# An Essay on Partial Evaluation

*Advanced Topics in Programming Languages*

Offline partial evaluation relies on a binding time analysis to identify which parts of a program are static and can therefore be specialized. At the same time binding-time improvements are necessary to enable greater portions of the code to be partially evaluated. Without using $\eta$-expansions as a binding time improvement, a greater burden falls on the programmer to make their code amenable to partial evaluation.

In *Eta-expansion does The Trick* Danvy, Malmkjær and Palsberg perform binding-time analysis as a series of typing judgments that augment the $\lambda$-calculus with binding time information. They use $\eta$ -expansion to fully specialize a program such that the only remaining terms are entirely dynamic. Sperber and Thiemann describe in *The Essence of LR Parsing* two implementations of a functional LR parser that can be partially evaluated to yield runtime improvements. However, the partial evaluator they use does not specialize their parsers sufficiently, leading to the manual introduction of binding time improvements that could have otherwise been captured using $\eta$-expansion.

## *Danvy, et al.'s Binding Time Improvements*

Danvy, et al. implement two sets of binding time improvements in their partial evaluator: continuation-passing style code-motion, and $\eta$-expansion. The former improvement is captured by Similix — the partial evaluator used in *The Essence of LR Parsing* — and will not be elaborated for the purposes of this essay.

An $\eta$-expansion wraps a term by its type's corresponding constructor and destructor to add a layer of indirection to the underlying data. For example, a dynamic pair $p$ of type $d$ is $\eta$-expanded to $\overline{\langle}\underline{\text{fst}}\, p, \underline{\text{snd}}\, p\overline{\rangle}$, where an $\overline{\text{overlined}}$ operator denotes that it is static, and an $\underline{\text{underlined}}$ operator is dynamic. $\eta$-expansions serve to coerce static terms into dynamic contexts and vice-versa. This is particularly useful when dealing with partially static data.

Say, we have a static function $f$ which we want to apply so that we can unwrap — and therefore specialize — it. Moreover, let $f$ take as input a sum of type $d + (d \times d)$. Unfortunately, in our program, when we apply $f$, our argument to $f$ is some dynamic value $v$. Applying $f \,\underline{@}\, v$ would prevent any specialization here. $\eta$-expansion allows us to coerce $v$ into a static value so that we can specialize $f$:

$$f \,\overline{@}\, \left(\underline{\text{case}}\, v\, \underline{\text{of}}\, \mathsf{L}\, x \to \overline{\mathsf{L}}\, x \mid \mathsf{R}\, y \to \overline{\mathsf{R}}\, \overline{\langle}\underline{\text{fst}}\, y, \underline{\text{snd}}\, y\overline{\rangle}\right) \rightsquigarrow$$
$$\underline{\text{case}}\, v\, \underline{\text{of}}\, \mathsf{L}\, x \to f \,\overline{@}\, \left(\overline{\mathsf{L}}\, x\right) \mid \mathsf{R}\, y \to f \,\overline{@}\, \left(\overline{\mathsf{R}}\, \overline{\langle}\underline{\text{fst}}\, y, \underline{\text{snd}}\, y\overline{\rangle}\right)$$

Thus, we have one example among many, showing how $\eta$-expansion works as a binding time improvement to enable further specialization.

## *Scheming the Type System*

The implementation of the LR-parsing algorithms given by Sperber and Thiemann is given in the Scheme language, which has a much more complicated type system compared to Danvy, et al's. In order to apply the desired $\eta$-expansion binding time improvements to the LR parsers, we must therefore extend Danvy et al.'s framework. Specifically, the parsing algorithms make use of let-expressions, scoped definitions, and recursive functions.

Let-expressions can simply be transformed into $\lambda$-expressions as follows:

$$\text{let}\, x = e_1 \ \text{in}\ e_2 \rightsquigarrow (\lambda x.e_2)\, e_1$$

The Scheme programs given in Sperber and Thiemann's paper use `define` statements which create potentially recursive scoped definitions. One way to implement these would be to simply inline all instances of any scoped variables, eliminating `define` statements altogether. This, however, may lead to a lot of code duplication in the residual program whenever such a definition is dynamic. Alternatively, the binding time analysis could include a second environment holding these scoped definitions with their corresponding augmented types such that specialization is only done when necessary.

Lastly, recursive functions will require more complex type inference to deduce the binding time types of these recursive functions. Such type inference is already common in most functional programming languages and should not add much overhead, but it will need to be augmented for dynamic and static types.

With these modifications to Danvy's two-level $\lambda$-calculus, we should be able to automatically apply $\eta$ -expansions for Sperber and Thiemann's parsers.

## *$\eta$-Expanding the Parsers*

Sperber and Thiemann describe two Scheme implementations of a functional LR parser, a direct-style "textbook" version, and a continuation-passing style version. The first parser implementation operates as follows: Given a parser state — a set of LR grammar rules — the function `parser` reduces a non-terminal in one of the current state's grammar rules until it reaches the bottom. Once it cannot reduce any rules further, it calls `act-on` which shifts the input of the rules in the current parsing state until we reach the end. At this point we have fully parsed a non-terminal and we unroll the call stack up to the non-terminal we finished parsing.

Results of a call to `parser` are given as a sum type: $1 + (\texttt{int} \times \texttt{int} \times d)$, where $d$ is the dynamic type of the leftover input that has yet to be parsed. Sperber and Thiemann represent the input as a list of tokens which we can represent as a product comprised of the head and tail of a list. To be able to partially evaluate this first parser, both `parser` and `act-on` must be applied to static arguments. Unfortunately, `act-on` takes these parse results as input which are partially dynamic, leading to the residual code containing fragments like the following:

```
let result = Reduce(2, 2, input) in
act-on (r-lhs result) (r-dot result) (r-inp result)
```

Here, the `r-xxx` functions retrieve the elements of a constructor `Reduce` element, with the type of `(r-inp result)` being $d$. This means that when we apply `act-on` it must be done dynamically since one of its arguments is dynamic, and the partial evaluator is unable to specialize further. The solution, is to use $\eta$-expansion on the argument as shown above, producing $\overline{\langle}\underline{\mathrm{fst}}\,(\texttt{r-inp result}), \underline{\mathrm{snd}}\,(\texttt{r-inp result})\overline{\rangle}$. This argument is now static and we can therefore specialize the body of the `act-on` function. Sperber and Thiemann get around this issue by adding rewriting rules as part of the Similix postprocessor. Having to do this, however, places an undue burden on the programmer to extend the partial evaluator themselves in order to get good results.

Instead of unrolling the stack once we finish parsing a non-terminal, the second implementation of the LR parser keeps a list of continuations that we can jump to. This naturally results in a faster implementation as we no longer have to unwind for every time we shift or reduce a symbol. However, since the continuations are dynamic, Sperber and Thiemann run into a problem here, as any operation that uses them cannot be specialized. For example, when extracting a continuation which we want to apply from the list of active continuations, their parser gets stuck:

```
list-get-n (cons c0 continuations) (length (item-rhs item))
```

The continuations are partially dynamic since the parser state to which they jump back to is dependent on the input. Hence, the application to the function `cons` is marked as dynamic which infects the rest of the expression as being dynamic. However, the size and composition of the list only depends on static data, i.e. the rules of the grammar, and so we should reasonably be allowed to extract elements from the list of continuations during specialization. Once again, we can $\eta$-expand the dynamic function `c0` such that it can be used in a static context:

$$\texttt{list-get-n (cons } \left(\overline{\lambda} x.\ \texttt{c0}\ \underline{@}\, x\right) \texttt{ continuations) (length (item-rhs item))}$$

Now, the application to `cons` is static and the partial evaluator can reduce the rest of the expression. Thus, we see the versatility of $\eta$-expansion as it is able to coerce any dynamic value to be used in any static context, and vice versa.

### *Learning to do 'The Trick'*

We further see in their paper how Sperber and Thiemann manually introduce 'The Trick' as a binding time improvement for both implementations of the parser. 'The Trick' expands a dynamic value in a static context into cases for each of the values it may take. So given a dynamic sum value $v$ found in context $C[v]$, The Trick performs the following transformation:

$$C[v] \rightsquigarrow \underline{\text{case}}\, v\, \underline{\text{of}}\, \mathsf{L}\, x \to C[x] \mid \mathsf{R}\, y \to C[y]$$

This allows the now static expressions $C[x]$ and $C[y]$ to be specialized. Danvy, et al. notice that this transformation is simply the $\eta$-expansion of the sum type coupled with a CPS transform, such that their binding time analysis and improvements are already able to capture this behavior. Their framework, thus, further relieves pressure from the programmer as they do not need to manually introduce the Trick.

In cases, such as a parser, where the set of values that $v$ may take is quite large (e.g. the set of terminals and non-termnials), applying The Trick in this way manually would bloat the size of the source code immeasurably, introducing multiple points of failure. This is why Sperber and Thiemann use a variation of the Trick that loops through all possible values of $v$, instead of having a giant `case` expression. While this reduces the size of the code, it still requires the programmer to identify instances in their code where The Trick applies and introduce it properly. This need for a manual implementation of The Trick reveals a deficiency in the partial evaluator, one that can be remedied by the introduction of $\eta$-expansions.

In conclusion, we have seen how many of the binding time improvements that Sperber and Thiemann have to manually implement in their implementations of a functional LR parser can be manually introduced using the $\eta$-expansion Danvy, et al. describe for their type system. Nevertheless, this type system is quite primitive and would require extending in order to be efficiently applied to more practical programming languages. For example, one drawback is the large `case` statements that 'The Trick' introduces without any looping expressions. Nevertheless, as $\eta$-expansion can be applied at every program point, it makes for an extremely versatile binding-time improvement resulting in much more extensive partial evaluation. Thus, $\eta$-expansion is necessary for a partial evaluator that does not burden the programmer with the task of enabling specialization themselves.