Reviving the Lost Art of Combinator Graph Reduction

Daan de Graaf
Master's student Computer Engineering
TU Delft / Eindhoven University of Technology
Eindhoven, Netherlands
d.j.a.degraaf@student.tudelft.nl / d.j.a.d.graaf@student.tue.nl

***Abstract*—Combinator Graph Reduction was once the dominant execution strategy for lazy functional languages, but inefficient translation algorithms and lack of an open-source implementation caused the technique to fall into obscurity in the late 1980s. The traditional translation algorithm produces a combinator expression that is in the worse case quadratic in the size of the input. Recently, Kiselyov proposed a new family of translation algorithms that generate smaller combinator expressions, with some variants being linear in the size of the input. While the effect on code size is known, the difference in runtime performance has remained unquantified. Our goal is to find out how the reduction in code size affects execution time on a modern, efficient Combinator Graph Reduction engine. In absence of a suitable engine, we have developed our own based on the design of Miranda and TIGRE. We compile a set of test programs with Kiselyov's translation algorithms and measure their execution time on our best engine.**

Our benchmarks show that our Miranda-style engine outperforms the original Miranda, and is therefore a good reference point to measure performance improvements against. We find that the threaded code approach proposed in TIGRE reduces the number of instructions and data accesses needed to execute programs, but frequent cache misses make it unattractive on architectures with split instruction and data caches. Kiselyov's new translation algorithms are shown to significantly boost performance in our benchmarks. The effect on execution time is even greater than we observe on code size. We conclude that the new translation algorithms significantly improve performance, and there may be further gains to be had by combining ideas from the different translation algorithms.

***Index Terms*—combinatory logic, graph reduction, lazy functional language, lazy evaluation**

I. Submission

Submitted in partial fulfillment of the requirements for completion of the TU Eindhoven course Advanced Functional Programming (Capita Selecta SET, 2IMP05), under the guidance of prof. Tom Verhoeff.

II. Introduction

Lazy evaluation of programs is a powerful idea. It allows a program to operate on infinitely large data structures, and if part of the computation turns out to be unnecessary, it is more efficient than a strict evaluation. In practice, delaying computation often introduces overhead, making languages based on lazy evaluation slower than their strict counterparts. To compete with a strict implementation, a lazy runtime must maximize laziness while minimizing overhead. It is a difficult balance, and over the years many different evaluator designs, or *abstract machines*, have been proposed [1]–[6].

One of the early approaches to implementing lazy functional languages is combinator graph reduction [1]. It was the basis for David Turner's Miranda language [7], the popular choice for pure functional languages and one of the few to be commercially supported at the time. Eventually, the interest in Miranda and combinator graph reduction with it waned. Part of the reason may be that Miranda was until very recently closed-source, licensed software (its source code was released only in 2020 [8]), leading to the creation of the open-source Haskell language [9]. But combinator reduction also deserves part of the blame. One of the major issues is that the translation to combinators may result in much larger programs. For a lambda calculus expression of size N, the traditional bracket abstraction algorithm produces an output of $O(N^2)$ in the worst case, where other approaches such as supercombinators are linear [10]. Along with other disadvantages, this led to combinator graph reduction being all but abandoned.

Recently, however, a new algorithm for the translation of lambda calculus expressions into combinatory logic has been proposed that is linear in the size of the input [11]. Kiselyov's translation algorithm is based on semantics rather than syntax and this deeper understanding of the program has been shown to produce more compact translations. While a clear improvement over previous translation strategies, it remains to be seen how big the impact is on real-world performance. Is it enough to bridge the performance gap with newer reduction engines based on different paradigms? This leads us to our first research question:

RQ1: Does Kiselyov's semantic translation make combinator graph reduction competitive with contemporary lazy functional evaluators?

We compare Kiselyov's translation algorithm to bracket abstraction by compiling a set of benchmark programs and measuring their execution time. Future work will add GHC as a reference point for a contemporary evaluator, along with additional benchmark programs.

We also need a fast Combinator Graph Reduction engine that can run programs compiled to combinator expressions. Alas, there is no performant reduction engine tailored to executing such programs. The now open-sourced Miranda software is a good starting point, but it is showing its age. The

2020 port compiles cleanly on a recent GCC, but enabling optimizations is known to break the garbage collector (Section V covers this issue in more detail). Miranda's architecture is based on cells with a tag, head, and tail as described in [1], but later work has shown that a tagless representation with direct pointers to code leads to better performance [5]. In absence of an up-to-date reference on building a practical combinator reduction engine, we formulate the second research question:

RQ2: How is combinator graph reduction implemented efficiently on general-purpose hardware?

We evaluate the state of the art in combinator graph reduction and build a fast implementation for a language that is simple but powerful enough to express our benchmark programs. Using this engine, we can then answer **RQ1**.

Concretely, our contributions are the following:

- Kiselyov develops not one but two algorithms. The first does not have linear complexity but tends to produce very compact code. The second is guaranteed to be of linear complexity, at the cost of being less compact in certain cases. Kiselyov's description requires that the reader be familiar with OCaml and the tagless-final style of programming. In Section IV we describe both algorithms in a more conventional style that is directly implementable in a general-purpose language.
- We implement a fast combinator graph reduction engine based on the ideas in Miranda [7] and later improvements [12]. We elaborate on its design in Sections V and VI.
- We evaluate the performance of Miranda, our engines, and Kiselyov's translation algorithms on a set of benchmark programs in Section VII.

For readers unfamiliar with lambda calculus, combinatory logic or bracket abstraction, we provide a short introduction to these topics in Section III.

III. Preliminaries

A. Lambda Calculus

For an accessible introduction to the Lambda Calculus, we can recommend [13].

The Lambda Calculus comes in many flavors. The most basic version has only function abstraction and application and is untyped. Kiselyov's description of his new algorithms [11] uses a typed variant. Our description of the translation is not defined as a type system, hence the untyped Lambda Calculus suffices for our treatment. We do, however, enrich the Lambda Calculus with a few primitives that are convenient in a practical implementation. They are:

- Machine integers (signed 32-bit). These can be emulated with e.g. Church Encoding, but arithmetic over a native integer type is faster.
- Arithmetic operations. To manipulate machine integers, we introduce the usual $+$, $-$, $*$ and $/$ operations.
- Comparison operations. To compare machine integers, we also add $==$, $<$ etc. These operations return 1 if true, and 0 if false (machine integers do double duty as a boolean type).

Most introductory resources, including [13], present Lambda Calculus with named bindings. Kiselyov's algorithms [11] use De Bruijn indices [14] instead. As this is core to the algorithm, we adopt the same approach for our algorithm descriptions in Section IV. However, in the example expressions we give in this paper, we continue to use the easier-to-read named bindings.

B. Combinatory Logic

Chapter 16 of [10] starts with a good introduction to Combinatory Logic and the Bracket Abstraction algorithm.

Combinatory Logic is a notation designed to remove variables (named or otherwise) from Lambda Calculus expressions. The core idea is to define a set of basic functions, or *combinators*, that can be combined to pass variables to the proper subexpressions. We only need three combinators to convert any Lambda Calculus expression:

- $S\ f\ g\ x = f\ x\ (g\ x)$: Passes an argument to both subexpressions of an application.
- $K\ x\ y = x$: Drops the second argument.
- $I\ x = x$[1]: The identity function that accepts an argument and returns it.

The traditional algorithm to compile Lambda Calculus expressions into Combinator Logic with these three combinators is Bracket Abstraction, which we present in the next section.

The set of combinators used in the translation may be extended to generate smaller programs. Kiselyov's algorithms [11] for example use the following additional combinators as special cases of S:

- $B\ f\ g\ x = f\ (g\ x)$: Passes an argument to the *right* subexpression.
- $C\ f\ g\ x = (f\ x)\ g$: Passes an argument to the *left* subexpression.

Another addition, specifically for the linear algorithm, is the addition of *bulk* combinators, variants of S, B and C that pass multiple arguments:

- $S_n\ f\ g\ x_1\ \ldots\ x_n = (f\ x_1\ \ldots\ x_n)\ (g\ x_1\ \ldots\ x_n)$
- $B_n\ f\ g\ x_1\ \ldots\ x_n = f\ (g\ x_1\ \ldots\ x_n)$
- $C_n\ f\ g\ x_1\ \ldots\ x_n = (f\ x_1\ \ldots\ x_n)\ g$

C. Bracket Abstraction

Bracket Abstraction is the traditional algorithm for compiling lambda calculus expressions into combinatory logic. The basic algorithm requires only the S, K, and I combinators. Its compilation function can be defined recursively as:

[1]$SKK = I$, so even this combinator is not strictly necessary, but all reasonable implementations do include I.

$$\begin{array}{lcl} C[\![\ e_1\ e_2\ ]\!] & = & C[\![\ e_1\ ]\!]\ C[\![\ e_2\ ]\!] \\ C[\![\ \lambda x.e\ ]\!] & = & A\ x\ [\![\ C[\![\ e\ ]\!]\ ]\!] \\ C[\![\ cv\ ]\!] & = & cv \\ & & \\ A\ x\ [\![\ f_1\ f_2\ ]\!] & = & S\ (A\ x\ [\![\ f_1\ ]\!])\ (A\ x\ [\![\ f_2\ ]\!]) \\ A\ x\ [\![\ x\ ]\!] & = & I \\ A\ x\ [\![\ cv\ ]\!] & = & K\ cv \end{array}$$

Any lambda calculus expression can be compiled using this algorithm. As an example, we can compile the expression $\lambda x.\lambda y.x$, and obtain:

$$\begin{array}{ll} C\ [\![\ \lambda x.\lambda y.x\ ]\!] & \Rightarrow \\ A\ x\ [\![\ C\ [\![\ \lambda y.x\ ]\!]\ ]\!] & \Rightarrow \\ A\ x\ [\![\ A\ y\ [\![\ C\ [\![\ x\ ]\!]\ ]\!]\ ]\!] & \Rightarrow \\ A\ x\ [\![\ A\ y\ [\![\ x\ ]\!]\ ]\!] & \Rightarrow \\ A\ x\ [\![\ K\ x\ ]\!] & \Rightarrow \\ S\ (A\ x\ [\![\ K\ ]\!])\ (A\ x\ [\![\ x\ ]\!]) & \Rightarrow \\ S\ (K\ K)\ I & \end{array}$$

The generated expression consists exclusively of combinators, as expected. We can show that it produces the same result as the original lambda expression by applying it to two abstract variables x and y:

$$\begin{array}{ll} S\ (K\ K)\ I\ x\ y & \Rightarrow \\ (K\ K\ x)\ (I\ x)\ y & \Rightarrow \\ K\ x\ y & \Rightarrow \\ x & \end{array}$$

While correct, the Bracket Abstraction algorithm is not optimal. Looking at the derivation above, $S\ (K\ K)\ I$ is identical to the K combinator. In Section IV, we cover the better compilation algorithms proposed in [11].

D. Graph Reduction

A good tutorial-style introduction to Graph Reduction may be found at [15]. For a more formal and detailed reference, see [10].

Graph Reduction is a common technique for implementing non-strict functional programming languages, i.e. languages that evaluate expressions only when their value is needed. Nodes in the graph are either constant values (leaf nodes) or application nodes (representing the application of a function to an argument). Edges are used to connect application nodes with their function and argument nodes. The usual convention is to denote application nodes with @. Here is how we might represent the expression `square 3`:

```
     @
    / \
square   3
```

This graph has three nodes: a reference to the function `square`, an argument 3, and an application node. We can *reduce* this graph by applying rewrite rules to it. For example, we know that an expression `square x` produces a value $x{*}x$. For the above graph, we can therefore state that:

```
     @
    / \     reduces to    9
square   3
```

The example graph happens to be a tree, but this is not always the case: A node may be referenced by multiple application nodes. Consider the graph for `square (square 3)`:

```
      @
     / \
    |   @
    |  / \
  square   3
```

The two application nodes *share* the `square` node. When we perform the reduction of `square 3`, the upper application node maintains its (now exclusive) reference to `square`, and the other application node is replaced with the constant value 9. Sharing nodes is important for efficiency because it prevents unnecessary re-evaluation of expressions. The graph representation of `(square 3) + (square 3)` highlights this:

```
        @
       / \
      @   |
     / \  |
    +   \ |
         @
        / \
   square   3
```

We would like to evaluate `square 3` only once. One way to do this is to replace the top application node of an evaluated expression with the new value. For the example graph above, we get:

```
        @
       / \
      @   |
     / \  |
    +   \ |
         9
```

Now both the left and right sides of the + see the updated value (notice that we are passing the two arguments to + using a double application node).

A graph could have multiple possible rewrites (reduction steps) that we could perform, in which case we need to decide what rewrite rules should be applied first. There are many possible ways to make this decision [10], but most practical implementations choose a *left-outermost* reduction strategy. In the graph representation, this amounts to traversing the graph from the root, following *left* edges of application nodes. This chain of edges from the root down is called the *spine* of the reduction. Eventually, we find a leaf node and apply the rewrite rule that matches that particular node (if any apply). The reduction then repeats from the root again, until eventually, no more rewrites are possible.

At this point, it could be that the graph has been reduced to a constant value (e.g. 9). Another possibility is that parts of the graph are still reduceable, but our reduction strategy does not ‘reach’ it. For example, we can have the following graph (including a K combinator) that our strategy will not reduce:

```
      @
     / \
    K   @
       / \
square   3
```

K requires two arguments, but we have given it only one, so the usual rewrite rule $K\ x\ y \Rightarrow x$ does not apply. Because the graph contains a subexpression that *could* be reduced, we say the reduction has reached *Weak Head Normal Form*, rather than *Normal Form*.

IV. Kiselyov’s algorithms

Kiselyov’s paper [11] describes three iterations of his non-linear semantic translation, and finally the linear algorithm. They are presented as a type system accompanied by an operational semantics, except for the η-optimization, which is given as an OCaml code patch. OCaml’s type system can represent these algorithms very concisely in a tagless-final style. Below we describe the same algorithms in a more conventional style. Our description places no requirements on the type system and can be directly implemented in any general-purpose programming language. For the first and most basic algorithm, we give full definitions of all functions inline. Definitions for the more advanced algorithms are given in Appendix B.

A. Strict algorithm

The compilation function below, which we denote strict, is based on the rules in Figure 6 of Kiselyov’s paper [11]. We call it *strict* it does not have lazy weakening: most notably it lacks the K-optimization required for full laziness [10].

Kiselyov describes the algorithm as a type system with an operational semantics. The type system describes how to assign a type, or context, to a lambda calculus term. The operational semantics define how a typed term may be converted into combinatory logic. A naive implementation thus requires two phases: an initial typing step, followed by a translation step. As Kiselyov already notes in his treatment of the linear algorithm further on in the paper, the translation does not care what elements are in the context, just that we keep track of the size of the context. This lets us express compilation as a recursive function, shown in Figure 1.

There are two cases for λe, and this is precisely where the number of elements in the context matters: if the context is empty, the first rule applies, otherwise, we use the second one. Figure 2 defines the semantic function.

These functions are equivalent to Kiselyov’s type system and can be implemented directly and efficiently in a conventional programming language.

$$
\begin{array}{lll}
\text{strict}\ \ z & = & 1 \models I \\
\text{strict}\ \ s\ e & = & n+1 \models (0 \models K) \coprod (n \models c) \\
 & & \text{where } (n \models c) := \text{strict}\, e \\
 & & \\
\text{strict}\ \ \lambda e & = & 0 \models K\ c \\
 & & \text{where } (0 \models c) := \text{strict}\, e \\
\text{strict}\ \ \lambda e & = & (n-1) \models K\ c \\
 & & \text{where } (n \models c) := \text{strict}\, e \\
 & & \\
\text{strict}\ \ e_1\ e_2 & = & (n \models ((n_1 \models c_1) \coprod (n_2 \models c_2)) \\
 & & \text{where } (n_1 \models c_1) := \text{strict}\, e_1 \\
 & & \text{where } (n_2 \models c_2) := \text{strict}\, e_2 \\
 & & \text{where } n := \max n_1\, n_2
\end{array}
$$

Fig. 1. Compilation function for the strict algorithm.

$$
\begin{array}{ll}
(0 \models c_1) \coprod (0 \models c_2) & = c_1\ c_2 \\
(0 \models c_1) \coprod (n_2 \models c_2) & = (0 \models Bc_1) \coprod (n_2 - 1 \models c_2) \\
(n_1 \models c_1) \coprod (0 \models c_2) & = (0 \models CCc_2) \coprod (n_1 - 1 \models c_1) \\
(n_1 \models c_1) \coprod (n_2 \models c_2) & = (n_1 - 1 \models (s \coprod l)) \coprod r) \\
 & \quad \text{where } s := (0 \models S) \\
 & \quad \text{where } l := (n_1 - 1 \models c_1) \\
 & \quad \text{where } r := (n_2 - 1 \models c_2)
\end{array}
$$

Fig. 2. Semantic function for the strict algorithm.

B. Lazy weakening

The addition of Kiselyov’s *Lazy Weakening* requires that we track which items in the context are ignored. The context representation changes from an integer to a list of boolean values. In the new compilation function lazy, shown in Figure 19, we use the shorthand t for the value *true* and f for *false*. We always match the last element of lists and denote the remainder of the list by Γ.

We have introduced yet another case for λe, this time to distinguish between t or f as the last element in the context of the inner expression. There is also the new function $\sqcup$ to merge the left and right contexts of a function application term, whose definition is given in Figure 20. Merging follows the natural rule that if either of the two contexts requires a particular item, the merged context also requires it.

In our implementation, we unroll the recursion into a while loop and repeatedly pop elements from the two arrays until both are empty. This builds the new context back to front, so as a final step, the array is reversed. Alternatively, an implementation can pad the shorter list (at the start!) with f and zip them. This is particularly convenient for languages that provide a zip function with a default value[2].

[2]such as `ziplongest` in Python.

Our definition of the semantic function now has to handle more cases, and thus is somewhat larger. See Figure 21 for the full definition. The first four cases are direct translations from the previous definitions, the others are new additions to handle ignored elements. The last two cases are not included in the original paper, but they are necessary to make the pattern matching exhaustive, and are present in the reference OCaml implementation[3].

C. Eta optimization

Kiselyov covers eta optimization as a change to his OCaml code rather than typing rules. Our recursive functions are derived from the typing rules, so these changes are not as straightforward to incorporate. The introduction of the V element means that a context may now be associated with either a closed expression or the top variable (we reuse V as a marker for this). Our implementation uses a tagged union to define a type that can be either an expression or the top variable. For the compilation function, we only need to change the definition of $\text{lazy}_\eta\ z$ and add an extra case for $\text{lazy}_\eta\ \lambda e$. Figure 22 shows the new compilation function.

Our context merge function $\sqcup$ now also depends on the compiled expression, since we have a special case for V. The updated definition is given in Figure 23.

The V value effectively encodes an additional implicit used element in the context. We also have some new cases for the semantic function, available in Figure 24.

The case $(\emptyset \models V) \coprod (\emptyset \models V)$ is impossible in this translation: it represents the application of a value to itself, e.g. $\lambda x.x\ x$. For such an x, we cannot assign a proper type. Implementations may throw an error or leave it as undefined behavior.

With the addition of the expression value to $\sqcup_\eta$, the function now takes the same input as $\coprod_\eta$ and has very similar matching patterns. Our implementation, therefore, folds context merging into $\coprod_\eta$ to make the code more concise.

D. Linear algorithm

The implementation of the linear algorithm is much simpler because it is based on the strict algorithm. The compilation function is identical save for a different semantic function $\coprod_{\text{linear}}$, shown in Figure 26. For the full definition, see Figure 25.

The main achievement is that the semantic function is no longer recursive. The linear algorithm visits each term exactly once, so arguing its linearity becomes an almost trivial exercise.

We leave the addition of full laziness and η-optimization to the linear algorithm as future work.

V. Miranda

For our base implementation of a Combinator Graph Reduction engine, we try to stick to Miranda's design as much as possible. While there is published work on Miranda [7], it does not provide much detail on the reduction engine. Fortunately, the source code for Miranda is now freely available [8], and from the code, we can deduce that its design is very similar to SASL [1], an earlier language developed by Turner.

[3]From [11]: https://okmij.org/ftp/tagless-final/skconv.ml

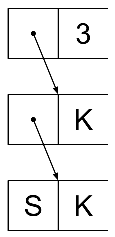

Fig. 3. SKK 3 in graph representation.

The approach we describe here is the one taken for our implementation, which may not fully correspond to Miranda's. Not all implementation issues are fully described in the literature, and in places where Miranda's implementation was not clear to us either, we developed our own solution. In particular, Section V-C covers *our* approach to the resumption of the main reduction after arguments to strict combinators have been reduced, but this may be implemented differently in Miranda.

Miranda converts source programs into combinatory logic using Turner's modified bracket abstraction algorithm [16]. The converted program, composed of combinators, constants and function application, is then loaded into a graph structure. The graph is binary: every node has exactly two children, and the node itself represents the application of the left node to the right one. Let us start with an example program SKK 3. The S combinator takes three arguments, but a node can only store the application to a single argument. The left associativity of function application lets us rewrite the example program as $((SK)\ K)$ 3. In this form, it is straightforward to construct the associated binary graph, shown in Figure 3.

To reduce this graph, we follow the left pointer until we find a combinator to apply. As we traverse the graph along what is often called the *spine*, we keep the nodes we have seen on a stack, because their right nodes contain the arguments to the combinator we eventually find.

In this example graph, after following two pointers we find the S combinator, and we apply the definition $Sfgx \Rightarrow (fx)(gx)$ to rewrite the graph, as shown in Figure 4.

The reduction process continues until the program has been reduced to a constant value, or there are insufficient arguments for the next combinator to reduce. The expression is said to be in *weak head normal form*: some inner part of the expression may be reducible, but we only perform outermost reduction, so this is as close as we can get to the normal form.

A. Sharing

In the reduction in Figure 4, we have copied the value 3 to two nodes. For another reduction where the x value is a pointer to a subgraph, performing the reduction may result in multiple pointers to the same expression. Expressions in Miranda are *referentially transparent*: we can safely replace

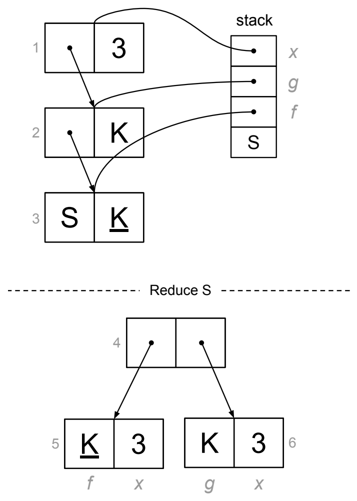

Fig. 4. First reduction step for SKK 3. One of the K combinators is underlined to make them easier to distinguish.

any expression with the value it computes. Computing x in both branches would therefore be wasteful. Instead, we would like to compute it once (in whatever branch runs first) and have the second branch directly reuse the result. It turns out that this is not too difficult to implement: when we make a reduction step, we update the outermost node with the contents of the new outermost node. In Figure 4, this would mean we overwrite node 1 with the contents of node 4. Now any other pointers to node 1 automatically point to the reduced equivalent. If the expression reduces to a single value v (say an integer), we can use the I combinator to make an *indirection node* Iv.

B. Representing the graph

Miranda represents the graph using three arrays, `hd` (for head) stores the left pointer, `tl` (for tail) stores the right pointer, and `tag` stores metadata for the cell. Owing to its early inception when 64-bit machines were rare, `hd` and `tl` are 32 bits wide. `tag` only needs to store a few bit flags and thus is a single byte wide. Our implementation stores two flags in `tag`:

- `WANTED`: Set if this cell is currently in use. Cells without this bit set are considered free and may be overwritten to store new nodes in the graph. More on this in Section V-D.
- `RHS_INT`: Set if the right pointer is not a pointer but an integer value. Note that the left pointer can never be an integer because the left pointer is always a function for which the right pointer is the first argument.

Miranda integers have unlimited precision, using GNU MP [17] internally. For simplicity (and performance), our implementation instead treats the cell pointer as a 32-bit signed integer.

It is unclear why a representation with three separate arrays was chosen over a single array where each element contains a 'cell' struct with a head, tail, and tag. We suspect decomposing the graph into separate arrays leads to poor data locality, as almost any operation will access all three parts of a cell. Our base implementation respects this design choice, later implementations do not.

Miranda reserves the lower range of cell pointer values for special purposes:

- 0..255 represents the Latin 1 character set.
- 256..305 for internal use in the lexer.
- 306..446 to encode combinators.

The first 446 cells in the heap are unused: when the reduction engine encounters their value, they are interpreted as a character, lexer rule or combinator. This explains why we have to tag integers but not combinators or character values.

Our implementation does not support character values and needs no reserved number for the lexer, so we can encode the first combinator as 0. The bulk combinators introduced by the `linear` algorithm do complicate the encoding somewhat: in principle, there is now an infinite amount of possible combinators. However, compiled programs only contain a finite subset of those combinators, and the reduction process only creates combinators that were present in the original program. We can have an efficient encoding of combinators, but only if we generate it for specific programs. Our implementation keeps an array of function pointers, each pointing to the code to reduce a particular combinator. The array is populated as the compiled program is loaded into the graph representation: when we encounter a combinator we have not seen before, its implementation is added to the array, and the combinator is encoded into the graph as its index into the array. At runtime, combinator resolution is done by indexing into the array and calling the associated function.

C. Strict combinators

Miranda has special built-in combinators to perform arithmetic operations and compare numbers. Where most combinators place no restrictions on their arguments, these combinators require their arguments to be fully reduced to integers (hence call them *strict*), and so require special treatment.

When a *strict* combinator is about to be reduced, the engine inspects the arguments. If all arguments are reduced, the combinator reduction is performed as usual. If one or more arguments are not yet reduced, we push the pointers to those arguments onto the stack and proceed to reduce from the top of the stack. Eventually, the top argument will have been fully reduced, producing an integer value. In general, if a reduction results in an integer value, it can mean one of two things:

- The value is the result of the program.
- The value is one of the arguments for a strict combinator.

We distinguish between these cases by looking at the size of the stack: if it is empty, we have computed the final result of the program. Otherwise, we continue reducing from the top of the stack, which will be either another argument or the original combinator expression. The pointer to the original complex argument is replaced with an indirection node to the value, using the update logic described earlier.

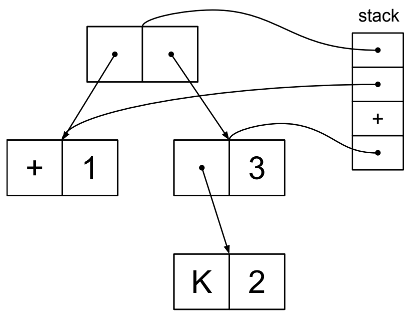

Fig. 5. Graph of $+\ 1\ (K\ 2\ 3)$ after the second argument has been added to the stack.

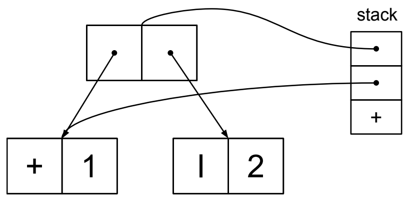

Fig. 6. Graph of $+\ 1\ (K\ 2\ 3)$ after the second argument has been reduced.

As an example, consider the expression $+\ 1\ (K\ 2\ 3)$. After we have traversed the spine and encountered the $+$ combinator, the engine notices that the second argument is not yet reduced, and adds it to the stack. Figure 5 shows the state of the graph at this point in the reduction process.

Once $K\ 2\ 3$ has been reduced to 3, the engine observes that the stack is not yet empty, so it creates an indirection node and resumes reduction from the top of the stack. Figure 6 reflects this state.

This example shows that it is not sufficient to check the `tag` of each argument to see if it is an integer: indirection nodes also count as fully reduced.

D. Garbage collection

The graph reduction process may lead to some nodes becoming unreachable. Coming back to the example in Figure 4, after the reduction step nodes 2 and 3 are no longer used: they are *garbage*. In this example, it is easy to conclude which nodes are garbage, but in larger programs, there are often multiple references to the same node, so we cannot mark nodes as garbage during the reduction step. Miranda employs a simple mark-and-sweep garbage collector that kicks in when the engine runs out of free cells while allocating a new node. In the first phase, it zeroes out the `WANTED` bit on all tags in the heap. Then in the marking phase, starting from the elements currently on the stack[4], the garbage collector traverses the graph and sets the `WANTED` bit on all reachable nodes. There is no real sweep phase: instead, the allocation routine walks over the tag array until it finds a cell that is not marked as `WANTED`. This also means that there is no list of currently free cells, which we suspect may lead to performance issues in large heaps.

We consider the garbage collector in Miranda to be somewhat simplistic and inefficient, and we suspect there is much to improve in terms of efficiency. Our benchmark programs do not use too much heap space, so we choose not to focus our efforts on this, and defer optimizations on this front to future work.

[4]Miranda keeps its reduction stack on the C stack. This makes it difficult to reliably enumerate all cell pointers on the stack at a given point in time, because the compiler may choose to keep some of the pointers in registers. For this reason, users are instructed not to enable optimizations when compiling Miranda, which hurts performance but is necessary for correctness. When Miranda was developed, compilers were not advanced enough to perform such optimizations, so this was not an issue.

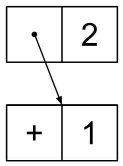

Fig. 7. Graph of $1+2$ in the Miranda-style engine. Tags omitted for brevity.

VI. TIGRE

The TIGRE engine, first presented in [5], is a combinator graph reduction engine like Miranda. Originally designed for the VAX instruction set, our implementation targets the x86-64 instruction set but mirrors the TIGRE design otherwise. The TIGRE paper focuses on key differences from Miranda, the elimination of tags and directly executable graph nodes, so where the TIGRE paper omits implementation details we take the same approach as our reference implementation based on Miranda.

A. Eliminating tags

The TIGRE authors point to tags as a major source of inefficiency in graph reduction engines. At evaluation time Miranda must check the tag of nodes to see if their right-hand side is a value or a pointer to a node. TIGRE does away with this by adding a special `LIT` combinator. The `LIT` combinator takes one argument guaranteed to be a literal value and returns it, like an `I` combinator specialized for constants. An apparent downside of this is that simple expressions require more indirection, and therefore a larger graph to reduce. Take for example the expression $1 + 2$, which in the Miranda-style engine can be represented with just two nodes, shown in Figure 7.

The TIGRE version of this program, shown in Figure 8, is larger, occupying four nodes.

In this example, the TIGRE graph may indeed be slower to evaluate, but the same does not hold for the programs in general. If the values 1 and 2 were the results of a previous computation, they would also be behind an indirection node in the Miranda engine. The TIGRE approach is only slower when performing operations on values that are constant in the source program, in other cases the reduced branching on tag values is expected to speed up execution.

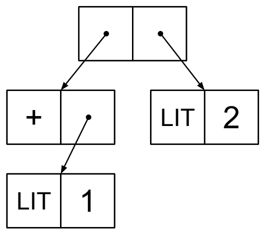

Fig. 8. Graph of $1 + 2$ in the TIGRE-style engine.

The TIGRE authors report that some engines also use the tag to distinguish between combinators and cell pointers. In Miranda the lowest cell pointers are reserved for combinators, so this does not need to be encoded in the tag, but the engine still needs to do a case distinction for low values of the pointer. TIGRE also eliminates this overhead by encoding nodes as executable machine instructions. We discuss this encoding in more detail in the next section. With this change any pointer in the graph points to executable code: nodes are directly executable, and for combinators, we can insert a pointer to their implementation function. The overhead of interpreting tags is thus fully eliminated.

Our implementation still has a tag byte in each cell, but it is exclusively used by the garbage collector to keep track of free and occupied cells. The TIGRE paper provides no details on garbage collection, so we adopt the same mark-and-sweep collection approach as Miranda.

B. Executable graph nodes

Due to the elimination of tags, graph nodes must be executable x86-64 machine instructions. Executing a node is expected to *unwind the spine*: a pointer to the current node should be pushed onto the stack, after which we proceed to evaluate the cell to the left. It turns out that this can be neatly encoded as a subroutine call in both the VAX and x86-64 instruction sets.

The x86-64 `CALL` *rel32* instruction[5] modifies the processor state as follows:

1) Pushes onto the stack a pointer to the instruction directly after the `CALL`. In a regular program, this is the *return address*, the address the program should jump to after the subroutine finishes.
2) Adds *rel32* to the program counter register `%rip`. *rel32* encodes the address of the subroutine as a signed 32-bit offset from the location of the `CALL`.

We exploit this by encoding nodes as a `CALL` instruction to the left pointer, followed by the right pointer as a raw value. The memory layout of a node is illustrated in Figure 9.

When execution jumps to a node, the `CALL` instruction executes. The address of the right pointer is pushed to the stack. This is convenient because when later a combinator executes, it only needs to read the value of the right pointer (the left is only needed to unwind the stack). Execution then jumps to the address of the left cell pointer, which is either another node or the code for a combinator. The `CALL` instruction and 64-bit right pointer take up 13 bytes. We add an extra byte to store the tag (only one bit is currently used) and 2 bytes of padding so the full node is 16 bytes wide. Having 16-byte wide nodes is convenient because Intel processors can execute jumps faster if the target is aligned to a 16-byte boundary[6].

The simpler combinators are easily expressed as a short sequence of assembly instructions. Figure 10 shows the implementation of the I combinator.

Combinators that return a value (`LIT` or strict combinators) can be implemented by computing the value and returning it with `RET`. See for example the implementation of `LIT` in Figure 11.

A strict combinator that needs to evaluate its arguments is easier to implement in a TIGRE-style engine compared to Miranda: the implementation can strictly evaluate arguments by jumping to the argument node with a `CALL` instruction, and control will return to the combinator implementation after the argument has been evaluated and the `RET` instruction executes. The fully evaluated argument is then available in the `%rax` register. This even works from Rust code: a pointer to a node can be cast to a function pointer with an integer as a return value and invoked normally.

To our surprise, we did not find the TIGRE-style engine more difficult to implement than the Miranda version, mainly because the reduction process happens to map rather naturally to subroutine calls. We discuss the main challenges encountered during the implementation in the next section.

C. Implementation issues

1) Writeable and executable memory: For our reduction engine to work, nodes must be stored in executable memory. Modern x86-64 CPUs support the NX-bit [18], which allows the operating system to mark regions of the memory as non-executable, triggering an exception if a program jumps to a page with the NX-bit set. The NX-bit is typically enabled by default on dynamically allocated heap memory. To allocate memory without the NX-bit set on Linux, we allocate memory for the graph using `mmap`[7] with the `PROT_EXEC` (to allow execution) and `PROT_WRITE` (to allow writing) flags. Some operating systems do not allow unprivileged programs to set `PROT_EXEC` and `PROT_WRITE` at the same time. One solution to this is to flip the memory protection settings from executable to writeable, make the changes, and flip back to executable and continue execution [19]. We anticipate this would be prohibitively expensive for our engine: After a combinator executes, it updates the graph with the new value, so that if the same value is used in a different subgraph, it is only evaluated once (just as Miranda does). The engine would have to issue two `mprotect` system calls at every reduction step, a heavy performance penalty. Luckily, the author's Linux

[5]https://c9x.me/x86/html/file_module_x86_id_26.html

[6]https://groups.google.com/g/golang-nuts/c/dhGbgC1pAmA/m/Rcqwcd5mGmoJ

[7]https://man7.org/linux/man-pages/man2/mmap.2.html

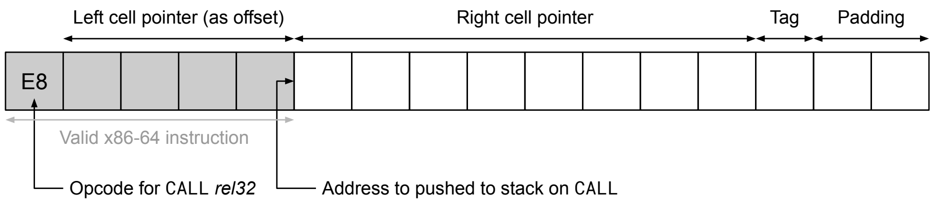

Fig. 9. Memory layout of a node in the TIGRE-style engine. Each square represents a byte of memory.

```
comb_I:
    ; Pop pointer to argument
    ; into rax
    pop rax
    ; Dereference and jump
    jmp [rax]
```

Fig. 10. I combinator in x86-64 assembly.

```
comb_LIT:
    ; Pop pointer to argument
    ; into rax
    pop rax
    ; Dereference and store
    ; in return value register
    mov rax, [rax]
    ; Return to caller
    ret
```

Fig. 11. LIT combinator in x86-64 assembly.

distribution does not enforce a W ^ X policy and allows the allocation of memory that is both writeable and executable.

2) Relative jumps: The CALL instruction takes as argument a 32-bit offset relative to the instruction pointer (the address of the CALL instruction), rather than a full 64-bit address. Therefore, some pointer arithmetic is required to calculate the value of this offset. More importantly, a 32-bit offset is too small to reach any target within the 64-bit address space. This is problematic for large programs where nodes may be very far apart. The programs we have used to evaluate the engine are relatively small, so we did not encounter such an issue. However, the offset may also point to an implementation of a combinator. The location of this code is determined by the linker when the engine itself is compiled, so it is difficult to influence its location in memory. To get around this, our engine registers the addresses of all combinators. Given this set of addresses, the engine can compute the *safe range*: a region of the address space from which a 32-bit offset would be sufficient to reach any combinator. When allocating memory to store the graph nodes, it finds an unmapped page within the safe range and allocates it. As an additional safeguard, all offsets are initially computed as 64-bit values, after which the engine verifies that the offset indeed fits in 32 bits.

The technique is not without its problems: for large programs (or a large set of combinators), allocation of large enough block of memory may be unsuccessful. A more robust solution would be to use trampolines: locations in memory close to nodes that contain assembly instructions to jump to locations far away in memory. This requires more bookkeeping and is therefore left as future work.

D. Stack alignment

The System-V ABI that is used on x86-64-Linux mandates that before a CALL instruction the stack is aligned to 16 bytes[8]. Our engine regularly chains CALL instructions (pushing an 8-byte return address onto the stack) until a combinator is reached, so it is impossible to guarantee statically the alignment of the stack upon a call to a combinator implementation. The entry point of each combinator that needs to call into Rust code checks the alignment of the stack right before the CALL instruction and adjusts it if necessary.

VII. Performance evaluation

Our performance evaluation consists of three parts:

- We compare our Miranda-style engine with the original Miranda, to demonstrate that our implementation is competitive.
- We compare our TIGRE-style engine to our Miranda-style engine, to see if the techniques described in [5] are viable on the X86 architecture.
- We compare the traditional bracket abstraction and the various semantic translation algorithms from [11] by compiling the same source programs with each translation algorithm and measuring the runtime on our Miranda-style engine, to see if the semantic translation algorithms produce compiled programs that execute faster.

The program used in these benchmarks is a recursive definition of a function to compute elements of the Fibonacci sequence. Figure 12 shows the definition of the function in the Miranda language (for use with the original Miranda), as well as a definition in the LISP-like language used by our tools. The two definitions are identical, all differences are syntactical. In our benchmarks, we vary the value of parameter n between 10 and 20 to show how the execution time changes as we increase the number of recursive calls (and therefore the number of operations performed).

[8]https://c9x.me/compile/doc/abi.html

```
-- Miranda
fib n = n,                        if n < 2
      = fib(n - 1) + fib(n - 2),  otherwise

-- Superg (LISP-like)
(defun fib (n)
   (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Fig. 12. Recursive function to compute the n-th element of the Fibonacci sequence.

We use the Criterion [20] benchmarking library to measure the performance of our engines and Miranda. The original Miranda does not capture performance metrics, so we apply a small patch to the source code to output the time elapsed during a reduction operation. The 20-line patch and instructions on how to apply it are included with the source code for our implementation. We use Criterion's default measurement strategy: each benchmark runs repeatedly for 5 seconds as a warmup, after which 100 samples of the execution time are collected. Finally, an estimate of the average execution time is computed using linear regression over the samples.

Benchmarks were run on an otherwise quiescent laptop equipped with an x86-64 AMD CPU (Ryzen 7 4800U), running Debian 11 (Bullseye). The original Miranda was compiled using GCC 10.2.1 without any optimization flags, to avoid breaking the garbage collector as discussed in Section V. All other code is written in Rust and compiled using the optimizing `release` profile (`rustc` version 1.65.0).

A. Comparison with original Miranda

Figure 13 shows the performance of our Miranda-style engine with the traditional bracket abstraction compared to the original Miranda [8].

We observe that our Miranda-style engine is more than twice as fast as the original Miranda. While we have not attempted an in-depth evaluation of the performance differences, we can point to two likely reasons why our implementation performs better on this benchmark:

- Miranda supports (exclusively) arbitrary precision integers. They are implemented by the high-performance GNU MP [17] library but are still slower than the native 32-bit machine integers that our engine uses.
- As mentioned in Section V, the original Miranda must be compiled with optimizations disabled. Without these optimizations, the compiler is likely to generate a slower implementation of key functions.

As the points above indicate, the comparison of the two systems is not fair. Therefore, we do not claim that our implementation is strictly better than the original, but rather that it is a reasonably fast reference implementation of a combinator graph reduction engine. The other benchmarks in this section all use the Miranda-style engine with bracket abstraction as a reference point for performance, based on which we can judge if the proposed changes are beneficial.

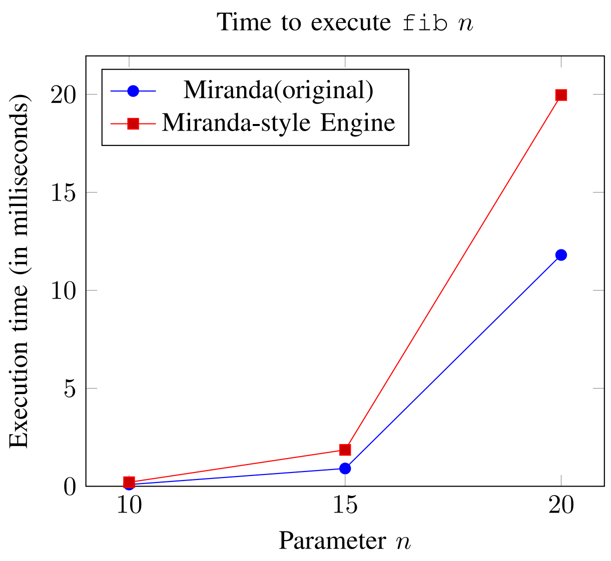

Fig. 13. Time to execute the Fibonacci benchmark program on the original Miranda and our Miranda-style engine. A lower value on the Y-axis is better.

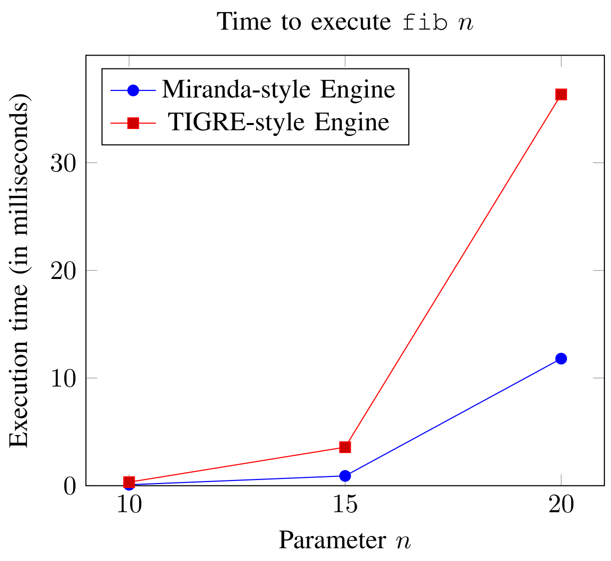

Fig. 14. Time to execute the Fibonacci benchmark program on Miranda-style and TIGRE-style engines. A lower value on the Y-axis is better.

B. TIGRE-style Engine

To benchmark the TIGRE-style engine we use the same Fibonacci program and bracket abstraction algorithm, and plot the execution time against our Miranda-style engine. Both engines are configured to use the same compiler, so they will execute identical combinator expressions. The results can be seen in Figure 14.

Contrary to our expectations, the TIGRE-style engine is significantly slower than the reference Miranda-style engine. Based on the flame graphs [21] of the two engines, available in Appendix A, we can conclude that the S combinator is significantly slower on this engine.

We used Valgrind's cachegrind [22] tool to inspect the `make_s` function and found that its execution time is dominated by data write operation to update the graph. While the TIGRE-style engine executes fewer instructions (22M vs. 79M) and performs fewer data accesses (13M vs. 49M), it has a much higher cache miss rate for writes (3% vs. 0.5%).

We hypothesize that updates to nodes in the graph are slow in the TIGRE-style engine because those same nodes have been loaded into the instruction cache, and therefore trigger an expensive cache invalidation to maintain coherency between the instruction and data caches. The node to be updated is part of the current spine, so its `CALL` instruction will have been executed before the update, loading it into the instruction cache. When a write occurs to this node, the CPU must conservatively invalidate or update that cached value, or it would risk executing a stale instruction if the code were to jump back to that address. Writes to addresses in the instruction cache are frequent in the TIGRE-style engine, but in most software, they are very rare. On some processor architectures, like ARM, programs must manually clear the instruction cache if they issue a write to a location that may be cached [23].

It appears that the TIGRE approach to implementing combinator graph reduction is a poor fit for modern architectures due to the split between instruction and data caches. Other innovations from TIGRE, such as `LIT` combinators and removing tags, may still be beneficial. In particular, it may be possible to get the benefits of TIGRE without cache coherency issues by storing jump addresses rather than call instructions in nodes. Rather than jump directly to a node and execute it, the runtime could load the jump address and push the argument pointer to the stack normally, then jump to the address instead. The implementation of such an engine is left as future work.

C. *Kiselyov's algorithms*

To compare the semantic translation algorithms from [11] with each other, we again use the Fibonacci test program and execute all compiled programs on the Miranda-style engine. The bracket abstraction algorithm has been added as a reference point. The results can be seen in Figure 15.

All of the algorithms presented in [11] produce more efficient code than the reference bracket abstraction algorithm. The lazy$_\eta$ compilation scheme produces the best code for the Fibonacci program. For many of the test programs in [11] it was shown to generate the shortest programs of all the algorithms presented there. This is also true for the Fibonacci program, as shown in Figure 16.

While the linear strategy produces slightly more compact code than strict and lazy, there is no clear performance improvement. Our small benchmark programs only require bulk combinators for up to 2 arguments, so the linear strategy cannot win much in terms of code size. We expect that for certain worst-case inputs, as given in [11], linear may still prove useful.

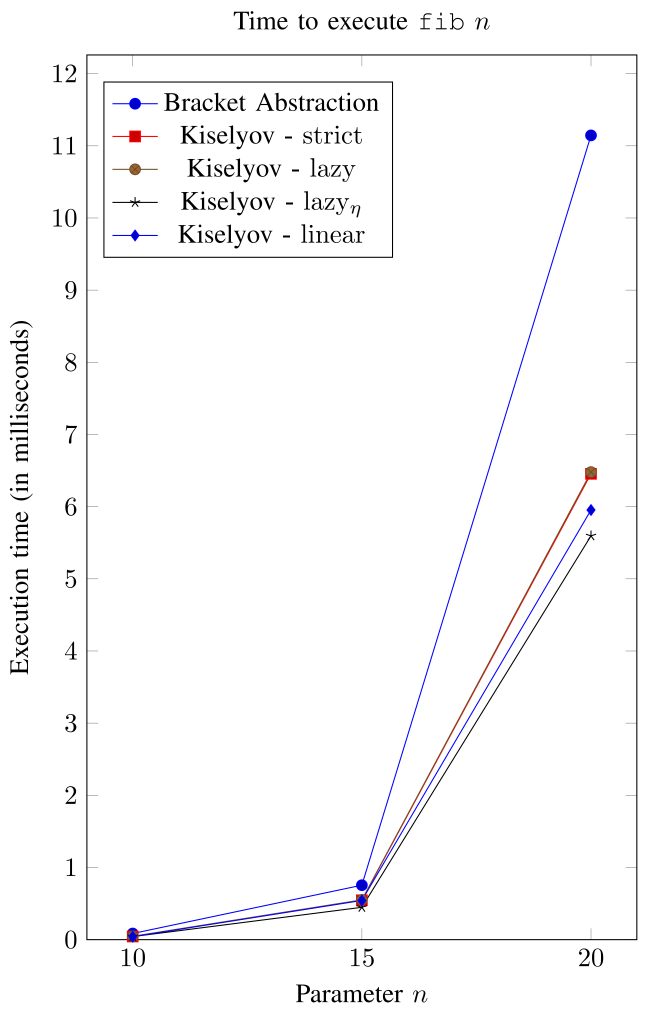

Fig. 15. Time to execute the Fibonacci benchmark program compiled with the algorithms from [11]. A lower value on the Y-axis is better.

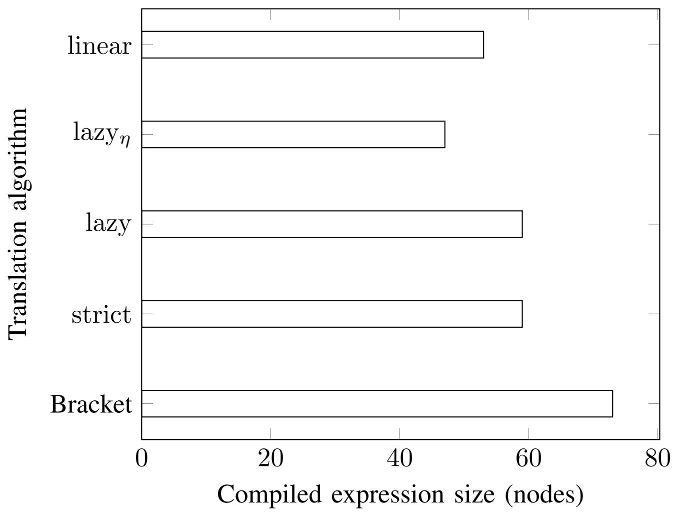

Fig. 16. Size of combinator expression generated for the Fibonacci program by different translation algorithm

It comes as no surprise that the shortest translation is also the fastest to execute, but it is worth noting that while the Bracket abstraction translation is only 55% larger than the lazy_η one, it runs 94% slower. We see a similar pattern for the Ackermann function, for which results may be found in appendix C. This suggests that while the semantic translation algorithms from [11] reduce the size of the combinator expression, they reduce runtime even further.

VIII. Related work

A. Super-combinators

Super-combinators [24] are the basis of many lazy functional runtimes [2]–[4], [6] developed after Miranda. A supercombinator is a lambda expression of the form $\lambda x_1.\lambda x_2....\lambda x_n.E$, where E has no free variables, and any lambda abstraction in E is also a supercombinator [10]. Any program in the lambda calculus may be transformed into supercombinators by *lifting* free variables. For example, we may consider $\lambda f.f\ (\lambda x.f\ x\ 2)$, which is not a supercombinator, because f is free in the inner lambda. To make this expression a super-combinator, we can pass f explicitly:

$$\lambda f.f\ ((\lambda f.\lambda x.f\ x\ 2)\ f)$$

This transformation is useful because the output of a supercombinator depends entirely on the value of its inputs, and thus its implementation may be described as a fixed set of (machine) instructions.

Programs translated into super-combinators have a code size that is linear in the size of the input in the worst case, an improvement over the worst-case quadratic size of bracket abstraction.

Perhaps more importantly, the reduction steps for a program based on super-combinators are larger. A downside is that this makes the execution less lazy, but an important advantage is that the larger step size makes the overhead of the reduction loop smaller. The small step size is an often noted issue with combinator reduction engines [10], and a major reason why they were largely abandoned.

B. Haskell

Haskell is without a doubt the most popular lazy functional programming language today. It too has its roots in the supercombinator approach. The popular Glasgow Haskell Compiler is based on the Spineless Tagless G-Machine [6], which in turn is based on the Spineless G-Machine [4] and G-Machine [2]. GHC compiles super-combinators into machine code ahead of time, which makes the resulting binaries self-contained and highly efficient. In future work, we will compare the performance of our runtime against that of GHC.

C. SIMD Combinator Reduction

In his talk *Combinator Revisited* [25], Kmett hypothesizes the use of SIMD instructions to perform combinator reductions. The rationale is that combinators have a relatively simple implementation, and SIMD instructions are getting more and more powerful, so potentially they can be used to implement the reduction of multiple combinators in a single instruction. Assuming the reduction of a single combinator uses 64-bit wide values, the same routine implemented with AVX-512 instructions could be used to perform as many as 8 reductions in parallel on a single core. An eightfold improvement in performance is certainly promising, although it is unclear what level of parallelism a typical program may allow for. A work-in-progress version of the compiler is available at https://github.com/ekmett/coda.

IX. Conclusion

Our objective with this work was to revive research around Combinator Graph Reduction by improving the performance. We used a recently proposed semantic translation algorithm to compile lambda calculus programs into smaller combinator expressions [11] and have implemented modern combinator graph reduction engines based on the designs of Miranda [1] and TIGRE [5].

We have developed a reference implementation of a combinator graph reduction engine based on the now open source Miranda [1]. Our implementation comprises under 1000 lines of Rust code and outperforms the original Miranda engine on our benchmarks.

Our port of the TIGRE [5] engine runs natively on stock x86-64 hardware and executes our benchmark program with fewer instructions and data accesses than our Miranda-style engine. Unfortunately, its performance still lags behind the Miranda-style engine because its inherent frequent writes to executable code sections trigger expensive invalidations of the instruction cache. We conclude that the TIGRE design is a poor fit for architectures with split instruction and data caches, though some concepts like `LIT` combinators and tag elimination may still prove effective.

Applying Kiselyov's translation algorithms [11] to our benchmarks was a resounding success. All proposed variations of the semantic translation produced a combinator expression that was smaller than the one generated by the reference bracket abstraction algorithm. Moreover, the effect of the smaller code size was amplified at execution time, with the smallest program running nearly twice as fast as the reference program.

There is ample room for improvement both on the compilation and the runtime side. Our linear translation, like Kiselyov's, is based on the strict translation rather than the lazy or lazy_η translation schemes. Combining bulk combinators with lazy weakening could further reduce the size of compiled programs, and therefore also improve performance.

On the runtime side, ideas from TIGRE could be selectively integrated into the Miranda-style engine to improve its performance, such as `LIT` combinators or eliminating tags.

Our current runtime is single-threaded: it always reduces one combinator at a time. However, reducing multiple combinators in parallel still leads to a correct execution, so it may be possible to run programs faster by evaluating separate parts of the graph in parallel on different cores. As suggested by

Kmett [25], it may even be possible to use SIMD instructions to reduce multiple combinators in parallel on a *single* core.

Our garbage collector and the one used by Miranda is a simple mark-and-sweep collector. While our benchmark programs do not run long enough to exhaust the available memory, we expect that for real programs a faster garbage collector could improve performance significantly. In particular, a concurrent garbage collector [26] could prevent execution stalls during collection time. We hypothesize that a limited form of reference counting would also be beneficial: the S combinator (B and C are very similar) currently requires the allocation of two new nodes when it executes, and it also drops a reference to two other nodes. With reference counting we could determine if those two nodes are not needed elsewhere in the graph, and reuse them, eliminating the creation of garbage.

Source code for our translation algorithms and runtimes, as well as associated tests and benchmarks, are available at https://github.com/wildarch/mono/tree/main/experiments/superg.

References

[1] D. A. Turner, "A new implementation technique for applicative languages," *Software: Practice and Experience*, vol. 9, no. 1, pp. 31–49, 1979, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380090105. [Online]. Available: http://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380090105

[2] R. B. Kieburtz, "The G-machine: A fast, graph-reduction evaluator," in *Functional Programming Languages and Computer Architecture*, ser. Lecture Notes in Computer Science, J.-P. Jouannaud, Ed. Berlin, Heidelberg: Springer, 1985, pp. 400–413.

[3] J. Fairbairn and S. Wray, "Tim: A simple, lazy abstract machine to execute supercombinators," in *Functional Programming Languages and Computer Architecture*, ser. Lecture Notes in Computer Science, G. Kahn, Ed. Berlin, Heidelberg: Springer, 1987, pp. 34–45.

[4] G. Burn, S. Peyton Jones, and J. Robson, "The spineless G-machine," pp. 244–258, Jan. 1988.

[5] P. J. Koopman and P. Lee, "A fresh look at combinator graph reduction," *ACM SIGPLAN Notices*, vol. 24, no. 7, pp. 110–119, Jun. 1989. [Online]. Available: http://doi.org/10.1145/74818.74828

[6] S. L. P. Jones, "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine Version 2.5," p. 87.

[7] D. A. Turner, "Miranda: A non-strict functional language with polymorphic types," in *Functional Programming Languages and Computer Architecture*, ser. Lecture Notes in Computer Science, J.-P. Jouannaud, Ed. Berlin, Heidelberg: Springer, 1985, pp. 1–16.

[8] D. Turner, "Open Sourcing Miranda," Mar. 2021. [Online]. Available: http://codesync.global/media/open-sourcing-miranda-david-turner-code-mesh-v-2020-codemeshv2020/

[9] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of Haskell: being lazy with class," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ser. HOPL III. New York, NY, USA: Association for Computing Machinery, Jun. 2007, pp. 12–1–12–55. [Online]. Available: http://doi.org/10.1145/1238844.1238856

[10] S. L. Peyton Jones, *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. USA: Prentice-Hall, Inc., 1987.

[11] O. Kiselyov, "lambda to SKI, Semantically," in *Functional and Logic Programming*, ser. Lecture Notes in Computer Science, J. P. Gallagher and M. Sulzmann, Eds. Cham: Springer International Publishing, 2018, pp. 33–50.

[12] P. Koopman, "An Architecture for Combinator Graph Reduction (TIGRE)," 1992. [Online]. Available: http://users.ece.cmu.edu/~koopman/tigre/

[13] R. Rojas, "A Tutorial Introduction to the Lambda Calculus," Mar. 2015, arXiv:1503.09060 [cs]. [Online]. Available: http://arxiv.org/abs/1503.09060

[14] N. G. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem," *Indagationes Mathematicae (Proceedings)*, vol. 75, no. 5, pp. 381–392, Jan. 1972. [Online]. Available: https://www.sciencedirect.com/science/article/pii/1385725872900340

[15] S. Peyton Jones and D. Lester, "Implementing functional languages: a tutorial," Apr. 2000.

[16] D. A. Turner, "Another Algorithm for Bracket Abstraction," *The Journal of Symbolic Logic*, vol. 44, no. 2, pp. 267–270, 1979, publisher: [Association for Symbolic Logic, Cambridge University Press]. [Online]. Available: http://www.jstor.org/stable/2273733

[17] T. Granlund and G. D. Team, *GNU MP 6.0 Multiple Precision Arithmetic Library*. London, GBR: Samurai Media Limited, 2015.

[18] L. Paulson, "New chips stop buffer overflow attacks," *Computer*, vol. 37, no. 10, pp. 28–, Oct. 2004, conference Name: Computer.

[19] J. de Mooij, "W^X JIT-code enabled in Firefox," Dec. 2015. [Online]. Available: https://jandemooij.nl/blog/wx-jit-code-enabled-in-firefox/

[20] B. Heisler, "Criterion.rs," Jan. 2023, original-date: 2014-05-26T14:14:22Z. [Online]. Available: https://github.com/bheisler/criterion.rs

[21] B. Gregg, "The flame graph," *Communications of the ACM*, vol. 59, no. 6, pp. 48–57, May 2016. [Online]. Available: https://doi.org/10.1145/2909476

[22] N. Nethercote, "Dynamic binary analysis and instrumentation," A dissertation submitted for the degree of Doctor of Philosophy, University of Cambridge, November 2004, it's related with Valgrind framework. fulltitle: Dynamic Binary Analysis and Instrumentation or Building Tools is Easy Parts of the research presented in this dissertation have been previously published or presented in the following papers. Nicholas Nethercote and Alan Mycroft. The cache behaviour of large lazy functional programs on stock hardware. In Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002), pages 44–55, Berlin, Germany, July 2002. Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. In Proceedings of the Third Workshop on Runtime Verification (RV'03), Boulder, Colorado, USA, July 2003. Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In Pro- ceedings of the Third Workshop on Runtime Verification (RV'03), Boulder, Colorado, USA, July 2003. Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recom- piling. In Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Com- puting Environments for Memory Management (SPACE 2004), Venice, Italy, January 2004. [Online]. Available: http://valgrind.org/docs/phd2004.pdf

[23] J. Bramley, "Caches and Self Modifying Code - Architectures and Processors blog - Arm Community blogs - Arm Community," Sep. 2013. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/caches-and-self-modifying-code

[24] R. J. M. Hughes, "Super-combinators a new implementation method for applicative languages," in *Proceedings of the 1982 ACM symposium on LISP and functional programming*, ser. LFP '82. New York, NY, USA: Association for Computing Machinery, Aug. 1982, pp. 1–10. [Online]. Available: https://doi.org/10.1145/800068.802129

[25] E. Kmett, "Combinators Revisited," Jul. 2018. [Online]. Available: https://www.youtube.com/watch?v=zhj_tUMwTe0

[26] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, 1st ed. Chapman & Hall/CRC, 2011.

Appendix

A. Flame graphs

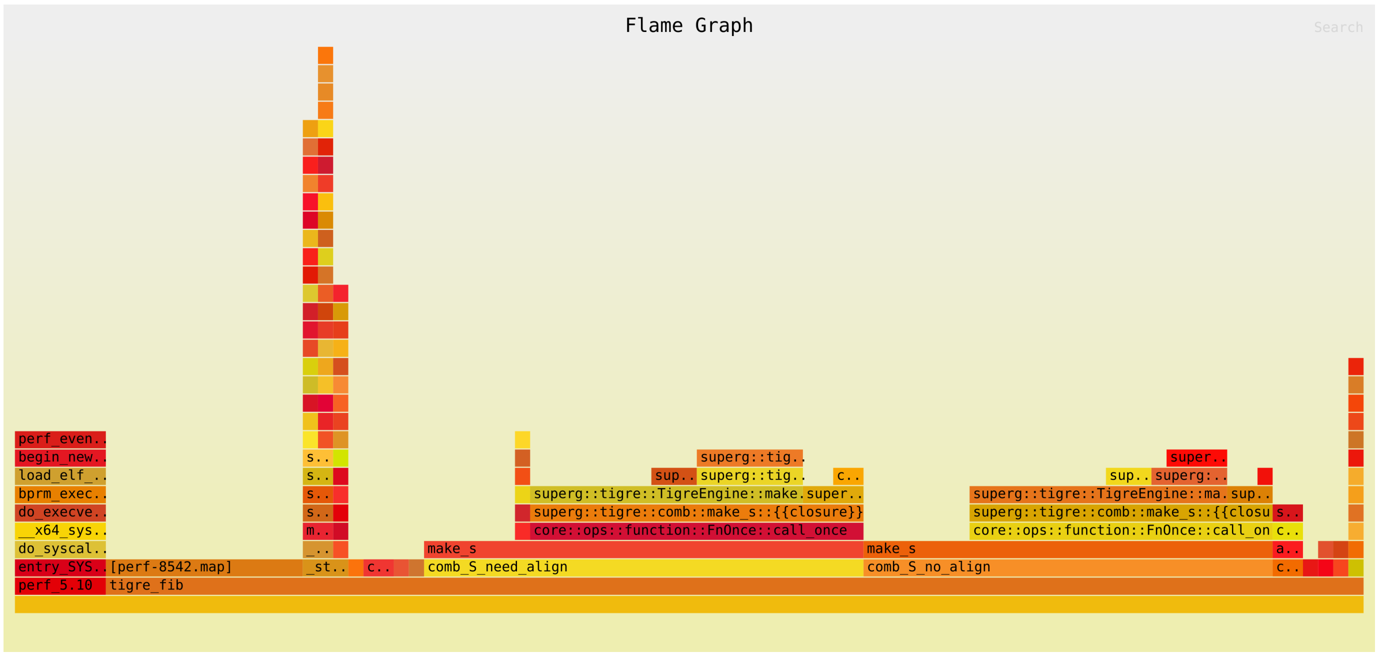

Fig. 17. Flame graph for $fib(20)$ on Miranda-style engine.

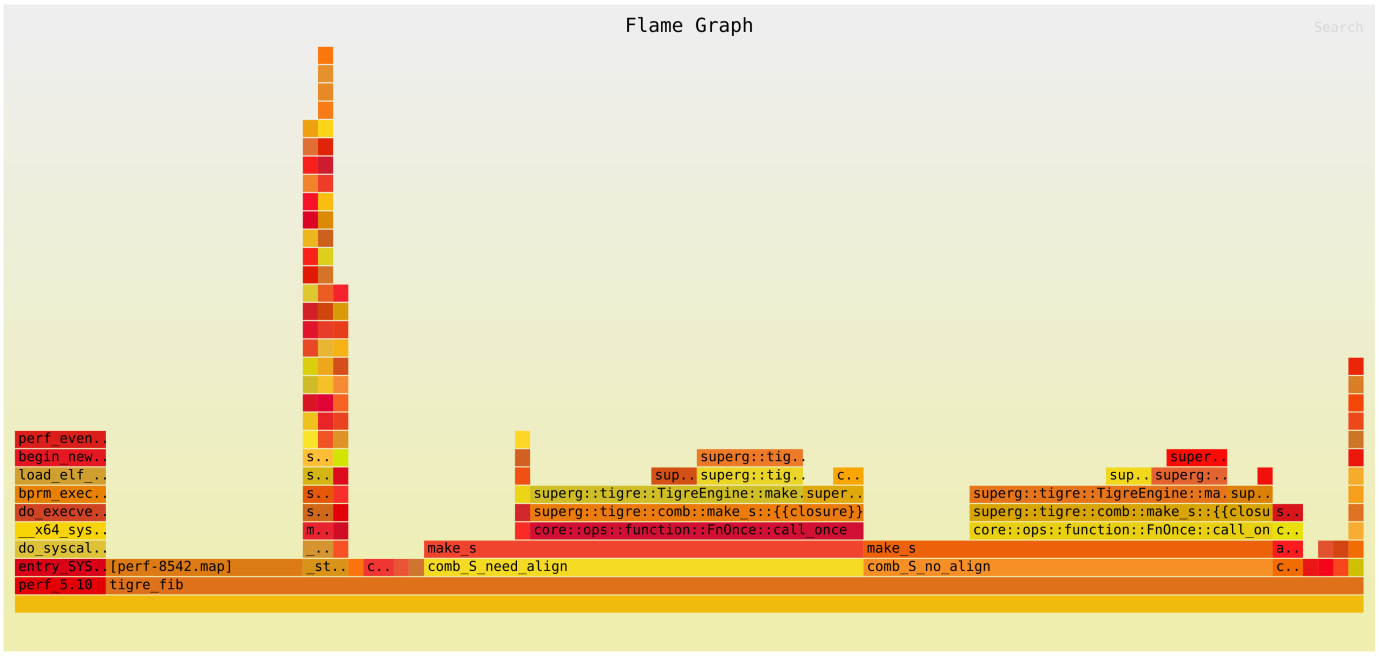

Fig. 18. Flame graph for $fib(20)$ on TIGRE-style engine.

B. Semantic translation definitions

.

$$\begin{array}{llll}
\text{lazy} & z & = & t \models I \\
\text{lazy} & s\ e & = & \Gamma, f \models c \\
 & & & \text{where } (\Gamma \models c) := \text{lazy}\, e \\
 & & & \\
\text{lazy} & \lambda e & = & \emptyset \models K\ c \\
 & & & \text{where } (\emptyset \models c) := \text{lazy}\, e \\
\text{lazy} & \lambda e & = & (\emptyset \models K) \coprod (\Gamma \models c) \\
 & & & \text{where } (\Gamma, f \models c) := \text{lazy}\, e \\
\text{lazy} & \lambda e & = & \Gamma \models K\ c \\
 & & & \text{where } (\Gamma, t \models c) := \text{lazy}\, e \\
 & & & \\
\text{lazy} & e_1\ e_2 & = & (\Gamma \models ((\Gamma_1 \models c_1) \coprod (\Gamma_2 \models c_2)) \\
 & & & \text{where } (\Gamma_1 \models c_1) := \text{lazy}\, e_1 \\
 & & & \text{where } (\Gamma_2 \models c_2) := \text{lazy}\, e_2 \\
 & & & \text{where } \Gamma := \Gamma_1 \sqcup \Gamma_2
\end{array}$$

Fig. 19. Compilation function with Lazy Weakening.

$$\begin{array}{lllll}
\Gamma_1 & \sqcup & \emptyset & = & \Gamma_1 \\
\emptyset & \sqcup & \Gamma_2 & = & \Gamma_2 \\
\Gamma_1, t & \sqcup & \Gamma_2, \_ & = & (\Gamma_1 \sqcup \Gamma_2), t \\
\Gamma_1, \_ & \sqcup & \Gamma_2, t & = & (\Gamma_1 \sqcup \Gamma_2), t \\
\Gamma_1, f & \sqcup & \Gamma_2, f & = & (\Gamma_1 \sqcup \Gamma_2), f
\end{array}$$

Fig. 20. Context merge function for Lazy Weakening.

$$\begin{array}{llll}
(\emptyset \models c_1) & \coprod (\emptyset \models c_2) & = & c_1\ c_2 \\
 & & & \\
(\emptyset \models c_1) & \coprod (\Gamma_2, t \models c_2) & = & (\emptyset \models Bc_1) \coprod (\Gamma_2 \models c_2) \\
 & & & \\
(\Gamma_1, t \models c_1) & \coprod (\emptyset \models c_2) & = & (\emptyset \models CCc_2) \coprod (\Gamma \models c_1) \\
 & & & \\
(\Gamma_1, t \models c_1) & \coprod (\Gamma_2, t \models c_2) & = & (\Gamma_1 \models (s \coprod l)) \coprod r) \\
 & & & \text{where } s := (\emptyset \models S) \\
 & & & \text{where } l := (\Gamma_1 \models c_1) \\
 & & & \text{where } r := (\Gamma_2 \models c_2) \\
 & & & \\
(\Gamma_1, f \models c_1) & \coprod (\Gamma_2, f \models c_2) & = & (\Gamma_1 \models c_1) \coprod (\Gamma_2 \models c_2) \\
 & & & \\
(\Gamma_1, f \models c_1) & \coprod (\Gamma_2, t \models c_2) & = & (\Gamma_1 \models (b \coprod l)) \coprod r) \\
 & & & \text{where } b := (\emptyset \models B) \\
 & & & \text{where } l := (\Gamma_1 \models c_1) \\
 & & & \text{where } r := (\Gamma_2 \models c_2) \\
 & & & \\
(\Gamma_1, t \models c_1) & \coprod (\Gamma_2, f \models c_2) & = & (\Gamma_1 \models (c \coprod l)) \coprod r) \\
 & & & \text{where } c := (\emptyset \models C) \\
 & & & \text{where } l := (\Gamma_1 \models c_1) \\
 & & & \text{where } r := (\Gamma_2 \models c_2) \\
 & & & \\
(\emptyset \models c_1) & \coprod (\Gamma_2, f \models c_2) & = & (\emptyset \models c_1) \coprod (\Gamma_2 \models c_2) \\
 & & & \\
(\Gamma_1, f \models c_1) & \coprod (\emptyset \models c_2) & = & (\Gamma_1 \models c_1) \coprod (\emptyset \models c_2)
\end{array}$$

Fig. 21. Semantic function with Lazy Weakening.

$$
\begin{array}{llcl}
\text{lazy}_\eta & z & = & \emptyset \models V \\
\text{lazy}_\eta & s\ e & = & \Gamma, f \models c \\
 & & & \text{where } (\Gamma \models c) := \text{lazy}_\eta\, e \\
\\
\text{lazy}_\eta & \lambda e & = & \emptyset \models I \\
 & & & \text{where } (\emptyset \models V) := \text{lazy}_\eta\, e \\
\text{lazy}_\eta & \lambda e & = & \emptyset \models K\ c \\
 & & & \text{where } (\emptyset \models c) := \text{lazy}_\eta\, e \\
\text{lazy}_\eta & \lambda e & = & (\emptyset \models K) \coprod_\eta (\Gamma \models c) \\
 & & & \text{where } (\Gamma, f \models c) := \text{lazy}_\eta\, e \\
\text{lazy}_\eta & \lambda e & = & \Gamma \models K\ c \\
 & & & \text{where } (\Gamma, t \models c) := \text{lazy}_\eta\, e \\
\\
\text{lazy}_\eta & e_1\ e_2 & = & (\Gamma \models ((\Gamma_1 \models c_1) \coprod_\eta (\Gamma_2 \models c_2)) \\
 & & & \text{where } (\Gamma_1 \models c_1) := \text{lazy}_\eta\, e_1 \\
 & & & \text{where } (\Gamma_2 \models c_2) := \text{lazy}_\eta\, e_2 \\
 & & & \text{where } \Gamma := \Gamma_1 \sqcup_\eta \Gamma_2
\end{array}
$$

Fig. 22. Compilation function with Eta optimization.

$$
\begin{array}{lllcl}
(\Gamma_1, \_ \models c_1) & \sqcup_\eta & (\emptyset \models V) & = & \Gamma_1, t \\
(\emptyset \models V) & \sqcup_\eta & (\Gamma_2, \_ \models c_2) & = & \Gamma_2, t \\
(\emptyset \models c_1) & \sqcup_\eta & (\emptyset \models V) & = & t \\
(\emptyset \models V) & \sqcup_\eta & (\emptyset \models c_2) & = & t \\
(\emptyset \models V) & \sqcup_\eta & (\emptyset \models V) & = & \textbf{impossible} \\
\\
(\Gamma_1 \models c_1) & \sqcup_\eta & (\emptyset \models c_2) & = & \Gamma_1 \\
(\emptyset \models c_1) & \sqcup_\eta & (\Gamma_2 \models c_2) & = & \Gamma_2 \\
(\Gamma_1, t \models c_1) & \sqcup_\eta & (\Gamma_2, \_ \models c_2) & = & (\Gamma_1 \sqcup_\eta \Gamma_2), t \\
(\Gamma_1, \_ \models c_1) & \sqcup_\eta & (\Gamma_2, t \models c_2) & = & (\Gamma_1 \sqcup_\eta \Gamma_2), t \\
(\Gamma_1, f \models c_1) & \sqcup_\eta & (\Gamma_2, f \models c_2) & = & (\Gamma_1 \sqcup_\eta \Gamma_2), f
\end{array}
$$

Fig. 23. Context merge function for Eta optimization.

$$
\begin{array}{llll}
(\Gamma_1, f \models c_1) & \coprod_\eta & (\emptyset \models V) & = c_1 \\
(\emptyset \models V) & \coprod_\eta & (\Gamma_2, f \models c_2) & = (\emptyset \models CI) \coprod_\eta (\Gamma_2 \models c_2) \\
(\Gamma_1, t \models c_1) & \coprod_\eta & (\emptyset \models V) & = (\Gamma_1 \models (s \coprod_\eta l)) \coprod_\eta i \\
 & & & \quad \text{where } l := (\Gamma_1 \models c_1) \\
 & & & \quad \text{where } s := (\emptyset \models S) \\
 & & & \quad \text{where } i := (\emptyset \models I) \\
(\emptyset \models V) & \coprod_\eta & (\Gamma_2, t \models c_2) & = (\emptyset \models SI) \coprod_\eta (\Gamma_2 \models c_2) \\
(\emptyset \models c_1) & \coprod_\eta & (\emptyset \models V) & = c_1 \\
(\emptyset \models V) & \coprod_\eta & (\emptyset \models c_2) & = CI\ c_2 \\
(\emptyset \models V) & \coprod_\eta & (\emptyset \models V) & = \textbf{impossible} \\
\\
(\Gamma_1 \models c_1) & \coprod_\eta & (\Gamma_2 \models) & = (\Gamma_1 \models c_1) \coprod (\Gamma_2 \models)
\end{array}
$$

Fig. 24. Semantic function with Eta optimization.

$$
\begin{array}{llcl}
\text{linear} & z & = & 1 \models I \\
\text{linear} & s\ e & = & n+1 \models (0 \models K) \coprod (n \models c) \\
 & & & \text{where } (n \models c) := \text{linear}\, e \\
\\
\text{linear} & \lambda e & = & 0 \models K\ c \\
 & & & \text{where } (0 \models c) := \text{linear}\, e \\
\text{linear} & \lambda e & = & (n-1) \models K\ c \\
 & & & \text{where } (n \models c) := \text{linear}\, e \\
\\
\text{linear} & e_1\ e_2 & = & (n \models ((n_1 \models c_1) \coprod_{\text{linear}} (n_2 \models c_2)) \\
 & & & \text{where } (n_1 \models c_1) := \text{linear}\, e_1 \\
 & & & \text{where } (n_2 \models c_2) := \text{linear}\, e_2 \\
 & & & \text{where } n := \max n_1\, n_2
\end{array}
$$

Fig. 25. Compilation function for Linear algorithm

$$
\begin{array}{lll}
(0 \models c_1) \coprod (0 \models c_2) & = c_1\ c_2 \\
\\
(0 \models c_1) \coprod (n \models c_2) & = B_n\ c_1\ c_2 \\
\\
(n \models c_1) \coprod (0 \models c_2) & = C_n\ c_1\ c_2 \\
\\
(n \models c_1) \coprod (n \models c_2) & = S_n\ c_1\ c_2 \\
\\
(n_1 \models c_1) \coprod (n_2 \models c_2) & = B_{n_2 - n_1}\ (S_{n_1}\ c_1)\ c_2 \\
 & \quad \text{where } n_1 < n_2 \\
(n_1 \models c_1) \coprod (n_2 \models c_2) & = C_{n_1 - n_2}\ (B_{n_1 - n_2}\ c_1)\ c_2 \\
 & \quad \text{where } n_1 > n_2
\end{array}
$$

Fig. 26. Semantic function for Linear algorithm

C. Ackermann benchmark results

```
(defun ack (x z) (if (= x 0)
                     (+ z 1)
                     (if (= z 0)
                         (ack (- x 1) 1)
                         (ack (- x 1) (ack x (- z 1))))))
```

Fig. 27. Ackermann's function.

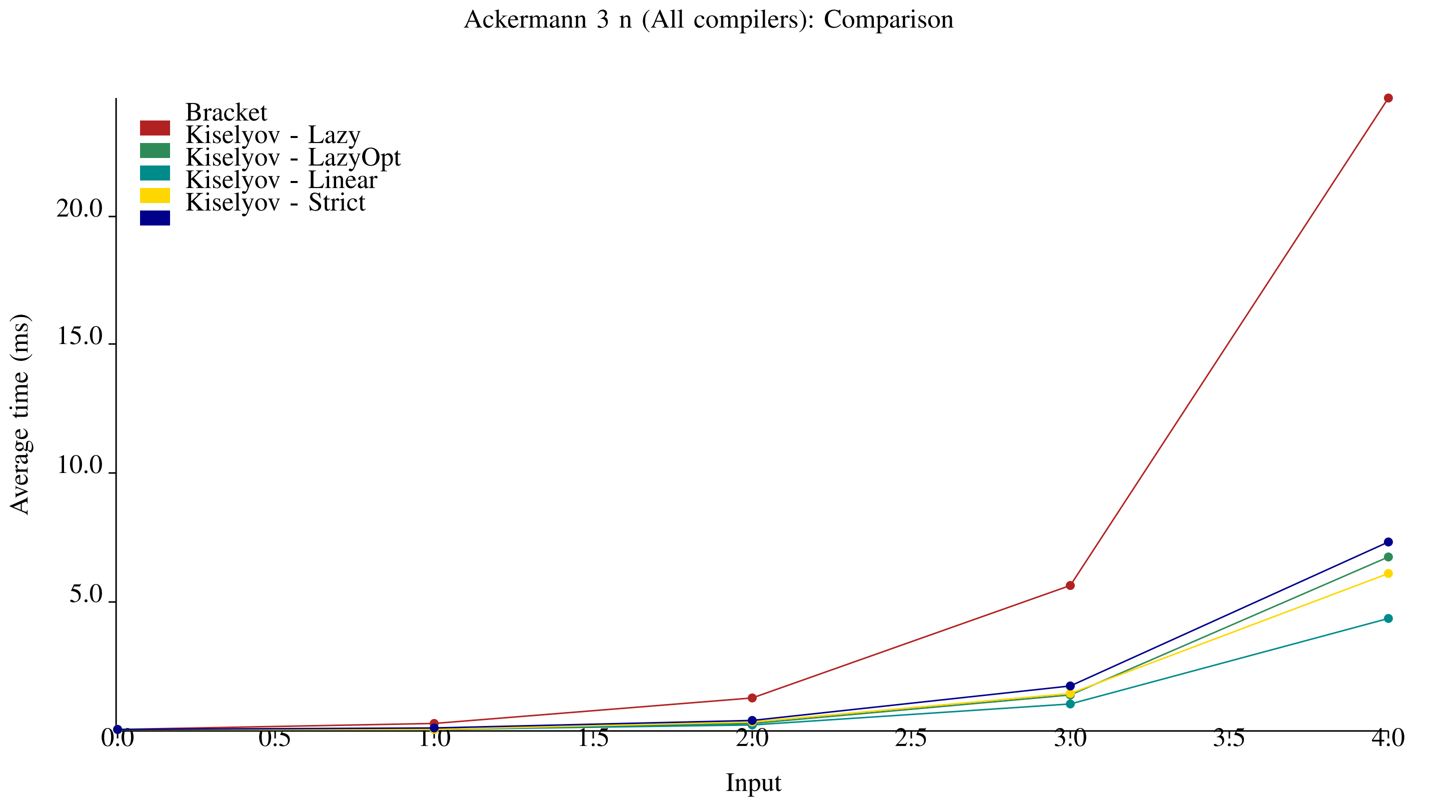

Fig. 28. Time to execute the Ackermann benchmark program compiled with the algorithms from [11]. A lower value on the Y-axis is better.

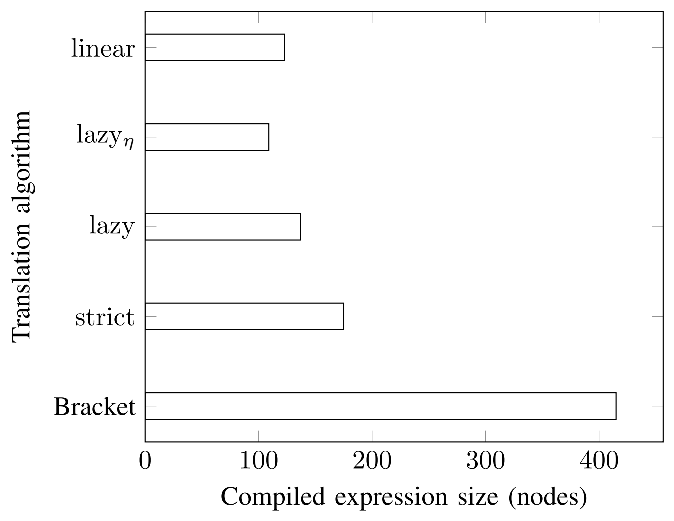

Fig. 29. Size of combinator expression generated for the Ackermann program by different translation algorithm