler optimizations we described by disabling them one by one. Our main four optimizations have a strong positive effect on runtime, with the exception of lazy environment instantiation. Lazy environment instantiation helps massively on fib, and some on Deriv, but generally hurts the rest slightly.

Fig. 12. Constant Fold and Deriv

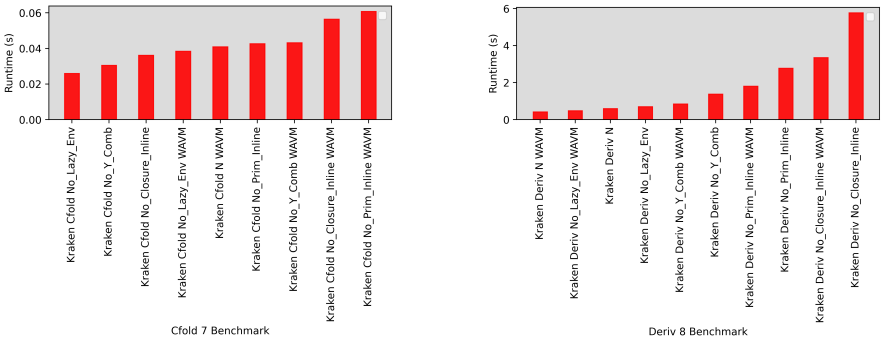
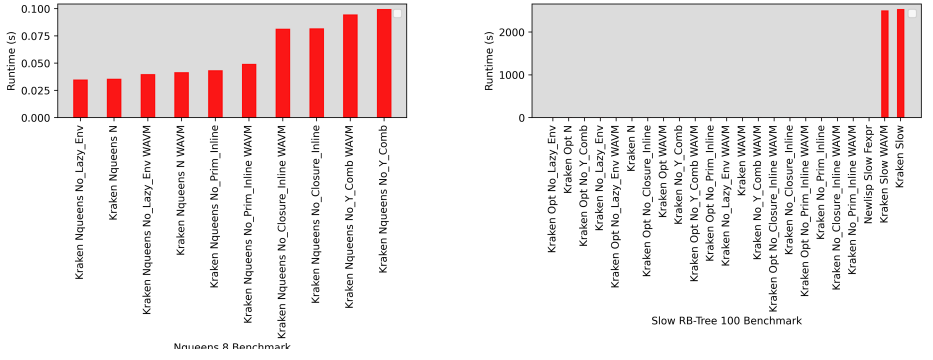


Fig. 13. N-Queens



6.3 Comparison against Others

To give a general idea of our current performance, we also show a Fibonacci benchmark that mostly exercises pure function-call speed and inlining as seen in Fig. 14. We include Python and Chez Scheme to give a general idea for where an exemplar slow and an exemplar fast dynamic language would fall. With the benefit of our partial evaluation, compilation, and leaning upon mature WebAssembly implementations, we beat both, but this should be taken with a grain of salt, as this is a very limited micro-benchmark only meant to give a general sense of the order of magnitude of our performance.

7 RELATED WORK

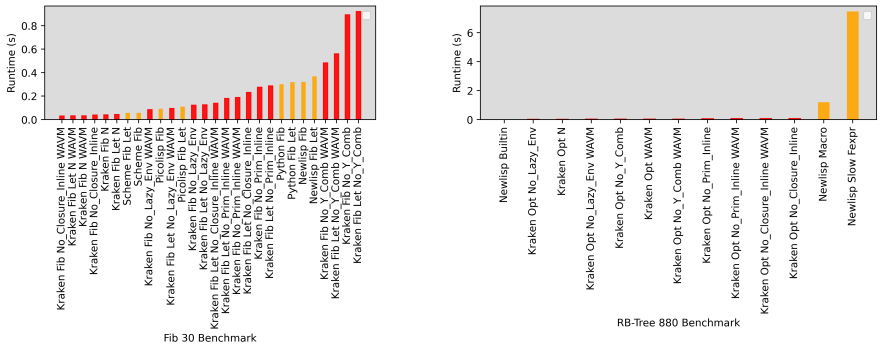
The focus of this paper has been on efficient fexpr language implementation using partial evaluation.

Other Fexpr Implementations:

Aside from NewLisp [Mueller 2018], picoLisp [Burger 2013] also includes fexprs as a language feature. Similar to NewLisp, picoLisp is a dynamic and interpreted language and does no special optimization of fexprs. Like NewLisp, most forms in picoLisp are implemented in the interpreter itself instead of being built up from smaller components, as is common to do with macros in other Lisps that use macros. While both of these languages originally lacked a macro implementation, NewLisp eventually added one due to the confusing evaluation rules and performance problems caused by its fexprs.

However, John Shutt demonstrated in his 2010 thesis [Shutt 2010] that if fexprs are re-formulated divorced from their historical context and co-existence with other now-extinct language features,

Fig. 14. Kraken vs. Others. Ordered by fastest to slowest



like dynamic scoping, they can be an elegant and well-behaved alternative to macros. It is important, however, that the language be designed with this in mind from the start. We have additionally shown that the performance problems that have plagued fexprs are solvable, fixing the other main issue that has kept fexprs from common usage.

Partial Evaluation:

Our work's main contribution is the partial evaluation scheme that provides the basis for the performance improvement of our fexpr implementation. Partial evaluation has been around for many years, including during the early years of Lisp [Lombardi 1964], and is utilized in many languages. Jones [Jones 1996] and Charles & Olivier [Consel and Danvy 1993] provide a high-level look at partial evaluation and its history in the community and explain well the differences between online and offline techniques. Ruf provides details of interesting online partial evaluation techniques in his thesis [Ruf 1993], and Sperber & Thiemann [Sperber and Thiemann 1996] give a description of the technique of using static data descriptions as a stand in for dynamic data in their paper on compilation using partial evaluation, which is similar to our handling of partially-static-data environments. Danvy, Malmkjær, and Palsberg use eta-expansion as a part of partial evaluation in their implementation of "The Trick" in [Danvy et al. 1996], whereas we use it during compilation to optimize the remnents of the Y-Combiner. Partial evaluation's application is broad, having been utilized for many imperative languages like C [Andersen 1992], Matlab [Elphick et al. 2003] and Pascal [Meyer 1991], as well as for other programming paradigms, such as logic programming, where Lloyd & Sheperdson [Lloyd and Shepherdson 1991] provided a theoretical foundation that got expanded by Alpuente et al [Alpuente et al. 1998]. Romph et al. use partial evaluation/staging to optimize data structures in Scala in [Rompf et al. 2013], Sperber & Thiemann further use partial evaluation in the generation of LR parsers in [Sperber and Thiemann 2000], and Sestoft uses partial evaluation to compile efficient ML pattern matching [Sestoft 1996].

8 CONCLUSION AND FUTURE WORK

In this work, we proposed a purely functional Lisp based on fexprs, *Kraken*, and the first-ever compilation framework to make fexprs performant by utilizing partial evaluation. The partial evaluator will evaluate away all fexprs that behave like macros (namely operatives of a particular form), showing that our marcro-esque fexprs can be as efficient as compile-time macros with a minor penalty to dynamic function calls. In addition, the partial evaluator with the compiler optimizations produced better-optimized code than the base interpreter (70,000x faster) or NewLisp's fexpr implementation [Mueller 2018] (233x faster). We have shown that fexprs can be a reasonably performant

alternative to macros and are a viable foundation for new expressive functional languages in the Lisp tradition.

In the future we will look at improving the partial evaluator to handle more flexible definitions of operatives that are less strictly macro-like, as well as investigation into adding features such has delimited continuations, which we had removed from Shutt's Vau calculi for simplicity.

REFERENCES

María Alpuente, Moreno Falaschi, and Germán Vidal. 1998. Partial evaluation of functional logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20, 4 (1998), 768–844.

Lars Ole Andersen. 1992. Self-applicable C Program Specialization. *PEPM* 92, 28 (1992), 54–61.

Andrew Berlin. 1990. Partial evaluation applied to numerical computation. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. 139–150.

Andrew Berlin and Daniel Weise. 1990. Compiling scientific code using partial evaluation. *Computer* 23, 12 (1990), 25–37.

Alexander Burger. 2013. The PicoLisp Reference.

William Clinger and Jonathan Rees. 1991. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. 155–162.

Charles Consel and Olivier Danvy. 1993. Tutorial Notes on Partial Evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 493–501. https://doi.org/10.1145/158511.158707

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-expansion does the trick. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18, 6 (1996), 730–751.

Eelco Dolstra and Andres Löh. 2008. NixOS: A Purely Functional Linux Distribution. *SIGPLAN Not.* 43, 9 (sep 2008), 367–378. https://doi.org/10.1145/1411203.1411255

Daniel Elphick, Michael Leuschel, and Simon Cox. 2003. Partial evaluation of MATLAB. In *International Conference on Generative Programming and Component Engineering*. Springer, 344–363.

Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, computers, controls* 25 (1971), 45–50.

Rich Hickey. 2008. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*. 1–1.

Neil D. Jones. 1996. An Introduction to Partial Evaluation. *ACM Comput. Surv.* 28, 3 (sep 1996), 480–503. https://doi.org/10.1145/243439.243447

Logan Kearsley. [n.d.]. Implementing a Vau-based Language With Multiple Evaluation Strategies. ([n. d.]).

H. Jan Komorowski. 1982. Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 255–267. https://doi.org/10.1145/582153.582181

J.W. Lloyd and J.C. Shepherdson. 1991. Partial evaluation in logic programming. *The Journal of Logic Programming* 11, 3 (1991), 217–242.

Lionello A Lombardi. 1964. Lisp as the language for an incremental computer. (1964).

Uwe Meyer. 1991. Techniques for partial evaluation of imperative languages. In *Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. 94–105.

Lutz Mueller. 2018. NewLisp.

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 96–111.

Tiark Rompf, Arvind K Sujeeth, Nada Amin, Kevin J Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 497–510.

Erik Steven Ruf. 1993. *Topics in online partial evaluation.* Stanford University.

Peter Sestoft. 1996. ML pattern match compilation and partial evaluation. In *Dagstuhl Seminar on Partial Evaluation*, Vol. 1110. 446–464.

John N Shutt. 2010. *Fexprs as the basis of Lisp function application or $vau: the ultimate abstraction.* Ph.D. Dissertation.

Michael Sperber and Peter Thiemann. 1996. Realistic Compilation by Partial Evaluation. *SIGPLAN Not.* 31, 5 (may 1996), 206–214. https://doi.org/10.1145/249069.231419

Michael Sperber and Peter Thiemann. 2000. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 2 (2000), 224–264.

Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement Detection with Near-Zero Cost. *Proc. ACM Program. Lang.* 6, ICFP, Article 115 (aug 2022), 32 pages. https://doi.org/10.1145/3547646

Patrick Henry Winston and Berthold K Horn. 1986. Lisp. (1986).

Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. *SIGPLAN Not.* 52, 6 (jun 2017), 662–676. https://doi.org/10.1145/3140587.3062381

A AUXILIARY HELPER RELATIONS

$$
\begin{aligned}
\text{neededIDs}(n) &= \emptyset \\
\text{neededIDs}(/\emptyset/s) &= \emptyset \\
\text{neededIDs}(/i/s) &= \{i\} \\
\text{neededIDs}(/\mathsf{val}/(T_1\ \ldots\ T_n)) &= \text{neededIDs}(T_1) \cup \cdots \cup \text{neededIDs}(T_n) \\
\text{neededIDs}(/\mathsf{freshCall}/(T_1\ \ldots\ T_n)) &= \{\mathsf{true}\} \\
\text{neededIDs}(/\mathsf{atmdCall}\ x\ y/(T_1\ \ldots\ T_n)) &= \{x\} \cup \text{neededIDs}(T_1) \cup \cdots \cup \text{neededIDs}(T \\
\text{neededIDs}(/x/\langle \mathbf{comb}\ n\ s'\ /i/E\ (s \ldots)\ T\rangle) &= (\text{neededIDs}(/i/E) \cup \text{neededIDs}(T)) - \{x\} \\
\text{neededIDs}(\langle n\ \mathbf{o}\rangle) &= \emptyset \\
\text{neededIDs}(/x/\langle\langle (s \leftarrow T) \ldots\ |s' \leftarrow /i'/E',\ /i/E\rangle\rangle) &= \text{neededIDs}(/i/E) \cup \text{neededIDs}(/i'/E') \\
\text{neededIDs}(/x/\langle\langle (s \leftarrow T) \ldots\ |,\ /i/E\rangle\rangle) &= \text{neededIDs}(/i/E)
\end{aligned}
$$

Fig. 15. Needed-For-Progress Relation

$$
\begin{aligned}
\text{upperIDs}(n) &= \emptyset \\
\text{upperIDs}(/x/s) &= \emptyset \\
\text{upperIDs}(/\mathsf{val}/(T_1\ \ldots\ T_n)) &= \text{upperIDs}(T_1) \cup \cdots \cup \text{upperIDs}(T_n) \\
\text{upperIDs}(/\mathsf{freshCall}/(T_1\ \ldots\ T_n)) &= \text{upperIDs}(T_1) \cup \cdots \cup \text{upperIDs}(T_n) \\
\text{upperIDs}(/\mathsf{atmdCall}\ x\ y/(T_1\ \ldots\ T_n)) &= \text{upperIDs}(T_1) \cup \cdots \cup \text{upperIDs}(T_n) \\
\text{upperIDs}(/x/\langle \mathbf{comb}\ n\ s'\ /i/E\ (s \ldots)\ T\rangle) &= \emptyset \\
\text{upperIDs}(\langle n\ \mathbf{o}\rangle) &= \emptyset \\
\text{upperIDs}(/x/\langle\langle (s \leftarrow T) \ldots\ |s' \leftarrow /i'/E',\ /i/E\rangle\rangle) &= \text{upperIDs}(/i/E) \cup \text{upperIDs}(/i'/E') \\
&\quad \text{if neededIDs}(/x/\langle\langle (s \leftarrow T) \ldots\ |/i'/E',\ /i/ \\
\text{upperIDs}(/x/\langle\langle (s \leftarrow T) \ldots\ |,\ /i/E\rangle\rangle) &= \text{upperIDs}(/i/E) \\
&\quad \text{if neededIDs}(/x/\langle\langle (s \leftarrow T) \ldots\ |,\ /i/E\rangle\rangle) = \\
\text{upperIDs}(/x/\langle\langle (s \leftarrow T) \ldots\ |s' \leftarrow /i'/E',\ /i/E\rangle\rangle) &= \{x\} \cup \text{upperIDs}(/i/E) \cup \text{upperIDs}(/i'/E') \\
&\quad \text{if neededIDs}(/x/\langle\langle (s \leftarrow T) \ldots\ |/i'/E',\ /i/ \\
\text{upperIDs}(/x/\langle\langle (s \leftarrow T) \ldots\ |,\ /i/E\rangle\rangle) &= \{x\} \cup \text{upperIDs}(/i/E) \\
&\quad \text{if neededIDs}(/x/\langle\langle (s \leftarrow T) \ldots\ |,\ /i/E\rangle\rangle) \neq
\end{aligned}
$$

Fig. 16. Needed-For-Progress-Upper Relation

$$
\begin{array}{rcl}
\text{resumeForms}(n) & = & \emptyset \\
\text{resumeForms}(/x/s) & = & \emptyset \\
\text{resumeForms}(/\mathsf{val}/(T_1\ \ldots\ T_n)) & = & \text{resumeForms}(T_1) \cup \cdots \cup \text{resumeForm} \\
\text{resumeForms}(/\mathsf{freshCall}/(T_1\ \ldots\ T_n)) & = & \text{resumeForms}(T_1) \cup \cdots \cup \text{resumeForm} \\
\text{resumeForms}(/\mathsf{atmdCall}\ x\ y/(T_1\ \ldots\ T_n)) & = & \{y\} \cup \text{resumeForms}(T_1) \cup \ldots \\
 & & \cup\ \text{resumeForms}(T_n) \\
\text{resumeForms}(/x/\langle \mathbf{comb}\ n\ s'\ /i/E\ (s\ldots)\ T\rangle) & = & \emptyset \\
\text{resumeForms}(\langle n\ \mathbf{o}\rangle) & = & \emptyset \\
\text{resumeForms}(/x/\langle\langle (s \leftarrow T)\ldots\ |s' \leftarrow /i'/E',\ /i/E\rangle\rangle) & = & \emptyset \\
\text{resumeForms}(/x/\langle\langle (s \leftarrow T)\ldots\ |,\ /i/E\rangle\rangle) & = & \emptyset
\end{array}
$$

Fig. 17. Needed-For-Progress-Resume-Infinite Relation

$$
\begin{array}{rcl}
 & & lookup(s, \\
lookup(s, \langle\langle (s_1 \leftarrow V_1)(s_2 \leftarrow V_2)\ldots\ |s' \leftarrow /i_x/E,\ /i'_{x'}/E'\rangle\rangle) & = & lookup(s, \langle\langle (s_2 \leftarrow V_2)\ldots\ |s' \leftarrow /i_x/E,\ /i'_{x'} \\
lookup(s, \langle\langle (s \leftarrow V)\ldots\ |s' \leftarrow /i_x/E,\ /i'_{x'}/E'\rangle\rangle) & = & V \\
lookup(s, \langle\langle\ |s \leftarrow /i_x/E,\ /i'_{x'}/E'\rangle\rangle) & = & /i_x/E \\
lookup(s, \langle\langle\ |s' \leftarrow /i_x/E,\ /i'_{x'}/E'\rangle\rangle) & = & lookup(s, E')
\end{array}
$$

Fig. 18. lookup

$$
\begin{array}{rcl}
returnOk(n, i) & = & \mathsf{true} \\
returnOk(o, i) & = & \mathsf{true} \\
returnOk(/\emptyset/s, i) & = & \mathsf{true} \\
returnOk(/\mathsf{val}/(V\ldots), i) & = & \mathsf{true} \\
returnOk(/i'/C, i) & = & \neg IDin(i, /i'/C) \\
returnOk(E, i) & = & \neg IDin(i, E) \\
 & & \\
returnOk(/y/(\langle -1\ \mathbf{veval}\rangle\ z\ /i'_{x'}/E), i) & = & returnOk(/i'_{x'}/E, i) \\
returnOk(/y/(f\ V\ldots), i) & = & \neg takesDE(f) \wedge \bigwedge returnOk(V, i)\ldots
\end{array}
$$

Fig. 19. returnOk

IDin (fg. 20) is quite simple, but pulled out to make returnOk more readable:

$$
IDin(x, i) \quad = \quad i \in \text{neededIDs}(x) \vee i \in \text{upperIDs}(x)
$$

Fig. 20. IDin

"takesDE" (fg. 21) is similarly simple, returning if the primitive or derived combiner takes in its dynamic environment.

$$
\begin{array}{rcll}
takesDE(\langle 0\ \mathbf{vau}\rangle) & = & \mathsf{true} & \\
takesDE(\langle 0\ \mathbf{if0}\rangle) & = & \mathsf{true} & \\
takesDE(\langle -1\ \mathbf{vif0}\rangle) & = & \mathsf{true} & \\
takesDE(\langle x\ \mathbf{y}\rangle) & = & \mathsf{false} & \text{(Other primitives don't)} \\
takesDE(/i''/\langle \mathbf{comb}\ n\ s'\ /i'_r/E'\ (s\dots)\ Tb\rangle) & = & s' \neq \emptyset & \\
takesDE(x) & = & \mathsf{true} & \text{(any other term, including suspended t} \\
 & & & \text{we count as true to be safe)}
\end{array}
$$

Fig. 21. takesDE

dropRV handles two main cases - the first is a call to veval, which is removed if it is redundant (if the ID of the explicit environment matches the ID of the current dynamic environment). The second is a suspended function call, which calls dropRV recursively on all parameters. If this does change the call, then we re-partially evaluate it, as perhaps with simpler parameters it can now be evaluated further.

$$
\begin{array}{rcl}
dropRV(/y/(\langle -1\ \mathbf{veval}\rangle\ z\ /i_f/E), /i_x/E, ES, FS) & = & dropRV(z, /i_x/E, ES, FS) \\
 & & \text{(only if } y \neq \mathsf{val}) \\
 & & \\
dropRV(/y/(\langle n\ \mathbf{o}\rangle\ V\dots), /i_x/E, ES, FS) & = & \mathbf{let}\ g\ =\ (\langle n\ \mathbf{o}\rangle\ V\dots) \\
 & & \mathbf{let}\ z\ =\ (\langle n\ \mathbf{o}\rangle\ dropRV(V, /i_x/E, ES, FS)\dots \\
 & & [\text{peval}\ /\mathsf{freshCall}/z, /i_x/E\ /i_x/E\ ES\ FS]\ \mathbf{if}\ z \\
 & & \mathbf{else}\ /y/z \\
 & & \text{(only if } y \neq \mathsf{val}, n \neq -1) \\
 & & \\
dropRV(& & \\
/y/(/i''/\langle \mathbf{comb}\ n\ s'\ /i'_r/E'\ (s\dots)\ Tb\rangle\ V\dots), & & \\
/i_x/E, ES, FS) & = & \mathbf{let}\ g\ =\ (/i''/\langle \mathbf{comb}\ n\ s'\ /i'_r/E'\ (s\dots)\ Tb\rangle \\
 & & \mathbf{let}\ z\ =\ (/i''/\langle \mathbf{comb}\ n\ s'\ /i'_r/E'\ (s\dots)\ Tb\rangle \\
 & & \qquad dropRV(V, /i_x/E, ES, FS)\dots) \\
 & & [\text{peval}\ /\mathsf{freshCall}/z, /i_x/E\ /i_x/E\ ES\ FS]\ \mathbf{if}\ z \\
 & & \mathbf{else}\ /y/z \\
 & & \\
dropRV(z, /i_x/E, ES, FS) & = & z \quad \text{Otherwise, return unchanged}
\end{array}
$$

Fig. 22. dropRV

$$
\begin{array}{rcl}
[\text{combine } \langle 0\ \textbf{eval}\rangle\ (V_1\ V_2)\ /i_x/E\ ES\ FS] & \to & [\text{combine } /i''/\langle -1\ \textbf{veval}\rangle \\
 & & \quad (\text{unval}(V_1)\ V_2) \\
 & & /i_x/E\ ES\ FS] \\
[\text{combine } \langle -1\ \textbf{veval}\rangle\ (V\ /i'_{x'}/E')\ /i_x/E\ ES\ FS] & \to & \textbf{let } V' = [\text{peval } V\ /i'_{x'}/E'\ (/i'_{x'}/E' \cup ES) \\
 & & [\text{under } V'\ (\langle -1\ \textbf{veval}\rangle\ V'\ /i'_{x'}/E')\ /i_x/E \\
[\text{under } z\ (\langle -1\ \textbf{veval}\rangle\ V'\ /i'_{x'}/E')\ /i_x/E\ ES\ FS] & \to & \textbf{let } o = \mathit{returnOk}(z, i')\ \textbf{in} \\
 & & \mathit{dropRV}(z, /i_x/E, ES)\ \textbf{if } o = \text{true } \textbf{else} \\
 & & /\text{atmdCall } \emptyset\ \emptyset/(\langle -1\ \textbf{veval}\rangle\ V'\ /i'_{x'}/E') \\
 & & \\
[\text{combine } \langle 0\ \textbf{vau}\rangle\ (s'\ (s\ldots)\ V)\ /i_x/E\ ES\ FS] & \to & [\text{peval} \\
 & & /\mathit{genID}()/\langle \textbf{comb } 0\ s'\ /i_f/E\ (s\ldots)\ \mathit{unva} \\
 & & /i_x/E\ ES\ FS] \\
 & & \\
 & & [\text{combine } \langle 0\ \textbf{wrap}\rangle \\
(/i''/\langle \textbf{comb } n\ s'\ /i'/E'\ (s\ldots)\ V\rangle)\ /i/E\ ES\ FS] & \to & /i''/\langle \textbf{comb } (S\ n)\ s'\ /i'/E'\ (s\ldots)\ V\rangle \\
 & & [\text{combine } \langle 0\ \textbf{unwrap}\rangle \\
(/i''/\langle \textbf{comb } (S\ n)\ s'\ /i'/E'\ (s\ldots)\ V\rangle)\ /i/E\ ES\ FS] & \to & /i''/\langle \textbf{comb } n\ s'\ /i'/E'\ (s\ldots)\ V\rangle \\
 & & \\
[\text{combine } \langle 0\ \textbf{if0}\rangle\ (V_c\ V_t\ V_e)\ /i_x/E\ ES\ FS] & \to & \textbf{let } F = (\langle 0\ \textbf{if0}\rangle\ V_c\ V_t\ V_e)\textbf{in} \\
 & & \textbf{let } FS' = \{F\} \cup FS\ \textbf{in} \\
 & & \textbf{let } V'_c = [\text{peval unval}(V_c)\ /i_x/E\ ES\ FS] \\
 & & \textbf{let } V'_t = \text{unval}(V_t)\ \textbf{if } F \in FS \\
 & & \quad \textbf{else } [\text{peval unval}(V_t)\ /i_x/E\ ES\ F \\
 & & \textbf{let } V'_e = \text{unval}(V_e)\ \textbf{if } F \in FS \\
 & & \quad \textbf{else } [\text{peval unval}(V_e)\ /i_x/E\ ES\ F \\
 & & [\text{under } V'_c\ (\langle -1\ \textbf{vif0}\rangle\ V'_c\ V'_t\ V'_e)\ /i_x/E\ ES \\
[\text{under } 0\ (\langle -1\ \textbf{vif0}\rangle\ V'\ V_t\ V_e)\ /i_x/E\ ES\ FS] & \to & [\text{peval } V'_t\ /i_x/E\ ES\ FS] \\
[\text{under } (S\ n)\ (\langle -1\ \textbf{vif0}\rangle\ V'_c\ V'_t\ V'_e)\ /i_x/E\ ES\ FS] & \to & [\text{peval } V'_e\ /i_x/E\ ES\ FS] \\
[\text{under } z\ (\langle -1\ \textbf{vif0}\rangle\ V'_c\ V'_t\ V'_e)\ /i_x/E\ ES\ FS] & \to & /\text{atmdCall } \emptyset\ \emptyset/(\langle -1\ \textbf{vif0}\rangle\ z\ V'_t\ V'_e) \\
 & & \quad \text{(otherwise for if)} \\
 & & \\
[\text{combine } \langle 0\ \textbf{int-to-symbol}\rangle\ (n)\ E\ ES\ FS] & \to & /\emptyset/'sn \text{ (a symbol made out of the numb} \\
[\text{combine } \langle 0\ \textbf{array}\rangle\ (V\ldots)\ E\ ES\ FS] & \to & /\text{val}/(V\ldots)
\end{array}
$$

Fig. 23. Semantics of Partial Eval Primitives

B PARTIAL EVALUATION PSEUDOCODE

Algorithm 1: Partial Evaluation Algorithm Pseudocode (except calls)

```
1  def PartialEval(form, env, env_stack, evaluating_forms_set): /*                    */
       Data: Our inputs are the current form to partially evaluate,
       the env to evaluate it in,
       the full call stack of currently active envs,
       and the set of evaluations currently taking place.
       Result: The partially evaluated form
2      /* First we check to see if we will make any progress by partially evaluating this form */
3      not_yet_evaled ← true ∈? neededIDs(x);
4      newly_real_env_ids ← ∃ i ∈ neededIDs(x) s.t. /i_x/E ∈ env_stack;
5      newly_unblocked ← evaluating_forms_set ∩ resumeForms(x) ≠ resumeForms(x);
6      if not_yet_evaled ∨ newly_real_env_ids ∨ newly_unblocked:
7          if form is an env value, /i_x/E:
8              /* grab the newer real version of this env, if it exists          */
9              if /i_r/E' ∈ env_stack:
10                 return /i_r/E'
11             else:
12                 return /i_x/E;
13         elif form is an derived-combiner value, /i/⟨comb n s /i'_x/E (s ...) Tb⟩:
14             /* recurse on the combiner's static env and body                   */
15             if x =? r:
16                 return /i/⟨comb n s /i'_x/E (s ...) Tb⟩;
17             else:
18                 new_env ← ⟨⟨(s ← /i/s) ... |s' ← /i/s', env⟩⟩;
19                 new_body ←
                     PartialEval(Tb, new_env, new_env ∪ env_stack, evaluating_forms_set);
20                 return ⟨comb n s env (s ...) new_body⟩;
21         else:
22             /* this is a call - broken out into Algorithm 2                    */
23             return PartialEvalCall(form, env, env_stack, evaluating_forms_set);
24     else:
25         /* partial evaluation won't make any progress, just return form unchanged */
26         return form;
```

Algorithm 2: Partial Evaluation Algorithm Pseudocode (for calls)

```
def PartialEvalCall(form, environment, environment_stack, evaluating_forms_set): /*                */
    Data: Our inputs are the current form to partially evaluate,
    the environment to evaluate it in,
    the full call stack of currently active environments,
    and the set of evaluations currently taking place.
    Result: The partially evaluated form
    /* first partially evaluate the combiner                                                       */
    /x/(T_1 T_2 ...) ← form;
    c ← PartialEval(T_1, env, env_stack, evaluating_forms_set);
    /* If the result is a suspended symbol lookup or a suspended call, return since we can't
       make progress                                                                               */
    if c = /x/s ∨ c = /atmdCall x y/(T' ...):
        return /atmdCall ∅ ∅/(c T_2 ...);
    /* Otherwise, c is a combiner, either primitive or derived. Get its wrap level                 */
    if c = ⟨n o⟩:
        wrap_level ← n
    else:
        /i/⟨comb n s' /i'_x'/E' (s ...) Tb⟩ ← c
        wrap_level ← n
    /* evaluate the the parameters until wrap_level is 0 (or not at all if it is -1)              */
    args ← (T_2 ...)
    if wrap_level > 0:
        args ← (PartialEval(T, env, env_stack, evaluating_forms_set) for T ∈ args);
        if any entry in args is not a value:
            return /atmdCall ∅ ∅/(c args ...);
        repeat
            args ← (unval(T) for T ∈ args);
            args ← (PartialEval(T, env, env_stack, evaluating_forms_set) for T ∈ args);
            if any entry in args is not a value:
                new_c = replacewraplevel(c, wrap_level)
                return /atmdCall ∅ ∅/(new_c args ...);
        until wrap_level = 0;
    if c = ⟨n o⟩:
        return drop_redundant_eval(o(env, env_stack, args), env, env_stack);
    else:
        /* make our inner environment                                                              */
        /i/⟨comb n s' /i'_x'/E' (s ...) Tb⟩ ← c
        inner_env ← /i/⟨⟨(s ← V for (s, V) ∈ zip((s ...), args) |s' ← env, /i'_x'/E'⟩⟩;
        /* Check if we're already evaluating this form, to prevent infinite recursion              */
        new_c = replaceWrapLevel(c, wrap_level)
        if (Tb, inner_env) ∈ evaluating_forms_set:
            return /atmdCall ∅ (Tb, inner_env)/(new_c args ...);
        result ←
          PartialEval(Tb, inner_env, {inner_env} ∪ env_stack, {(Tb, inner_env)} ∪ evaluating_forms_set);
        if combiner_return_ok(result, env.id):
            return drop_redundant_eval(result, env, env_stack);
        elif s' ≠ ∅:
            /* If this combiner takes in the dynamic environment, track the current dynamic
               environment ID as needed for this call to progress                                  */
            return /atmdCall env.id ∅/(new_c args ...);
        else:
            return /atmdCall ∅ ∅/(new_c args ...);
```