Tail Recursion Modulo Context – An Equational Approach

Microsoft Technical Report, MSR-TR-2022-18, 2022-07-07 (v1)

DAAN LEIJEN, Microsoft Research, USA
ANTON LORENZEN, University of Bonn, Germany

The tail-recursion modulo *cons* transformation can rewrite functions that are not quite tail-recursive into a tail-recursive form that can be executed efficiently. In this article we generalize tail recursion modulo *cons* (TRMc) to modulo *contexts* (TRMC), and calculate a general TRMC algorithm from its specification. We can instantiate our general algorithm by providing an implementation of application and composition on abstract contexts and showing that our *context laws* hold. We provide some known instantiations of TRMC, namely modulo *evaluation contexts* (CPS), and *associative operations*, and novel instantiations as suggested by our generic approach, such as *defunctionalized* evaluation contexts, *monoids*, *semirings*, and *exponents*. We study the modulo *cons* instantiation in particular and prove that an instantiation using Minamide's hole calculus is sound. We also calculate a second instantiation in terms of the Perceus heap semantics to precisely reason about the soundness of in-place update. While all previous approaches to TRMc fail in the presence of non-linear control (for example induced by call/cc, shift/reset or algebraic effect handlers), we can elegantly extend the heap semantics to a hybrid approach which dynamically adapts to non-linear control flow. We have a full implementation of hybrid TRMc in the Koka language and our benchmark shows the TRMc transformed functions are always as fast or faster than using manual alternatives.

1 INTRODUCTION

The tail-recursion modulo *cons* (TRMc) transformation can rewrite functions that are not quite tail-recursive into a tail-recursive form that can be executed efficiently. This transformation was described already in the early 70's by Risch (1973) and Friedman and Wise (1975), and more recently studied by Bour et al. (2021) in the context of OCaml. The prototypical example of a function that can be transformed this way is the `map` function which applies a function to every element of a list:

```
fun map( xs : list<a>, f : a -> e b ) : e list<b>
  match xs
    Cons(x,xx) -> Cons( f(x), map(xx,f) )
    Nil        -> Nil
```

We can see that the recursive call to `map` is behind a constructor, and thus `map` as written is not tail-recursive and uses stack space linear in the length of the list. Of course, it is well known that we can rewrite `map` by hand into a tail-recursive form by using an extra accumulating argument, but this comes at the cost of losing the simplicity of the original definition.

The TRMc transformation can automatically transform a function like `map` to a tail-recursive variant, but also improves on the efficiency of the manual version by using in-place updates on the accumulation argument. In previous work (Risch, 1973; Friedman and Wise, 1975; Bour et al., 2021), TRMc algorithms are given but all fall short of showing why these are correct, or provide particular insight in what other transformations may be possible. In this article we generalize tail recursion modulo *cons* (TRMc) to modulo *contexts* (TRMC), and try to bring the general principles out of the shadows of particular implementations and into the light of equational reasoning.

- Inspired by the elegance of program calculation as pioneered by Richard Bird (1984), Gibbons (2022), Hutton (2021), Meertens (1986), and many others, we take an equational approach where we *calculate* a general tail-recursion modulo context transformation from its specification and two general *context laws*. The resulting generic algorithm is concise (with only 4 equations) and independent of any particular instantiation of the abstract contexts as long as their operations satisfy the context laws (Section 3).
- We can instantiate the algorithm by providing an implementation of application and composition on abstract contexts, and show that these satisfy the context laws. In Section 4 we provide known

instantiations of TRMC, namely modulo *evaluation contexts* (CPS), and modulo *associative operations*, and show that those instances satisfy the context laws. We then proceed to show various novel instantiations as suggested by our generic approach, namely modulo *defunctionalized* evaluation contexts, modulo *monoids*, modulo *semirings*, and modulo *exponents*.

- In Section 5 we turn to the most important instance in practice, modulo *cons*. We show how we can instantiate our operations to the hole calculus of Minamide (1998), and that this satisfies the context laws and the imposed linear typing discipline. This gives us an elegant and sound in-place updating characterization of TRMc where the in-place update is hidden behind a purely functional (linear) interface.
- This is still somewhat unsatisfying as it does not provide insight in the actual in-place mutation as such implementation is only alluded to in prose (Minamide, 1998). We proceed by giving a second instantiation of modulo *cons* where we target the heap semantics of Reinking, Xie et al. (2021) to be able to reason explicitly about the heap and in-place mutation. Just like we could calculate the generic TRMC translation from its specification, we again *calculate* the efficient in-place updating versions for context application and composition from the abstract context laws. These calculated reductions are the exact implementation as used in our implementation in the Koka compiler.
- A well-known problem with the modulo *cons* transformation is that the efficient in-place mutating implementation fails if the semantics is extended with non-local control operations, like `call/cc`, `shift/reset` (Shan, 2007), or algebraic effect handlers (Plotkin and Power, 2003; Plotkin and Pretnar, 2009). This is in particular troublesome for a language like Koka which relies foundationally on algebraic effect handlers (Leijen, 2017; Xie and Leijen, 2021). In Section 5.3 we show two novel solutions to this: The general approach generates two versions for each TRMc translation and chooses at runtime the appropiate version depending if non-linear control is possible. This duplicates code though, and may be too pessimistic where the slow version is used even if no non-linear control actually occurs. Suggested by our heap semantics, we can do better though – in the *hybrid* approach we rely on the precise reference counts (Reinking, Xie et al., 2021), together with runtime support for *context paths*. This way we can efficiently detect at runtime if a context is unique, and fall back to copying only if required due to non-linear control.
- We have fully implemented the hybrid TRMc approach in the Koka compiler, and our benchmarks show that this approach can be very efficient. We measure various variants of modulo *cons* recursive functions and for linear control the TRMc transformed version is always faster than alternative approaches (Section 6).

2 AN OVERVIEW OF TAIL RECURSION MODULO CONS

As shown in the introduction, the prototypical example of a function that can be transformed by TRMc is the `map` function. One way to rewrite the `map` function manually to become tail-recursive is to use continuation passing style (CPS) where we add a continuation parameter k:[1]

```
fun mapk( xs : list<a>, f : a -> e b, k : list<b> -> e list<b> ) : e list<b>
  match xs
    Cons(x,xx) -> val y = f(x) in mapk(xx, f, compose(k, fn(ys) Cons(y, ys)) )
    Nil        -> apply(k,Nil)

fun map( xs : list<a>, f : a -> e b ) : e list<b>
  mapk(xs, f, id)
```

where `id` is the identity function, and `apply` and `compose` regular function application and composition:

[1] All our code examples use the Koka language (Leijen, 2021) since it has a full implementation of TRMc using the design in this paper including support for non-linear control (which cannot be handled by previous TRMc techniques). Every function arrow in Koka has three arguments where the type `a -> e b` denotes a function from type `a` to `b` with a potential (side) effect `e`. The type of `map` signifies that the polymorphic effect `e` of the `map` function itself is the same as the effect `e` of the passed in function `f`.

```
fun compose( f : b -> e c, g : a -> e b ) : (a -> e c) = fn(x) f(g(x))
fun apply( f : a -> e b, x : a ) : e b = f(x)
fun id( x : a ) : a = x
```

Note that we have to evaluate `f(x)` before allocating the closure `fn(ys) Cons(y,ys)` since `f` may have an observable (side) effect.

2.1 Continuation Style TRMc

Our new tail-recursive version of `map` may not consume any extra stack space, but it achieves this at the cost of allocating many intermediate closures in the heap, that each allocate a `Cons` node for the final result list. The TRMc translation is based on the insight that for many contexts around a tail-recursive call we can often use more efficient implementations than function composition.

In this paper, we are going to abstract over particular constructor contexts and instead represent use abstract program contexts as `ctx<a>` with three operations. First, the `ctx body` expression creates such contexts which can contain a single hole denoted as □; for example `ctx Cons(1,Cons(2,□)) : ctx<list<int>>`. We can see here that the context type `ctx<a>` is parameterized by the type of the hole `a`, which for our purposes must match the result type as well. Furthermore, we can compose and apply these abstract contexts as:

```
fun comp( k1 : ctx<a>, k2 : ctx<a> ) : ctx<a>
fun app( k : ctx<a>, x : a ) : a
```

Our general TRMC translation can convert a function like `map` automatically to a tail-recursive version by recognizing that each recursive invocation to `map` is under a constant constructor context (Section 5), leading to:

```
fun mapk( xs : list<a>, f : a -> e b, k : ctx<list<b>> ) : e list<b>
  match xs
    Cons(x,xx) -> val y = f(x) in mapk(xx, f, comp(k, ctx Cons(y,□)))
    Nil        -> app(k, Nil)

fun map( xs : list<a>, f : a -> e b ) : e list<b>
  mapk(xs, f, ctx □)
```

This is essentially equivalent to our manually translated CPS-style `map` function where we replaced function application and composition with context application and context composition, and the identity function with `ctx □`.

Thus, an obvious way to give semantics to our abstract contexts `ctx<a>` is to represent them as functions `a -> a`, where a context expression is interpreted as a function with a single parameter for the hole, e.g. `ctx Cons(1,Cons(2,□))` = `fn(x) Cons(1,Cons(2,x))` (and therefore `ctx □` = `fn(x) x` = `id`). Context application and composition then map directly onto function application and composition:

```
alias ctx<a> = a -> a
fun comp( k1 : ctx<a>, k2 : ctx<a> ) : ctx<a>  = compose(k1,k2)
fun app( k : ctx<a>, x : a ) : a               = apply(k,x)
```

Of course, using such semantics is equivalent to our original manual implementation and does not improve efficiency.

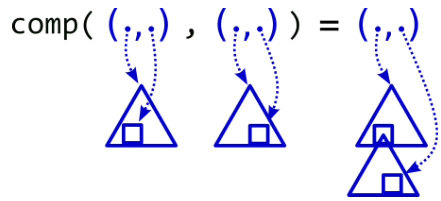
2.2 Linear Continuation Style

The insight of Risch (1973) and Friedman and Wise (1975) that leads to increased efficiency is to observe that the transformation always uses the abstract context k in a linear way, and we can implement the composition and application by updating the context holes *in-place*. Following the implementation strategy of Minamide (1998) for their hole-calculus, we can represent our abstract contexts as a *Minamide tuple* with a `res` field pointing to the final result object, and a `hole` field which points directly at the field containing the hole inside the result object. Assuming an assignment primitive `(:=)`, we can then implement composition and application efficiently as:

```
value type ctx<a>
  Id
  Ctx( res : a, hole : ptr<a> )

fun comp( k1 : ctx<a>, k2 : ctx<a> ) : ctx<a>
  Ctx( app(k1,k2.res), k2.hole)

fun app( k : ctx<a>, x : a ) : a
  match k
    Id -> x
    Ctx(res,hole) -> { hole := x; res }
```



where the empty `ctx` □ is represented as `Id` (since we do not yet have an address for the `Ctx.hole` field). If we inline these definitions in the `mapk` function, we can see we end up with an very efficient implementation where each new `Cons` cell is directly appended to the partially build final result list. In our actual implementation we optimize a bit more by defining the `ctx` type as a value type with only the `Ctx` constructor where we represent the `Id` case with a `hole` field containing a null pointer. Such tuple is passed at runtime in two registers and leads to efficient code where the `match` in the `app` function for example just zero-compares a register (see Appendix F). Section 6 shows detailed performance figures that show that the TRMc transformation always outperforms alternative implementations (for linear control flow).

In the following sections we formalize our calculus and calculate a general tail-recursion modulo *contexts* algorithm (Section 3) that we then instantiate to various use cases (Section 4), and in particular we study the efficient modulo *cons* instantiation (Section 5), and finally conclude with benchmarks (Section 6) and related work.

3 CALCULATING TAIL-RECURSION-MODULO-CONTEXT

In order to precisely reason about our transformation, we define a small calculus in Figure 1. The calculus is mostly standard with expressions e consisting of values v, application $e_1\ e_2$, let-bindings, and pattern matches. We assume well-typed programs that cannot go wrong, and where pattern matches are always complete and cannot get stuck. Since we reason in particular over recursive definitions, we add a special environment F of named recursive functions f. We could have encoded this using a fix combinator but using explicitly named definitions is more convenient for our purposes.

Following the approach of Wright and Felleisen (1994), we define applicative order evaluation contexts E. Generally, contexts are expressions with one subexpression denoted as a hole □. We write $\mathsf{E}[v]$ for the substitution $\mathsf{E}[\square := v]$. The definition of E ensures a single reduction order where we never evaluate under a lambda. The operational semantics can now be given using small step reduction rules of the form $e \longrightarrow e'$ together with the (*step*) rule to reduce in any evaluation context $\mathsf{E}[e] \longmapsto \mathsf{E}[e']$ (and in essence, an E context is an abstraction of the program stack and registers). We write $\longmapsto^*$ for the reflexive and transitive closure of the $\longmapsto$ reduction relation. The small step operational rules are standard, except for the (*fun*) rule that assumes a global F environment of recursive function definitions.

When $e \longmapsto^* v$, we call e *terminating* (also called *valuable* (Harper, 2012)). When an evaluation does not terminate, we write $e \Uparrow$. We write $e \cong e'$ if e and e' are extensionally equivalent: either $e \longmapsto^* v$ and $e' \longmapsto^* v$, or both $e \Uparrow$ and $e' \Uparrow$. During reasoning, we often use the rule that when e' is terminating, then $(\lambda x.\ e)\ e' \cong e[x{:=}e']$.

3.1 Abstract Contexts

Before we start calculating our general TRMC transformation, we first define *abstract contexts* as an abstract type *ctx* τ in our calculus. There are three context operations: creation (as ctx), application (as app), and composition (as (•)). These are not available to the user but instead are only generated as the target calculus of our TRMC translation. We extend the calculus as follows:

$v \ ::= \ \ldots \mid \mathsf{ctx}\ \mathsf{E} \mid \_ \bullet \_ \mid \mathsf{app}$

Expressions:

$$\begin{array}{llll|llll} e & ::= & v & \text{(value)} & v & ::= & x, y & \text{(variables)} \\ & \mid & e\ e & \text{(application)} & & \mid & f & \text{(recursive functions)} \\ & \mid & \mathsf{let}\ x\ =\ e\ \mathsf{in}\ e & \text{(let binding)} & & \mid & \lambda x.\ e & \text{(functions)} \\ & \mid & \mathsf{match}\ e\ \{\ \overline{p_i \mapsto e_i}\ \} & \text{(matching)} & & \mid & C^k & \text{(constructor of arity } k\ \geqslant\ 0) \\ \\ p & ::= & C^k\ x_1 \ldots x_k & \text{(pattern)} & F & ::= & \{\ \overline{f_i\ =\ \lambda x.\ e_i}\ \} & \text{(recursive definitions)} \end{array}$$

Syntax:

$$\begin{array}{lll} f\ x_1 \ldots x_n\ =\ e & \doteq & f\ =\ \lambda x_1 \ldots x_n.\ e \\ \lambda x_1 \ldots x_n.\ e & \doteq & \lambda x_1. \ldots \lambda x_n.\ e \end{array}$$

Evaluation Contexts:

$$\mathsf{E}\ ::=\ \square \mid \mathsf{E}\ e \mid v\ \mathsf{E} \mid \mathsf{let}\ x\ =\ \mathsf{E}\ \mathsf{in}\ e \mid \mathsf{match}\ \mathsf{E}\ \{\ \overline{p_i \mapsto e_i}\ \} \quad \text{(strict, left-to-right)}$$

Operational Semantics:

$$\begin{array}{lll} (\mathit{let}) & \mathsf{let}\ x\ =\ v\ \mathsf{in}\ e & \longrightarrow\ e[x{:=}v] \\ (\mathit{beta}) & (\lambda x.\ e)\ v & \longrightarrow\ e[x{:=}v] \\ (\mathit{fun}) & f\ v & \longrightarrow\ e[x{:=}v]\ \text{with}\ f = \lambda x.\ e\ \in F \\ (\mathit{match}) & \mathsf{match}\ (C^k\ v_1 \ldots v_k)\ \{\ \overline{p_i \mapsto e_i}\ \} & \longrightarrow\ e_i[x_1{:=}v_1, \ldots, x_k{:=}v_k]\ \text{with}\ p_i\ =\ C^k\ x_1 \ldots x_k \end{array}$$

$$\frac{e \longrightarrow e'}{\mathsf{E}[e] \longmapsto \mathsf{E}[e']}\ [\textsc{step}]$$

Fig. 1. Syntax and operational semantics.

where we assume that the abstract context operations are always terminating. In order to reason about contexts as an abstract type, we assume two context laws. The first one relates the application with the construction of a context:

$$(\mathit{appctx}) \quad \mathsf{app}\ (\mathsf{ctx}\ \mathsf{E})\ e \quad =\ \mathsf{E}[e]$$

The second law states that composition of contexts is equivalent to a composition of applications:

$$(\mathit{appcomp}) \quad \mathsf{app}\ (k_1 \bullet k_2)\ e \quad =\ \mathsf{app}\ k_1\ (\mathsf{app}\ k_2\ e)$$

When we instantiate to a particular implementation contexts, we need to show the context laws are satified. When we do this though we only need to show this for terminating expressions e, since if $e \Uparrow$, the laws hold by definition. In particular, for (appctx) it follows directly that $\mathsf{app}\ k\ e \Uparrow$ and $\mathsf{E}[e] \Uparrow$. Of particular note is that the latter only holds for E contexts and that is one reason why evaluation contexts are the *maximum* context possible for our TRMC translation. Similarly, for $(\mathit{appcomp})$ it follows directly that $(\mathit{app}\ (k_1 \bullet k_2)\ e) \Uparrow$ and $\mathsf{app}\ k_1\ (\mathsf{app}\ k_2\ e) \Uparrow$.

3.2 Calculating a General Tail-Recursion-Modulo-Contexts Algorithm

In this section we are going to calculate a general TRMC translation algorithm from its specification. For clarity we use single parameter functions for proofs and derivations (but of course the results extend straightforwardly to multiple parameter functions).

Consider a function $f\ x\ =\ e_f$ with its TRMC transformed version denoted as $\hat{f}$:

$$\hat{f}\ x\ k\ =\ [\![e_f]\!]_{f,k} \quad (k \notin \mathsf{fv}(e_f))$$

Our goal is to calculate the static TRMC transformation algorithm $[\![\_]\!]_{f,k}$ from its specification. The first question is then how we should even specify the intended behaviour of such function?

We can follow the standard approach for reasoning about continuation passing style (CPS) here. For example, Gibbons (2022) calculates the CPS version of the factorial function, called fact', from its specification as: $k\ (\mathit{fact}\ n) \ \cong\ \mathit{fact}'\ n\ k$, and similarly, Hutton (2021) calculates the CPS version of an evaluator from its specification as: exec $k\ (\mathit{eval}\ e) \ \cong\ \mathit{eval}'\ e\ k$. Following that approach, we use $\mathsf{app}\ k\ (f\ e) \ \cong\ \hat{f}\ e\ k$ **(a)** as our initial specification. This seems a good start since it implies:

$$
\begin{array}{lll}
 & f\ e & \\
= & \square[f\ e] & \{\ \mathit{context}\ \} \\
= & \mathsf{app}\ (\mathsf{ctx}\ \square)\ (f\ e) & \{\ (\mathit{appctx})\ \} \\
\cong & \hat{f}\ e\ (\mathsf{ctx}\ \square) & \{\ \mathit{specification}\ (a)\ \}
\end{array}
$$

and we can thus replace any applications of $f\ e$ in the program with applications to the TRMC translated $\hat{f}$ instead as $\hat{f}\ e\ (\mathsf{ctx}\ \square)$.

Unfortunately, the specification is not yet general enough as it does not include the translation function $[\![\_]\!]_{f,k}$ itself which limits what we can derive. Can we change this? Let's start by deriving how we can satisfy our initial specification (a):

$$
\begin{array}{lll}
 & \mathsf{app}\ k\ (f\ e) & \\
\cong & \mathsf{app}\ k\ e_f[x{:=}e] & \{\ (\mathit{fun}),\ e\ \mathit{is\ terminating}\ \} \\
= & (\mathsf{app}\ k\ e_f)[x{:=}e] & \{\ x \notin \mathsf{fv}(k)\ \} \\
= & [\![e_f]\!]_{f,k}[x{:=}e] & \{\ \mathit{calculate\ specification}\ (b)\ \} \\
\cong & \hat{f}\ v\ k & \{\ (\mathit{fun})\ \}
\end{array}
$$

(and if $e \Uparrow$, then $\mathsf{app}\ k\ (f\ e) \Uparrow$ and $\hat{f}\ e\ k \Uparrow$ follow directly).

This suggests a more general specification as $\mathsf{app}\ k\ e \ \cong\ [\![e]\!]_{f,k}$ **(b)** (for any e) which both implies our original specification, but also includes the translation function now. The new specification directly gives us a trivial solution for the translation as:

$$(\mathit{base}) \quad [\![e]\!]_{f,k} \ =\ \mathsf{app}\ k\ e$$

That is not quite what we need for general TRMC though since this does not translate any tail calls modulo a context. However, we can be more specific by matching on the shape of e. In particular, we can match on general tail-modulo-context calls as $e \ =\ \mathsf{E}[f\ e']$. We can then calculate:

$$
\begin{array}{lll}
 & \mathsf{app}\ k\ \mathsf{E}[f\ e'] & \\
= & \mathsf{app}\ k\ (\mathsf{app}\ (\mathsf{ctx}\ \mathsf{E})\ (f\ e')) & \{\ (\mathit{appctx})\ \} \\
= & \mathsf{app}\ (k \bullet \mathsf{ctx}\ \mathsf{E})\ (f\ e') & \{\ (\mathit{appcomp})\ \} \\
\cong & \hat{f}\ e'\ (k \bullet \mathsf{ctx}\ \mathsf{E}) & \{\ \mathit{specification}\ (a)\ \} \\
= & [\![\mathsf{E}[f\ e']]\!]_{f,k} & \{\ \mathit{calculate}\ \}
\end{array}
$$

which leads to the following set of equations:

$$
\begin{array}{llll}
(\mathit{tail}) & [\![\mathsf{E}[f\ e]]\!]_{f,k} & =\ \hat{f}\ e\ (k \bullet \mathsf{ctx}\ \mathsf{E}) & \text{iff } (\star) \\
(\mathit{base}) & [\![e]\!]_{f,k} & =\ \mathsf{app}\ k\ e & \text{otherwise}
\end{array}
$$

Note that the equations overlap – for a particular instance of the algorithm we generally constrain the (tail) rule to only apply for certain contexts E given by the condition $(\star)$ (for example, constructor contexts), falling back to (base) otherwise. Similarly, the (tail) case allows a choice in where to apply the tail call for expressions like $f\ (f\ e)$ for example and a particular instantiantion of $(\star)$ should disambiguate for an actual algorithm. By default, we assume that any instantiation matches on the innermost application of f (for reasons discussed in Section 4.1.1).

This is still a bit constrained, as these equations do not consider any evaluation contexts E where the recursive call is under a let or match expression. We can again match on these specific forms of e. For example $\mathsf{let}\ x \ =\ e'\ \mathsf{in}\ e$ where $e' \neq \mathsf{E}[f\ e'']$ (so it does not overlap with E contexts):

$$
\begin{array}{llcll}
(\mathit{tlet}) & [\![\mathsf{let}\ x\ =\ e'\ \mathsf{in}\ e]\!]_{f,k} & = & \mathsf{let}\ x\ =\ e'\ \mathsf{in}\ [\![e]\!]_{f,k} & \\
(\mathit{tmatch}) & [\![\mathsf{match}\ e'\ \{\ \overline{p_i \to e_i}\ \}]\!]_{f,k} & = & \mathsf{match}\ e'\ \{\ \overline{p_i \to [\![e_i]\!]_{f,k}}\ \} & \\
(\mathit{tail}) & [\![\mathsf{E}[f\ e_1\ \ldots\ e_n]]\!]_{f,k} & = & \hat{f}\ e_1\ \ldots\ e_n\ (k \bullet (\mathsf{ctx}\ \mathsf{E})) & \text{iff } (\star) \\
(\mathit{base}) & [\![e]\!]_{f,k} & = & \mathsf{app}\ k\ e & \text{otherwise}
\end{array}
$$

where $e' \neq \mathsf{E}[f\ e_1\ \ldots\ e_n]$.

Fig. 2. Calculated algorithm for general selective tail recursion modulo context transformation, parameterized by the $(\star)$ condition.

$$
\begin{array}{cll}
 & \mathsf{app}\ k\ (\mathsf{let}\ x\ =\ e'\ \mathsf{in}\ e) & \\
\cong & \mathsf{app}\ k\ e[x{:=}e'] & \{\ (\mathit{let}),\ e'\ \textit{is terminating}\ \} \\
= & (\mathsf{app}\ k\ e)[x{:=}e'] & \{\ x \notin \mathsf{fv}(k)\ \} \\
\cong & \mathsf{let}\ x\ =\ e'\ \mathsf{in}\ \mathsf{app}\ k\ e & \{\ e'\ \textit{is terminating}\ \} \\
\cong & \mathsf{let}\ x\ =\ e'\ \mathsf{in}\ [\![e]\!]_{f,k} & \{\ \textit{specification}\ \} \\
= & [\![\mathsf{let}\ x\ =\ e'\ \mathsf{in}\ e]\!]_{f,k} & \{\ \textit{calculate}\ \}
\end{array}
$$

(and if $e' \Uparrow$, then also $\mathsf{app}\ k\ (\mathsf{let}\ x\ =\ e'\ \mathsf{in}\ e) \Uparrow$ and $[\![\mathsf{let}\ x\ =\ e'\ \mathsf{in}\ e]\!]_{f,k} \Uparrow$). We can do the same for matches, where $e' \neq \mathsf{E}[f\ e'']$:

$$
\begin{array}{cll}
 & \mathsf{app}\ k\ (\mathsf{match}\ e'\ \{\ \overline{p_i \to e_i}\ \}) & \\
\cong & \mathsf{app}\ k\ e_i[x_1{:=}v_1, \ldots, x_n{:=}v_n] & \{\ p_i\ =\ C_i\ x_1\ \ldots\ x_n,\ e'\ \cong\ C_i\ v_1\ \ldots\ v_n,\ \mathbf{1}\ \} \\
= & (\mathsf{app}\ k\ e_i)[x_1{:=}v_1, \ldots, x_n{:=}v_n] & \{\ x_j \notin \mathsf{fv}(k)\ \} \\
\cong & [\![e_i]\!]_{f,k}[x_1{:=}v_1, \ldots, x_n{:=}v_n] & \{\ \textit{specification}\ \} \\
\cong & \mathsf{match}\ e'\ \{\ \overline{p_i \to [\![e_i]\!]_{f,k}}\ \} & \{\ (1)\ \} \\
= & [\![\mathsf{match}\ e'\ \{\ \overline{p_i \to e_i}\ \}]\!]_{f,k} & \{\ \textit{calculate}\ \}
\end{array}
$$

(and if $e' \Uparrow$, then also $\mathsf{app}\ k\ (\mathsf{match}\ e'\ \{\ \ldots\ \}) \Uparrow$ and $[\![\mathsf{match}\ e'\ \{\ \ldots\ \}]\!] \Uparrow$).

Figure 2 shows all four of the calculated equations for our generic tail recursion modulo contexts transformation (extended to multiple parameters). We can instantiate this algorithm by defining the context type $\mathit{ctx}\ \alpha$, the context construction (ctx), composition ($\bullet$), and application (app) operations, and finally the $(\star)$ condition constrains the allowed context E to fit the particular context type.

4 INSTANTIATIONS OF THE GENERAL TRMC TRANSFORMATION

With the general TRMC transformation in hand, we discuss various instantiations in this section. In the next section we look at the update-in-place modulo cons (TRMc) instantiation in detail.

4.1 Modulo Evaluation contexts

If we use *true* for the $(\star)$ condition, we can translate any recursive tail modulo evaluation context functions. Representing our abstract context directly as an E context is usually not possible though as E contexts generally contain *code*. The usual way to represent an arbitrary evaluation context E is simply as a (continuation) function $\lambda x.\ \mathsf{E}[x]$ with a context type $\mathit{ctx}\ \alpha\ =\ \alpha \to \alpha$:

$$
\begin{array}{llcll}
(\mathit{ectx}) & \mathsf{ctx}\ \mathsf{E} & = & \lambda x.\ \mathsf{E}[x] & (x \notin \mathsf{fv}(\mathsf{E})) \\
(\mathit{ecomp}) & k_1 \bullet k_2 & = & k_1 \circ k_1 & \\
(\mathit{eapp}) & \mathsf{app}\ k\ e & = & k\ e &
\end{array}
$$

This is an intuitive definition where $\mathsf{ctx}\ \square$ corresponds to the identity function and context composition to function composition. If we apply the TRMC translation we are essentially performing a selective CPS translation where the context E is represented as the continuation function. We can verify that the context laws hold for this instantiation (where we can assume e is terminating):

Composition:

$$
\begin{array}{lll}
& \mathsf{app}\ (k_1 \bullet k_2)\ e & \\
= & \mathsf{app}\ (k_1 \circ k_2)\ e & \{\ (ecomp)\ \} \\
= & \mathsf{app}\ (\lambda x.\ k_1\ (k_2\ x))\ e & \{\ def\ \circ\ \} \\
= & (\lambda x.\ k_1\ (k_2\ x))\ e & \{\ (eapp)\ \} \\
\cong & k_1\ (k_2\ e) & \{\ e\ term.,\ (beta)\ \} \\
= & k_1\ (\mathsf{app}\ k_2\ e) & \{\ (eapp)\ \} \\
= & \mathsf{app}\ k_1\ (\mathsf{app}\ k_2\ e) & \{\ (eapp)\ \}
\end{array}
$$

and application:

$$
\begin{array}{lll}
& \mathsf{app}\ (\mathsf{ctx}\ \mathsf{E})\ e & \\
= & \mathsf{app}\ (\lambda x.\ \mathsf{E}[x])\ e & \{\ (ecomp)\ \} \\
= & (\lambda x.\ \mathsf{E}[x])\ e & \{\ (eapp)\ \} \\
\cong & (\mathsf{E}[x])[x{:=}e] & \{\ e\ term.,\ (beta)\ \} \\
= & \mathsf{E}[e] & \{\ x \notin \mathsf{fv}(\mathsf{E})\ \}
\end{array}
$$

As a concrete example, let's apply the modulo evaluation context to the *map* function:

$$
\begin{array}{l}
map\ xs\ f\ =\ \mathsf{match}\ xs\ \{\ Nil \rightarrow Nil \\
\qquad\qquad\qquad\qquad\quad Cons\ x\ xx \rightarrow \mathsf{let}\ y\ =\ f\ x\ \mathsf{in}\ Cons\ y\ (map\ xx\ f)\ \}
\end{array}
$$

which translates to:

$$
\begin{array}{l}
map'\ xs\ f\ k\ =\ \mathsf{match}\ xs\ \{\ Nil \rightarrow \mathsf{app}\ k\ Nil \\
\qquad\qquad\qquad\qquad\quad Cons\ x\ xx \rightarrow \mathsf{let}\ y\ =\ f\ x\ \mathsf{in}\ map'\ xx\ f\ (k \bullet (\mathsf{ctx}\ (Cons\ y\ \square)))\ \}
\end{array}
$$

and which the compiler can further simplify into:

$$
\begin{array}{l}
map'\ xs\ f\ k\ =\ \mathsf{match}\ xs\ \{\ Nil \rightarrow k\ Nil \\
\qquad\qquad\qquad\qquad\quad Cons\ x\ xx \rightarrow \mathsf{let}\ y\ =\ f\ x\ \mathsf{in}\ map'\ xx\ f\ (\lambda x.\ k\ (Cons\ y\ x))\ \}
\end{array}
$$

where we derived exactly the standard CPS style version of `map` as shown in Section 2. A general evaluation context transformation creates more opportunities for tail-recursive calls, but this also happens at the cost of potentially heap allocating continuation closures. As such, it is not common for strict languages to use this instantiation. The exception would be languages like Scheme that always guarantee tail-calls but in that case the modulo evaluation contexts instantiation is already subsumed by general CPS conversion.

4.1.1 Nested Translation

The current instantiation is already very general as it applies to any E context but we can do a little better. While the innermost non-tail call $\mathsf{E}[f\ e]$ becomes $\hat{f}\ e\ (k \bullet \mathsf{ctx}\ \mathsf{E})$, the context E may contain itself further recursive calls to f. Since k is just a variable this allocates a closure for each composition ($\bullet$) and invokes every nested call $f\ e$ with an empty context as $\hat{f}\ e\ (\mathsf{ctx}\ \square)$ before composing with k. This is not ideal, and in the classic CPS translation this is avoided by passing k itself into the closure for $\mathsf{ctx}\ \mathsf{E}$ directly. Fortunately, we can achieve the same by *specialising* the compose function using the specification (b):

$$
\begin{array}{lll}
& k \bullet (ctx\ \mathsf{E}) & \\
= & \lambda x.\ k\ ((\mathsf{ctx}\ \mathsf{E})\ x) & \{\ (ecomp),\ (\bullet)\ \} \\
\cong & \lambda x.\ k\ \mathsf{E}[x] & \{\ (ectx),\ (beta)\ \} \\
= & \lambda x.\ \mathsf{app}\ k\ \mathsf{E}[x] & \{\ (eapp)\ \} \\
\cong & \lambda x.\ [\![\mathsf{E}[x]]\!]_{f,k} & \{\ specification\ (b)\ \}
\end{array}
$$

That is, in the compiler, instead of generating $k \bullet (\mathsf{ctx}\ \mathsf{E})$, we invoke the TRMC translation recursively in the $(tail)$ case and generate $\lambda x.\ [\![\mathsf{E}[x]]\!]_{f,k}$ instead. This avoids the allocation of function composition closures and directly passes the continuation k to any nested recursive calls.

4.2 Modulo Defunctionalized Evaluation Contexts

In order to better understand the shapes that evaluation contexts can take, we want to consider the *defunctionalization* (Reynolds, 1972; Danvy and Nielsen, 2001) of the general evaluation context transformation. It turns out that this yields an interesting context in its own right. First, we observe that in any recursive function the evaluation context can only take a finite number of shapes depending on the number of recursive calls. We write this as:

$\mathsf{E} ::= \square \mid \mathsf{E}_1 \mid \ldots \mid \mathsf{E}_n$

We define an *accumulator* datatype by creating a constructor H for the $\square$ context and for each E_i a constructor A_i that carries the free variables of E_i. The compiler then generates an app function where we interpret A_i by evaluating E_i with the stored free variables:

$$
\begin{array}{lllr}
(\mathit{dctx}) & \mathsf{ctx}\ \mathsf{E}_i & = \mathsf{A}_i\ x_1 \ldots x_m\ \mathsf{H} & \text{where } x_1, \ldots, x_m = \mathsf{fv}(\mathsf{E}_i) \\
(\mathit{dcomp}) & k_1 \bullet \mathsf{H} & = k_1 & \\
(\mathit{dcomp}) & k_1 \bullet (\mathsf{A}_i\ x_1 \ldots x_m\ k_2) & = \mathsf{A}_i\ x_1 \ldots x_m\ (k_1 \bullet k_2) & \\
(\mathit{dapp}) & \mathsf{app}\ \mathsf{H}\ e & = e & \\
(\mathit{dapp}) & \mathsf{app}\ (\mathsf{A}_i\ x_1 \ldots x_m\ k)\ e & = [\![\mathsf{E}_i[e\,,\ x_1 \ldots x_m]]\!]_{f,k} &
\end{array}
$$

Just as we saw in Section 4.1.1, we need to use the translated evaluation context in the definition of app translate nested calls. The context laws now follow by induction – see Appendix C.1 for the derivations. In Appendix D we also show that the translation remains typable in System F as this not generally the case for defunctionalized programs (Pottier and Gauthier, 2004). Applying this instantiation to the `map` function, we obtain:

$\mathsf{type}\ \mathit{ctx}\ \alpha\ =\ \mathsf{H} \mid \mathsf{A}_1\ \alpha\ (\mathit{ctx}\ \alpha)$
$\mathit{map}'\ \mathit{xs}\ f\ k\ =\ \mathsf{match}\ \mathit{xs}\ \{\ \mathit{Nil} \rightarrow \mathit{app}\ \mathsf{H}\ \mathit{Nil};\ \mathit{Cons}(x, \mathit{xx}) \rightarrow \mathsf{let}\ y\ =\ f\ x\ \mathsf{in}\ \mathit{map}'\ \mathit{xx}\ f\ (\mathsf{A}_1\ y\ k)\ \}$

In the *Cons* branch we have inlined $k \bullet (\mathsf{A}_1\ y\ \mathsf{H})$. The *app* function interprets A_1 by calling itself recursively on the stored evaluation context:

$\mathit{app}\ k\ \mathit{xs}\ =\ \mathsf{match}\ k\ \{\ \mathsf{H} \rightarrow \mathit{xs};\ \mathsf{A}_1(y,\ k') \rightarrow \mathit{app}\ k'\ (\mathit{Cons}\ y\ \mathit{xs})\ \}$

As we can see, using the modulo defunctionalized evalution context translation, we derived exactly the accumulator version of the `map` function that reverses the accumulated list in the end (where `app` is `reverse`)! In particular, for the special case where the all evaluation contexts are constructor contexts $C^m\ x_1 \ldots (f \ldots) \ldots x_m$ (as is the case for `map`), the accumulator datatype stores a path into the datastructure we are building and thus essentially becomes a zipper structure (Huet, 1997).

This defunctionalized approach might resemble general closure conversion at first [Appel:cont]: In both approaches, we store the free variables in a datatype. However, in closure conversion the datatype typically also contains a machine code pointer and one jumps to the code by calling this pointer, while in our case we match on the specialized constructors (similar to the approach of Tolmach and Oliva (1998)).

4.2.1 Reuse

As the defunctionalization makes the evaluation context explicit, we can optimize it further. As Sobel and Friedman (1998) note, the defunctionalized closure is only applied once and we can reuse its memory for other allocations. In Koka, this can happen automatically due to reuse analysis (Lorenzen and Leijen, 2022). In particular, in the `app` function, the match:

$\mathsf{A}_1\ y\ k' \rightarrow \mathit{app}\ k'\ \mathit{Cons}(y,\ \mathit{xs})$

can reuse the A_1 in-place to allocate the *Cons* node if the A_1 is unique at runtime. In our case, the context is actually always unique (we show this formally in Section 5.1), and the A_1 nodes are always reused! Even better, if the initial list is unique, we also reuse the initial *Cons* cell for the A_1 accumulator itself in map' and no allocation takes place at all – the program is functional but in-place (Reinking, Xie et al., 2021).

Generally, reuse does not apply if the size of the accumulator and that of the result do not match up. For our translation, we believe the sizes will often be equal though. For example, if the recursion is in a constructor context $C^m_i\ x_1 \ldots x_{i-1}\ (f \ldots)\ x_{i+1} \ldots x_m$, we generate a constructor $\mathsf{A}_i\ x_1 \ldots x_{i-1}\ x_{i+1} \ldots x_m\ k$ which also has m arguments, and these match up for reuse. More formally, McBride (2001) shows that the derivative of a regular type T creates m constructors on $m-1$

arguments for each C^m constructor of T, and then shows that the one-hole context is a list of this derivative. In our case we merge the list into the accumulator (as the last parameter k) and thus we have exactly m arguments for each contructor C^m that we recurse into.

4.3 Modulo Associative Operator Contexts

In the previous instantiations we considered general evalution contexts. However, we can often derive more efficient instantiations by considering more restricted contexts. A particularly nice example are monoidal contexts. For any monoid with an associative operator $\odot : \tau \to \tau \to \tau$ and a unit value $unit : \tau$, we can define a restricted operator context as:

$$\mathsf{A} ::= \square \mid v \odot \mathsf{A}$$

For a concrete example, consider the *length* function defined as:

$$length\ xs\ =\ \text{match}\ xs\ \{\ Cons\ x\ xx \to 1\ +\ length\ xx;\ \ Nil \to 0\ \}$$

which applies for integer addition ($\odot\ =\ +,\ unit\ =\ 0$). The idea is now to define a compile-time *fold* function $(\!|\_|\!)$ over a context A to always reduce the context to a single element of type τ:

$$\begin{aligned} (\!|\square|\!) &= unit \\ (\!|v \odot \mathsf{A}|\!) &= v \odot (\!|\mathsf{A}|\!) \end{aligned}$$

We can now instantiate the abstract contexts by defining the ($\star$) condition to constrain the E context to A, and the context type to $ctx\ \tau\ =\ \tau$, where we use the fold operation to represent contexts always as a single element of type τ:

$$\begin{array}{lrcl} (lctx) & \text{ctx}\ \mathsf{A} & = & (\!|\mathsf{A}|\!) \\ (lcomp) & k_1 \bullet k_2 & = & k_1 \odot k_2 \\ (lapp) & \text{app}\ k\ e & = & k \odot e \end{array}$$

The context laws hold for this definition. For composition we can derive:

$$\begin{array}{lll} & \text{app}\ (k_1 \bullet k_2)\ e & \\ = & \text{app}\ (k_1 \odot k_2)\ e & \{\ (lcomp)\ \} \\ = & (k_1 \odot k_2) \odot e & \{\ (lapp)\ \} \\ = & k_1 \odot (k_2 \odot e) & \{\ assoc.\ \} \\ = & \text{app}\ k_1\ (\text{app}\ k_2\ e) & \{\ (lapp)\ \} \end{array}$$

and for context application we have:

$$\begin{array}{lll} & \text{app}\ (\text{ctx}\ \mathsf{A})\ e & \\ = & \text{app}\ (\!|\mathsf{A}|\!)\ e & \{\ (lctx)\ \} \\ = & (\!|\mathsf{A}|\!) \odot e & \{\ (lapp)\ \} \end{array}$$

We proceed by induction over A.

Case $\mathsf{A}\ =\ \square$:

$$\begin{array}{lll} = & (\!|\square|\!) \odot e & \\ = & unit \odot e & \{\ fold\ \} \\ = & e & \{\ unit\ \} \\ = & \square[e] & \{\ \square\ \} \end{array}$$

and the case $\mathsf{A}\ =\ v \odot \mathsf{A}'$:

$$\begin{array}{lll} = & (\!|v \odot \mathsf{A}'|\!) \odot e & \\ = & (v \odot (\!|\mathsf{A}'|\!)) \odot e & \{\ fold\ \} \\ = & v \odot ((\!|\mathsf{A}'|\!) \odot e) & \{\ assoc.\ \} \\ = & v \odot \mathsf{A}'[e] & \{\ induction\ hyp.\ \} \\ = & \mathsf{A}[e] & \{\ \mathsf{A}\ context\ \} \end{array}$$

Common instantiations include integer addition ($\odot\ =\ +,\ unit\ =\ 0$) and integer multiplication ($\odot\ =\ \times,\ unit\ =\ 1$). The TRMC algorithm with A contexts instantiated with integer addition, translates the previous *length* function to the following tail-recursive version:

$$length'\ xs\ k\ =\ \text{match}\ xs\ \{\ Cons\ x\ xx \to length'\ xx\ (k \bullet (\text{ctx}\ (1\ +\ \square)));\ \ Nil \to \text{app}\ k\ 0\ \}$$

The intention is that the fold function is performed by the compiler, and the compiler can simplify this further as:

$k \bullet (\text{ctx}\ (1\ +\ \Box))\ =\ k\ +\ (\text{ctx}\ (1\ +\ \Box))\ =\ k\ +\ (\!|1+\Box|\!)\ =\ k\ +\ 1$

such that we end up with:

$\mathit{length}'\ \mathit{xs}\ k\ =\ \text{match}\ \mathit{xs}\ \{\ \mathit{Cons}\ x\ \mathit{xx} \rightarrow \mathit{length}'\ \mathit{xx}\ (k\ +\ 1);\ \mathit{Nil} \rightarrow k\ \}$

This time we derived exactly the text book accumulator version of *length*.

4.3.1 Using Right Biased Contexts

Our defined context only allows the recursive call on the left, but we can also define a right biased context:

$\mathsf{A}\ ::= \Box \mid \mathsf{A} \odot v$

with the fold defined as:

$$\begin{aligned} (\!|\Box|\!) &= \mathit{unit} \\ (\!|\mathsf{A} \odot v|\!) &= (\!|\mathsf{A}|\!) \odot v \end{aligned}$$

and where we now compose in opposite order:

$$\begin{array}{lll} (\mathit{rctx}) & \text{ctx}\ \mathsf{A} & = (\!|\mathsf{A}|\!) \\ (\mathit{rcomp}) & k_1 \bullet k_2 & = k_2 \odot k_1 \\ (\mathit{rapp}) & \text{app}\ k\ e & = e \odot k \end{array}$$

We can again show that the context laws hold for this definition (see Appendix C.2). As an example, we can instantiate $\odot$ as list append $+\!\!+$ with the empty list as the unit element to transform the *reverse* function:

$\mathit{reverse}\ \mathit{xs}\ =\ \text{match}\ \mathit{xs}\ \{\ \mathit{Cons}\ x\ \mathit{xx} \rightarrow \mathit{reverse}\ \mathit{xx} +\!\!+ [x];\ \mathit{Nil} \rightarrow [\,]\ \}$

First, our TRMC algorithm transforms it into:

$\mathit{reverse}'\ \mathit{xs}\ k =\ \text{match}\ \mathit{xs}\ \{\ \mathit{Cons}\ x\ \mathit{xx} \rightarrow \mathit{reverse}'\ \mathit{xx}\ (k \bullet (\text{ctx}\ (\Box +\!\!+ [x])));\ \mathit{Nil} \rightarrow \text{app}\ k\ [\,]\ \}$

and with our instantiated context, this simplifies to:

$\mathit{reverse}'\ \mathit{xs}\ k =\ \text{match}\ \mathit{xs}\ \{\ \mathit{Cons}\ x\ \mathit{xx} \rightarrow \mathit{reverse}'\ \mathit{xx}\ ([x] +\!\!+ k);\ \mathit{Nil} \rightarrow [\,] +\!\!+ k\ \}$

Using right-biased contexts, we derived the text book accumulator version of reverse.

4.4 Modulo Monoid Contexts

To handle general monoids, we need to consider recursive calls on both sides of the associative operation:

$\mathsf{A}\ ::= \Box \mid v \odot \mathsf{A} \mid \mathsf{A} \odot v$

This context A expresses arbitrarily nested applications of $\odot$. As monoid operations may not be commutative we cannot use a single element to represent the context. Instead we need to use a *product context* where we accumulate the left- and right context separately:

$$\begin{array}{lll} (\!|\Box|\!) & = (\mathit{unit},\ \mathit{unit}) & \\ (\!|v \odot \mathsf{A}|\!) & = (v \odot l,\ r) & \text{where}\ (l,\ r)\ =\ (\!|\mathsf{A}|\!) \\ (\!|\mathsf{A} \odot v|\!) & = (l,\ r \odot v) & \text{where}\ (l,\ r)\ =\ (\!|\mathsf{A}|\!) \end{array}$$

which we compose as:

$$\begin{array}{lll} (\mathit{actx}) & \text{ctx}\ \mathsf{A} & = (\!|\mathsf{A}|\!) \\ (\mathit{acomp}) & (l_1,\ r_1) \bullet (l_2,\ r_2) & = (l_1 \odot l_2,\ r_2 \odot r_1) \\ (\mathit{aapp}) & \text{app}\ (l, r)\ e & = l \odot e \odot r \end{array}$$

We can again show that the context laws hold for this definition (see Appendix C.3).

4.5 Modulo Semiring Contexts

We can also combine the associative operators of two monoids, as long as one distributes over the other. This is the case for semirings in particular (although we do not need commutativity of +). Semiring contexts are relatively common in practice. For example, consider the following hashing function for a list of integers as shown by Bloch (2008):

$$hash\ xs\ =\ \text{match}\ xs\ \{\ Cons\ x\ xx \rightarrow x\ +\ 31 * (hash\ xx)\ ;\ Nil \rightarrow 17\ \}$$

Implementing modulo semiring contexts in a compiler may be worthwhile as deriving a tail recursive version manually for such contexts is not always straightforward (and the interested reader may want to pause here and try to rewrite the *hash* function in a tail recursive way before reading on). We can define a general context for semirings as:

$$\mathsf{A} ::= \square \mid v\ +\ \mathsf{A} \mid v\ *\ \mathsf{A} \mid \mathsf{A}\ +\ v \mid \mathsf{A}\ *\ v$$

For simplicity, we assume we have a commutative semiring where both addition and multiplication commute. This allows us to use again a product representation at runtime where we accumulate the additions and multiplications seperately (and without commutativity we need a quadruple instead). In the definition of the fold we take into account that the multiplication distributes over the addition:

$$\begin{array}{lll}
(\!|\square|\!) & = (unit^{+},\ unit^{*}) & \\
(\!|v\ +\ \mathsf{A}|\!) & = (v\ +\ l,\ r) & \text{where } (l,\ r)\ =\ (\!|\mathsf{A}|\!) \\
(\!|v\ *\ \mathsf{A}|\!) & = (v\ *\ l,\ v\ *\ r) & \text{where } (l,\ r)\ =\ (\!|\mathsf{A}|\!) \\
(\!|\mathsf{A}\ +\ v|\!) & = (\!|v\ +\ \mathsf{A}|\!) & (+\ commutes) \\
(\!|\mathsf{A}\ *\ v|\!) & = (\!|v\ *\ \mathsf{A}|\!) & (*\ commutes)
\end{array}$$

Finally, to compose the contexts we need to use distributivity again. Note how the (*scomp*) rule mirrors the definition of $(\!|\mathsf{A}|\!)$ above:

$$\begin{array}{lll}
(sctx) & \text{ctx } \mathsf{A} & = (\!|\mathsf{A}|\!) \\
(scomp) & (l_1,\ r_1) \bullet (l_2,\ r_2) & = (l_1\ +\ (r_1 * l_2),\ r_1 * r_2) \\
(sapp) & \text{app } (l, r)\ e & = l\ +\ r * e
\end{array}$$

We can show the context laws hold for these definitions:

$$\begin{array}{lll}
 & \text{app } ((l_1,\ r_1) \bullet (l_2,\ r_2))\ e & \\
= & \text{app } (l_1\ +\ (r_1 * l_2),\ r_1 * r_2)\ e & \{\ (scomp)\ \} \\
= & (l_1\ +\ (r_1 * l_2))\ +\ (r_1 * r_2) * e & \{\ (sapp)\ \} \\
= & l_1\ +\ r_1 * (l_2\ +\ r_2 * e) & \{\ assoc\ and\ distr.\ \} \\
= & \text{app } (l_1,\ r_1)\ (\text{app } (l_2,\ r_2)\ e) & \{\ (sapp)\ \}
\end{array}$$

and

$$\begin{array}{lll}
 & \text{app } (\text{ctx } \mathsf{A})\ e & \\
= & \text{app } (\!|\mathsf{A}|\!)\ e & \{\ (sctx)\ \} \\
= & l\ +\ r * e & \{\ (sapp),\ for\ (l,\ r)\ =\ (\!|\mathsf{A}|\!)\ \}
\end{array}$$

We proceed by induction over A (where we compress some cases for brevity):

case $\mathsf{A}\ =\ \square$:

$$\begin{array}{lll}
= & l\ +\ r * e & \{\ (\!|\square|\!)\ =\ (l,\ r)\ \} \\
= & unit^{+}\ +\ unit^{*} * e & \{\ fold\ \} \\
= & e & \{\ unit\ \} \\
= & \square[e] & \{\ \square\ \}
\end{array}$$

and $\mathsf{A}\ =\ v_1\ +\ v_2\ *\ \mathsf{A}'$:

$$\begin{array}{lll}
= & l\ +\ r * e & \{\ (\!|v_1\ +\ v_2 * \mathsf{A}'|\!)\ =\ (l,\ r)\ \} \\
= & (v_1\ +\ v_2 * l')\ +\ (v_2 * r') * e & \{\ (\!|\mathsf{A}'|\!)\ =\ (l',\ r')\ \} \\
= & v_1\ +\ v_2 * (l'\ +\ r' * e) & \{\ assoc.\ and\ distr\ \} \\
= & v_1\ +\ v_2 * \mathsf{A}'[e] & \{\ induction\ hyp.\ \} \\
= & \mathsf{A}[e] & \{\ \mathsf{A}\ context\ \}
\end{array}$$

When we apply this to the `hash` function, we derive the tail recursive version as:

$$hash'\ xs\ k\ =\ \text{match}\ xs\ \{\ Cons\ x\ xx \rightarrow hash'\ xx\ (k \bullet (\text{ctx}\ (x\ +\ 31 * \square)));\ Nil \rightarrow \text{app}\ k\ 17\ \}$$

which further simplifies to:

$$hash'\ xs\ (l, r)\ =\ \text{match}\ xs\ \{\ Cons\ x\ xx \rightarrow hash'\ xx\ (l\ +\ r * x,\ r * 31);\ Nil \rightarrow l\ +\ r * 17\ \}$$

The final definition may not be quite so obvious and we argue that the modulo *semiring* instantiation may be a nice addition to any optimizing compiler.

4.6 Modulo Exponent Contexts

As a final example of an efficient representation of contexts we consider *exponent* contexts that consists a sequence of calls to a function g:

$$\mathsf{E} ::= \square \mid g\ \mathsf{E}$$

If we use a defunctionalized evaluation context from Section 4.2 we derive a datatype that is isomorphic to the peano-encoded natural numbers: the continuation *counts* how often we still have to apply g. As such, we can represent it more efficiently by an integer, where we fold an evaluation context into a count:

$$
\begin{aligned}
(\!| \square |\!) &= 0\\
(\!| g\ \mathsf{A} |\!) &= (\!| \mathsf{A} |\!) + 1
\end{aligned}
$$

We can define the primitive operations as:

$$
\begin{array}{lll}
(\textit{xctx}) & \mathsf{ctx}\ \mathsf{A} & = (\!| \mathsf{A} |\!)\\
(\textit{xcomp}) & k_1 \bullet k_2 & = k_1 + k_2\\
(\textit{xapp}) & \mathsf{app}\ 0\ e & = e\\
(\textit{xapp}) & \mathsf{app}\ (k+1)\ e & = \mathsf{app}\ k\ (g\ e)
\end{array}
$$

where app k e applies the function g to its argument k times. See Appendix C.4 for the derivations that show the context laws hold for this definition.

Note that if g is the enclosing function f, then the (*xapp*) specification is not tail-recursive. In that case, we can again use specification (b) to replace app k $(g\ e)$ by $[\![g\ e]\!]_{f,k}$ at compile time (as shown in Section 4.1.1). A nice example of such an exponent context is given by Wand (1980) who considers McCarthy's 91-function:

$$g\ x\ =\ \mathsf{if}\ x > 100\ \mathsf{then}\ x - 10\ \mathsf{else}\ g\ (g\ (x+11))$$

Using the exponent context with the recursive (*xapp*), we obtain a mutually tail-recursive version:

$$
\begin{aligned}
g'\ x\ k\ &=\ \mathsf{if}\ x > 100\ \mathsf{then}\ \textit{app}\ k\ (x - 10)\ \mathsf{else}\ g'\ (x+11)\ (k+1)\\
\textit{app}\ k\ e &=\ \mathsf{if}\ k = 0\ \mathsf{then}\ e\ \mathsf{else}\ g'\ x\ (k-1)
\end{aligned}
$$

5 MODULO CONSTRUCTOR CONTEXTS

As shown in the introduction, the most interesting instantiation is of course the modulo *cons* transformation on constructor contexts, since that particular case can be implemented using in-place updates which can usually not be replicated by the programmer. We can define a constant constructor context K as:

$$
\begin{array}{lcl}
\mathsf{K} & ::= & \square \mid C^k\ w_1 \ldots \mathsf{K} \ldots w_k\\
w & ::= & v \mid C^k\ w_1 \ldots w_k
\end{array}
$$

Note that this definition is a bit too restricted in practice where we would like to allow expressions e as constructor fields as well (as in `Cons(f(x),map(xx,f))` for example). We address this in Section 5.6, where we show how we can further generalize K contexts depending on the evalution order. For the formal development we use the more restricted K contexts for now.

We define the ($\star$) condition in the TRMC translation to restrict the context E to K contexts only. A possible way to define the contexts is to directly use K as a runtime context:

$$
\begin{array}{lll}
(\textit{kctx}) & \mathsf{ctx}\ \mathsf{K} & = \mathsf{K}\\
(\textit{kcomp}) & \mathsf{K}_1 \bullet \mathsf{K}_2 & = \mathsf{K}_1[\mathsf{K}_2]\\
(\textit{kapp}) & \mathsf{app}\ \mathsf{K}\ e & = \mathsf{K}[e]
\end{array}
$$

Similar to general evaluation contexts (Section 4.1), the context laws hold trivially for such definition (Appendix C.5) – and just as with general evaluation contexts, the *map* functions translates to:

$$\begin{array}{l} map'\ xs\ f\ k\ =\ \mathsf{match}\ xs\ \{ \\ \quad Nil \rightarrow \mathsf{app}\ k\ Nil \\ \quad Cons\ x\ xx \rightarrow \mathsf{let}\ y\ =\ f\ x\ \mathsf{in}\ map'\ xx\ f\ (k \bullet (\mathsf{ctx}\ (Cons\ y\ \square)))\ \} \end{array}$$

Even though this is a valid instantiation, it does not yet imply that this can be efficient. In particular, composition creates a fresh context every time as $\mathsf{K}_1[\mathsf{K}_2]$ and it may be difficult to implement such substitution efficiently at runtime as it needs to copy K_1 along the path to the hole. What we are looking for instead is an *in-place updating* instantiation that can compose in constant time.

5.1 Minamide

Minamide (1998) presents a "hole calculus" that can directly express our contexts in a functional way, but also allows an efficient in-place updating implementation. Using the hole calculus as our target calculus, we can instantiate the translation function using Minamide's system.

We define the context type as a "hole function" ($\hat{\lambda} x.\ e$), where $ctx\ \alpha \equiv\ hfun\ \alpha\ \alpha$. and instantiate the context operations to use the primitives as given by Minamide (1998):

$$\begin{array}{lrcl} (hctx) & \mathsf{ctx}\ \mathsf{K} & = & \hat{\lambda} x.\ \mathsf{K}[x] \\ (hcomp) & k_1 \bullet k_2 & = & hcomp\ k_1\ k_2 \\ (happ) & \mathsf{app}\ k\ e & = & happ\ k\ e \end{array}$$

Satisfyingly, our primitives turn out to map directly to the hole calculus primitives. The reduction rules for *happ* and *hcomp* specialized to our calculus are (Minamide, 1998, fig. 5):

$$\begin{array}{lll} (happly) & happ\ (\hat{\lambda} x.\ \mathsf{K})\ v & \longrightarrow \mathsf{K}[x{:=}v] \\ (hcompose) & hcomp\ (\hat{\lambda} x.\ \mathsf{K}_1)\ (\hat{\lambda} y.\ \mathsf{K}_2) & \longrightarrow \hat{\lambda} y.\ \mathsf{K}_1[x{:=}\mathsf{K}_2] \end{array}$$

This means that for any context k, we have $k\ \cong\ \hat{\lambda} x.\ \mathsf{K}[x]$ **(1)**. We can now show that our context laws are satisfied for this system:

Composition:

$$\begin{array}{lll} & \mathsf{app}\ (k_1 \bullet k_2)\ e & \\ = & \mathsf{app}\ (hcomp\ k_1\ k_2)\ e & \{\ (hcomp)\ \} \\ = & happ\ (hcomp\ k_1\ k_2)\ e & \{\ (happ)\ \} \\ \cong & happ\ (hcomp\ (\hat{\lambda} x.\ \mathsf{K}_1[x])\ (\hat{\lambda} y.\ \mathsf{K}_2[y]))\ e & \{\ (1),\ \mathbf{2}\ \} \\ \cong & happ\ (\hat{\lambda} y.\ \mathsf{K}_1[x][x{:=}\mathsf{K}_2[y]])\ e & \{\ (hcomp)\ \} \\ \cong & (\mathsf{K}_1[x][x{:=}\mathsf{K}_2[y]])[y{:=}e] & \{\ (happly)\ \} \\ = & \mathsf{K}_1[\mathsf{K}_2[e]] & \{\ contexts\ \} \\ \cong & \mathsf{K}_1[happ\ (\hat{\lambda} y.\ \mathsf{K}_2[y])\ e] & \{\ (happly)\ \} \\ \cong & happ\ (\hat{\lambda} x.\ \mathsf{K}_1[x])\ (happ\ (\hat{\lambda} y.\ \mathsf{K}_2[y])\ e) & \{\ (happly)\ \} \\ \cong & happ\ k_1\ (happ\ k_2\ e) & \{\ (1),\ (2)\ \} \\ = & \mathsf{app}\ k_1\ (\mathsf{app}\ k_2\ e) & \{\ (happ)\ \} \end{array}$$

and application:

$$\begin{array}{lll} & \mathsf{app}\ (\mathsf{ctx}\ \mathsf{K})\ e & \\ = & \mathsf{app}\ (\hat{\lambda} x.\ \mathsf{K}[x])\ e & \{\ (hctx)\ \} \\ = & happ\ (\hat{\lambda} x.\ \mathsf{K}[x])\ e & \{\ (happ)\ \} \\ \cong & \mathsf{K}[x][x{:=}e] & \{\ (happly)\ \} \\ = & \mathsf{K}[e] & \{\ contexts\ \} \end{array}$$

The hole calculus is restricted by a linear type discipline where the contexts $ctx\ \alpha \equiv\ hfun\ \alpha\ \alpha$ have a linear type. This is what enables an efficient in-place update implementation while still having a pure functional interface. For our needs, we need to check separately that the translation ensures that all uses of a context k are indeed linear. Type judgements in Minamide's system (Minamide, 1998, fig. 4) are denoted as $\Gamma\ ;\ H \vdash_{\mathsf{M}}\ e\ :\ \tau$ where Γ is the normal type environment, and H for linear bindings containing at most one linear value. The type environment Γ can itself contain linear values with a linear type (like *hfun*) but only pass those linearly to a single premise. The environment restricted to non-linear values is denoted as $\Gamma|_{\mathsf{N}}$. We can now show that our translation can indeed be typed under the linear type discipline:

Theorem 1. (*TRMC uses contexts linearly*)
If $\Gamma|_\mathsf{N}\ ;\ \varnothing \vdash_\mathsf{M}\ \mathsf{fun}\, f\ =\ \lambda x_1 \ldots x_n.\ e\ :\ \tau_1 \to \ldots \to \tau_n \to \tau$ and k fresh
then $\Gamma|_\mathsf{N},\ f\ ;\ \varnothing \vdash_\mathsf{M}\ \mathsf{fun}\, \hat{f}\ =\ \lambda x_1 \ldots x_n.\ \lambda k.\ [\![e]\!]_{f,k}\ :\ \tau_1 \to \ldots \to \tau_n \to ((\tau,\ \tau)\ \mathit{hfun}) \to \tau$.

To show this, we need a variant of the general replacement lemma (Wright and Felleisen, 1994, Lemma 4.2; Hindley and Seldin, 1986, Lemma 11.18) to reason about linear substitution in an evaluation context:

Lemma 1. (*Linear replacement*)
If $\Gamma|_\mathsf{N}\ ;\ \varnothing \vdash_\mathsf{M}\ \mathsf{K}[e]\ :\ \tau$ for a constructor context K then there is a sub-deduction $\Gamma|_\mathsf{N}\ ;\ \varnothing \vdash_\mathsf{M}\ e\ :\ \tau'$ at the hole and $\Gamma|_\mathsf{N}\ ;\ x\ :\ \tau' \vdash_\mathsf{M}\ \mathsf{K}[x]\ :\ \tau$.

Interestingly, this lemma requires constructor contexts and we would not be able to derive the Lemma for general contexts as the linear type environment is not propagated through applications. The proofs can be found in Appendix C.6, which also contains the full type rules adapted to our calculus.

5.2 In place update

The instantiation with Minamide's system is using fast in-place updates and proven sound, but it is still a bit unsatisfactory as *how* such in-place mutation is done (or why this is safe) is only described informally. In Minamide's system, a suggested implementation for a context is as a tuple $\langle \mathsf{K}, x@i \rangle$ where K is (a pointer to) a context and $x@i$ is the address of the hole as the i^{th} field of object x (in K). The empty tuple $\langle\rangle$ is used for an empty context ($\Box$). Composition and application directly update the hole pointed to by $x@i$ by overwriting the hole with the child context or value.

In contrast, Bour et al. (2021) show a TRMC translation for OCaml that uses *destination passing* style which makes it more explicit how the in-place update of the hole works. In particular, the general construct $x.i := v$ overwrites the i^{th} field of any object x with v. Like Minamide's work this is also described informally only.

To gain more insight of why in-place update is possible and correct, we are going to use the explicit heap semantics of Perceus (Reinking, Xie et al., 2021; Lorenzen and Leijen, 2022). In such semantics, the heap is explicit and all objects are explicitly reference counted. Using the Perceus derivation rules, we can soundly translate our current calculus to the Perceus target calculus where the reference counting instructions (dup and drop) are derived automatically by the derivation rules (Reinking, Xie et al., 2021, fig. 5). The Perceus heap semantics reduces the derived expressions using reduction steps of the form $H \mid e\ \longmapsto_r\ H' \mid e'$, which reduces a heap H and an expression e to a new heap H' and expression e' (Reinking, Xie et al., 2021, fig. 7). The heap H maps objects x with a reference count $n \geqslant 1$ to values, denoted as $x \mapsto^n v$. In this system, we can express in-place updates directly, and it turns out we can even *calculate* the in-place updating reduction rules for comp and app from the context laws. Before we do that though, we first need to establish some terminology and look carefully at what "in-place update" actually means.

5.2.1 The Essence of In-Place Update

Let's consider a generic copy function, ($x.i$ as y), that changes the i^{th} field of an object x to y, for any generic constructor C:

$x.i$ as $y\ =\ \mathsf{match}\ x\ \{\ C^k\ x_1 \ldots x_i \ldots x_k \to C^k\ x_1 \ldots y \ldots x_k\ \}$

When we apply the Perceus algorithm (Reinking, Xie et al., 2021) we need to insert a single drop:

$x.i$ as $y\ =\ \mathsf{match}\ x\ \{\ C^k\ x_1 \ldots x_i \ldots x_k \to \mathsf{drop}\ x_i;\ C^k\ x_1 \ldots y \ldots x_k\ \}$

In the special case that x is unique at runtime (i.e. the reference count of x is 1), we can now derive the following:

$$
\begin{array}{lll}
 & H,\ x \mapsto^1 C^k\ x_1 \ldots x_i \ldots x_k \mid x.i \ \mathsf{as}\ y & \{\ x \notin H,\ \mathbf{1}\ \} \\
= & H,\ x \mapsto^1 C^k\ x_1 \ldots x_i \ldots x_k \mid & \\
 & \mathsf{match}\ x\ \{\ C^k\ x_1 \ldots x_i \ldots x_k \to \mathsf{drop}\ x_i;\ C^k\ x_1 \ldots y \ldots x_k\ \} & \{\ \mathit{def.}\ \} \\
\longrightarrow_{\mathsf{r}} & H,\ x \mapsto^1 C\ \overline{x_j} \mid\ \mathsf{dup}(\overline{x_j});\ \mathsf{drop}(x);\ \mathsf{drop}(x_i);\ C^k\ x_1 \ldots y \ldots x_k & \{\ (\mathit{match}_r)\ \} \\
\longrightarrow_{\mathsf{r}}^{*} & H',\ x \mapsto^1 C\ \overline{x_j} \mid\ \mathsf{drop}(x);\ \mathsf{drop}(x_i);\ C^k\ x_1 \ldots y \ldots x_k & \{\ (\mathit{dup}_r),\ H'\ \mathit{has}\ \overline{x_j}\ \mathit{dupped},\ \mathbf{2}\ \} \\
\longrightarrow_{\mathsf{r}} & H' \mid \mathsf{drop}(\overline{x_j});\ \mathsf{drop}(x_i);\ C^k\ x_1 \ldots y \ldots x_k & \{\ (\mathit{drop}_r)\ \} \\
\longrightarrow_{\mathsf{r}} & H \mid \mathsf{drop}(x_i);\ C^k\ x_1 \ldots y \ldots x_k & \{\ \mathit{cancel}\ H'\ \mathit{dupped}\ \overline{x_j}\ (2)\ \} \\
\cong & H \mid \mathsf{let}\ z\ =\ C^k\ x_1 \ldots y \ldots x_k\ \mathsf{in}\ \mathsf{drop}(x_i);\ z & \{\ \mathit{drop\ commutes}\ \} \\
\longrightarrow_{\mathsf{r}} & H,\ z \mapsto^1 C\ x_1 \ldots y \ldots x_k \mid \mathsf{drop}(x_i);\ z & \{\ (\mathit{con}_r),\ \mathsf{fresh}\ z,\ \mathbf{3}\ \} \\
= & H,\ x \mapsto^1 C\ x_1 \ldots y \ldots x_k \mid \mathsf{drop}(x_i);\ x & \{\ \alpha\ \mathit{rename}\ (1), (3)\ \}
\end{array}
$$

And this is the essence of in-place mutation: when an object is unique, an in-place update corresponds to allocating a fresh copy, discarding the original (due to the uniqueness of x), and α-renaming to reuse the original "address".

We will write $(x.i := z)$ for $(x.i\ \mathsf{as}\ z)$ in the special case of updating a field in a unique constructor, where we can derive the following reduction rule:

$$(\mathit{assign}) \quad H,\ x \mapsto^1 C \ldots x_i \ldots \mid x.i := y \quad \longrightarrow_{\mathsf{r}}^{*} \quad H,\ x \mapsto^1 C \ldots y \ldots \mid \mathsf{drop}\ x_i;\ x$$

and in the case the field is a $\square$, we can further refine this to:

$$(\mathit{assignn}) \quad H,\ x \mapsto^1 C \ldots \square_i \ldots \mid x.i := y \quad \longrightarrow_{\mathsf{r}}^{*} \quad H,\ x \mapsto^1 C \ldots y \ldots \mid x$$

For convenience, we will from now on use the notation $C \ldots x_i \ldots$, and $C \ldots \square_i \ldots$ to denote the i^{th} field in a constructor if there is no ambiguity.

5.2.2 Linear Chains

We need a bit more generality to express hole updates in contexts. In particular, we will see that all objects along the path from the top of the context to the hole are unique by construction. We call such unique path a *linear chain*, denoted as $[H]^n_x$:

$$[H]^n_x \ =\ [x \mapsto^n v_0,\ x_1 \mapsto^1 v_1,\ \ldots,\ x_m \mapsto^1 v_m]^n_x \quad (m\ \geqslant\ 0)$$

where for all $x_i \in (\mathsf{dom}(H)\ -\ \{x\})$, we have $x_i \in \mathsf{fv}(v_{i-1})$ (and therefore for all $y \in \mathsf{dom}(H)$ we have $\mathsf{reachable}(H, x)$). Since the objects in H besides x are all unique and not reachable otherwise, we also say that x dominates H. When the dominator is also unique, we call it a *unique linear chain* (of the form $[H]^1_x$). We can define linear chains inductively as well since a single object always forms a linear chain:

$$(\mathit{linearone}) \quad x \mapsto^n v\ =\ [x \mapsto^n v]^n_x$$

and we can always extend with a unique linear chain:

$$(\mathit{linearcons}) \quad x \mapsto^n \ldots z \ldots,\ [H]^1_z\ =\ [x \mapsto^n \ldots z \ldots, H]^n_x$$

Using $(\mathit{linearcons})$ we can derive that we can append a unique linear chain as well:

$$(\mathit{linearapp}) \quad [H_1, y \mapsto^1 \ldots z \ldots]^n_x,\ [H_2]^1_z\ =\ [H_1, y \mapsto^1 \ldots z \ldots, H_2]^n_x$$

5.2.3 Contexts as a Linear Chain

To simplify the proofs, we assume in this sub section that all fields in K contexts are variables:

$$\mathsf{K}\ ::= \square \mid C\ x_1 \ldots \mathsf{K} \ldots x_n$$

since we can always arrange any K to have this form by let-binding the values w. It turns out that a constructor context then always evaluates to a unique linear chain:

Lemma 2. (*Contexts evaluate to unique linear chains*)
For any K, we have $H \mid \mathsf{K}[C \ldots \Box_i \ldots] \longrightarrow_{\mathsf{r}}^{*} H,\ [H', y \mapsto^{1} C \ldots \Box_i \ldots]^{1}_{x} \mid x$.

We can show this by induction on the shape of K (Appendix C.7).

5.2.4 Calculating the Fold

Following Minamide's approach, we are going to denote our contexts as a tuple $\langle x, y@i \rangle$ where x is (a pointer to) a constructor context and $y@i$ is the address of the hole as the i^{th} field of object y. We define ctx $\mathsf{K} = (\!|\mathsf{K}|\!)$. For an empty context we use an empty tuple ($(\!|\Box|\!) = \langle\rangle$), but otherwise we can specify the fold as:

$$(\textit{foldspec}) \quad H \mid (\!|\mathsf{K}[C \ldots \Box_i \ldots]|\!) \;\cong\; H \mid \mathsf{let}\ x = \mathsf{K}[C \ldots \Box_i \ldots]\ \mathsf{in}\ \langle x, [x]@i \rangle$$

where we use the notation $[x]$ do denote the last object of the linear chain formed by K (Lemma 2). We can now calculate the definition of $(\!|\_|\!)$ from its specification (see Appendix C.8), where we get following definition for $(\!|\_|\!)$:

$$\begin{array}{lll}
(\!|\Box|\!) & = & \langle\rangle \\
(\!|C \ldots \Box_i \ldots|\!) & = & \mathsf{let}\ x = C \ldots \Box_i \ldots\ \mathsf{in}\ \langle x,\ x@i \rangle \\
(\!|C \ldots \mathsf{K} \ldots|\!) & = & \mathsf{let}\ \langle z, x@i \rangle = (\!|\mathsf{K}|\!)\ \mathsf{in}\ \langle C \ldots z \ldots,\ x@i \rangle \quad (\mathsf{K} \neq \Box)
\end{array}$$

This builds up the context using let bindings, while propagating the address of the hole. As before, the intention is that the compiler expands the fold statically. For example, the *map* function translates to:

$$\begin{array}{l}
\textit{map}'\ \textit{xs}\ f\ k \;=\; \mathsf{match}\ \textit{xs}\ \{ \\
\quad \textit{Nil} \rightarrow \mathsf{app}\ k\ \textit{Nil} \\
\quad \textit{Cons}\ x\ \textit{xx} \rightarrow \mathsf{let}\ y = f\ x\ \mathsf{in}\ \textit{map}'\ \textit{xx}\ f\ (k \bullet (\mathsf{let}\ z = \textit{Cons}\ y\, \Box\ \mathsf{in}\ \langle z, z@2 \rangle))\ \}
\end{array}$$

where $z@2$ correctly denotes the address of the hole field in the context.

5.2.5 Updating a Context

Before we can define in-place application, we need an in-place substitution operation subst $\langle x, y@i \rangle\ z$ that substitutes z at the hole (at $y@i$) in the context x. Note that in our representation of a context as a tuple $\langle x, y@i \rangle$ we treat $y@i$ purely as an address and do not reference count y as such. The y part is a "weak" pointer and we cannot use it directly without also having an "real" reference. This means that if we want to define an in-place substitution we cannot define it directly as $y.i := z$ (since we have no real reference to y). Instead, we are going to calculate an in-place updating substitution from its specification:

$$(\textit{subspec}) \quad H, [H', y \mapsto^{1} C \ldots \Box_i \ldots]^{1}_{x} \mid \mathsf{subst}\ \langle x, y@i \rangle\ z \quad \cong \quad H, [H', y \mapsto^{1} C \ldots z \ldots]^{1}_{x} \mid x$$

We do this by induction of the shape of the linear chain. For the singleton case we have:

$$\begin{array}{lll}
 & H, [y \mapsto^{1} C \ldots \Box_i \ldots]^{1}_{y} \mid \mathsf{subst}\ \langle y, y@i \rangle\ z & \\
= & H, [y \mapsto^{1} C \ldots \Box_i \ldots]^{1}_{y} \mid y.i := z & \{\ \textit{calculate},\ (\textit{we have a y reference!})\ \} \\
\longrightarrow & H, [y \mapsto^{1} C \ldots z \ldots]^{1}_{y} \mid y & \{\ (\textit{assignn})\ \}
\end{array}$$

and for the extension we have:

$$\begin{array}{lll}
 & H, [x \mapsto^{1} C \ldots x'_j \ldots,\ [H', y \mapsto^{1} C \ldots \Box_i \ldots]^{1}_{x'}]^{1}_{x} \mid \mathsf{subst}\ \langle x, y@i \rangle\ z & \\
= & H, [x \mapsto^{1} C \ldots x'_j \ldots,\ [H', y \mapsto^{1} C \ldots \Box_i \ldots]^{1}_{x'}]^{1}_{x} & \\
 & \mid \mathsf{dup}\ x';\ x.j := \Box;\ x.j := \mathsf{subst}\ \langle x', y@i \rangle\ z & \{\ \textit{calculate}\ \} \\
\longrightarrow^{*} & H, [x \mapsto^{1} C \ldots \Box_j \ldots,\ [H', y \mapsto^{1} C \ldots \Box_i \ldots]^{1}_{x'}]^{1}_{x} \mid x.j := \mathsf{subst}\ \langle x', y \rangle\ z & \{\ (\textit{dup}_r),\ (\textit{assign})\ \} \\
\cong & H, [x \mapsto^{1} C \ldots \Box_j \ldots,\ [H', y \mapsto^{1} C \ldots z \ldots]^{1}_{x'}]^{1}_{x} \mid x.j := x' & \{\ \textit{induction hyp.}\ \} \\
\longrightarrow & H, [x \mapsto^{1} C \ldots x'_j \ldots,\ [H', y \mapsto^{1} C \ldots z \ldots]^{1}_{x'}]^{1}_{x} \mid x & \{\ (\textit{assignn})\ \}
\end{array}$$

This leads to the following inductive definition of subst:

$H \mid \mathsf{subst}\ \langle x, x@i\rangle\ z \ = \ H \mid x.i := z$
$H \mid \mathsf{subst}\ \langle x, y@i\rangle\ z \ = \ H \mid \mathsf{dup}\ x';\ x.j := \square;\ x.j := \mathsf{subst}\ \langle x', y@i\rangle\ z$
where $x \neq y\ \wedge [x \mapsto^1 C \ldots x'_j \ldots,\ [H']^1_{x'}]^1_x \in H$

That is, to update the last element of the chain in-place, we need traverse down while separating the links such that when we reach the final element it has a unique reference count and can be updated in-place. We then traverse back up fixing up all the links again. Of course, we would not actually use this implementation in practice – the derivation here just shows that the substitution specification is sound, and we can thus implement the (*subspec*) reduction by indeed using the tuple address $y@i$ directly to update the hole in-place. In essence, due to the uniquess of the of the elements in the chain, the y is uniquely reachable through x, and thus it is safe to use it directly in this case.

5.2.6 Calculating Application and Composition

With the specification for fold and in-place substitution, we can use the context laws to calculate the in-place updating version of application and composition. Starting with application, we can calculate (for $\mathsf{K} \neq \square$):

$$
\begin{array}{lll}
 & H \mid \mathsf{app}\ (\mathsf{ctx}\ \mathsf{K})\ e & \\
= & H \mid \mathsf{app}\ (\!|\mathsf{K}|\!)\ e & \{\ \mathit{def.}\ \} \\
\cong & H \mid \mathsf{app}\ (\mathsf{let}\ x\ =\ \mathsf{K}[\square]\ \mathsf{in}\ \langle x, [x]@i\rangle)\ e & \{\ \mathit{fold\ specification},\ \mathsf{K} \neq \square\ \} \\
\cong & H,\ [H', y \mapsto^1 C \ldots \square_i \ldots]^1_x \mid \mathsf{app}\ \langle x, [x]@i\rangle\ e & \{\ \mathit{lemma}\ 2,\ \mathbf{1}\ \} \\
= & H,\ [H', y \mapsto^1 C \ldots \square_i \ldots]^1_x \mid \mathsf{app}\ \langle x, y@i\rangle\ e & \{\ \mathit{def.}\ \} \\
\cong & H,\ z \mapsto^1 v,\ [H', y \mapsto^1 C \ldots \square_i \ldots]^1_x \mid \mathsf{app}\ \langle x, y@i\rangle\ z & \{\ \mathit{e\ is\ terminating}\ \mathbf{2}\ \} \\
= & H,\ z \mapsto^1 v,\ [H', y \mapsto^1 C \ldots \square_i \ldots]^1_x \mid \mathsf{subst}\ \langle x, y@i\rangle\ z & \{\ \mathit{calculate}\ \} \\
\cong & H,\ z \mapsto^1 v,\ [H', y \mapsto^1 C \ldots z \ldots]^1_x \mid x & \{\ (\mathit{subspec})\ \} \\
\cong & H,\ z \mapsto^1 v \mid \mathsf{K}[z] & \{\ \mathit{lemma}\ 2,\ (1)\ \} \\
\cong & H \mid \mathsf{K}[e] & \{\ (2)\ \}
\end{array}
$$

And thus we define application directly in terms of in-place substitution as:

$$(\mathit{uapp})\ H \mid \mathsf{app}\ \langle x, y@i\rangle\ z \quad \longrightarrow_{\mathsf{r}} \quad H \mid \mathsf{subst}\ \langle x, y@i\rangle\ z$$

We arrived exactly at the "obvious" implementation where the hole inside a unique context is updated in-place in constant time. This also corresponds to the informal implementation given in Section 2.2. For composition, it turns out we can define it in terms of applications:

$$(\mathit{ucomp})\ H \mid \langle x_1, y_1@i\rangle \bullet \langle x_2, y_2@j\rangle \quad \longrightarrow_{\mathsf{r}} \quad H \mid \langle \mathsf{app}\ \langle x_1, y_1@i\rangle\ x_2,\ y_2@j\rangle$$

where the derivation is in Appendix C.9. Again we arrived at the efficient translation where the hole in the first unique context is updated in-place (and in constant time) with a pointer to the second context. The full rules for application and composition are (with the derivations for the empty contexts in Appendix C.9):

$$
\begin{array}{llll}
(\mathit{uapph}) & H \mid \mathsf{app}\ \langle\rangle\ x & \longrightarrow_{\mathsf{r}} & H \mid x \\
(\mathit{uapp}) & H \mid \mathsf{app}\ \langle x, y@i\rangle\ z & \longrightarrow_{\mathsf{r}} & H \mid \mathsf{subst}\ \langle x, y@i\rangle\ z \\
(\mathit{ucomp}) & H \mid \langle x_1, y_1@i\rangle \bullet \langle x_2, y_2@j\rangle & \longrightarrow_{\mathsf{r}} & H \mid \langle \mathsf{app}\ \langle x_1, y_1@i\rangle\ x_2,\ y_2@j\rangle \\
(\mathit{ucompl}) & H \mid \langle\rangle \bullet \langle x_2, y_2@j\rangle & \longrightarrow_{\mathsf{r}} & H \mid \langle x_2, y_2@j\rangle \\
(\mathit{ucompr}) & H \mid \langle x_1, y_1@i\rangle \bullet \langle\rangle & \longrightarrow_{\mathsf{r}} & H \mid \langle x_1, y_1@i\rangle
\end{array}
$$

Note that (*ucompr*) is not really needed since by construction our translation never generates empty contexts for the second argument. The rules also correspond with the informal implementation given in Section 2.2 where `Id` was used to represent the empty tuple.

With these definitions, we still need to show that we can be *efficient* and that we never get *stuck*. For efficiency, we need to show that a context $\langle x, y@i\rangle$ is always a linear chain so we don't have to check that at runtime in (*subspec*). This follows by construction since any initial context ctx K is a linear chain (Lemma 2), and any composition as well (*ucomp*). Secondly, the reference count of the dominator should always be 1 or otherwise (*subspec*) may not apply – that is, contexts should be used linearly. This follows indirectly from Lemma 4 where we show that our translation adheres to Minamide's linear type discipline. A more direct approach would be to show that Perceus never derives a dup operation for a context k in our translation. However, we refrain from doing so here, as it turns out that with general algebraic effect handlers, the linearity of a context may no longer be guaranteed!

5.3 Non-Linear Control

A long standing issue in a TRMc transformation is that it is unsound in the presence of non-local control operations like *call/cc*, *shift/reset* (Shan, 2007), or in general with algebraic effect handlers (Plotkin and Power, 2003; Plotkin and Pretnar, 2009). Since the Koka language relies foundationally on effect handlers (Leijen, 2017, 2021; Xie and Leijen, 2021) we need to tackle this problem. Algebraic effect handlers extend the syntax with a handle expression, handle $h\ e$, and operations, op, that are handled by a handler h. There are two more reduction rules (Leijen, 2014):

$$\begin{array}{lll} (\mathit{return}) & \text{handle } h\ v & \longrightarrow\quad v \\ (\mathit{handle}) & \text{handle } h\ \mathsf{E}[\text{op } v] & \longrightarrow\quad e[x{:=}v,\ \mathit{resume}{:=}\ \lambda y.\ \text{handle } h\ \mathsf{E}[y]] \\ & & \text{where } (op\ \mapsto \lambda x.\ \lambda \mathit{resume}.\ e) \in h\ \wedge op \notin \mathsf{E} \end{array}$$

That is, when an operation is invoked it yields all the way up to the innermost handler for that operation and continues from there with the operation clause. Besides the operation argument, it also receives a resumption *resume* that allows the operation to return to the original call site with a result y. The culprit here is that the resumption captures the delimited evaluation context E in a lambda expression, and this can violate linearity assumptions. In particular, if we regard a TRMC context k as a linear value (as in Minamide), then such k may be in the context E of the (*handle*) rule and captured in a non-linear lambda. Whenever the operation clause calls the resumption more than once, any captured linear values may be used more than once!

A nice example in practice of this occurs in the well known Knapsack problem as described by Wu et al. (2014) where they use multiple resumptions to implement a non-determinism handler:

```
effect nondet
  ctl flip() : bool   // a control operation that may resume more than once
  ctl fail() : a      // or not at all

fun select( xs : list<a> ) : nondet a    // pick an element from a list
  match xs
    Nil        -> fail()
    Cons(x,xx) -> if flip() then x else select(xx)

fun knapsack(w : int, vs : list<int> ) : <nondet,div> list<int>
  if w < 0 then fail()
  elif w == 0 then []
  else val v = select(vs) in Cons(v, knapsack(w - v, vs))
```

The `knapsack` function picks items from a list of item weights `vs` that together do not exceed the capacity `w` (of the knapsack). It uses the `select` function that picks an element from a list using the `nondet` effect. We can now provide an effect handler that systematically explores all solutions using multiple resumptions:

```
val solutions = handler
  return(x)  [x]
  ctl fail() []
  ctl flip() resume(True) ++ resume(False)

fun test() : div list<list<int>>
  with solutions
  knapsack(3,[3,2,1])
```

That is, the `solutions` handler implements the `flip` function by resuming twice and appending the results. Even though `knapsack` returns a single solution as a list, the `test` function returns a list of all possible solution lists (as `[[3],[2,1],[1,2],[1,1,1]]`). The `knapsack` function is in the modulo *cons* fragment, and gets translated to a tail recursive version by our translation into something like:

```
fun knapsack'(w : int, vs : list<int>, k : ctx<list<int>> ) : <nondet,div> list<int>
  if w < 0 then app(k,fail())
  elif w == 0 then app(k,[])
  else val v = select(vs) in knapsack'(w - v, vs, val z = Cons(v,□) in comp(k,<z,z@2>))
```

Instead of having a runtime that captures evaluation contexts E directly, Koka usually uses an explicit monadic transformation to translate effectful computations into pure lambda calculus. The effect handling is then implemented explicitly using a generic multi-prompt control monad `eff` (Xie and Leijen, 2021, 2020). This transforms our `knapsack` function into something like:

```
fun knapsack'(w : int, vs : list<int>, k : ctx<list<int>> ) : eff<<nondet,div>,list<int>>
  if w < 0 then ...
  elif w == 0 then Pure( app(k,[]) )
  else match select(vs)
         Pure(v)    -> knapsack'(w - v, vs, val z = Cons(v,□) in comp(k,<z,z@2>))
         Yield(yld) -> Yield( yield-extend(yld,
                                fn(v) knapsack'(w - v, vs, val z = Cons(v,□) in comp(k,<z,z@2>) ))
```

Every computation in the effect monad either returns with a result (`Pure`) or is yielding up to a handler (`Yield`). Here we inlined the monadic bind operation where the result of `select(vs)` is explicitly matched. we see that in the `Yield` case, the continuation expression is now explicitly captured under a lambda expression – including the supposedly linear context `k`! This is how we can end up at runtime with a context that is shared (with a reference count > 1) and where the rule (*ucomp*) should not be applied.

5.4 A Hybrid Approach

Our context composition is defined in terms of context application, which in turn relies on on the in-place substitution (Section 5.2.5):

$$(subspec) \quad H, [H', y \mapsto^1 C \ldots \square_i \ldots]^1_x \mid \mathsf{subst}\ \langle x, y@i\rangle\ z \quad \cong \quad H, [H', y \mapsto^1 C \ldots z \ldots]^1_x \mid x$$

This is the operation that eventually fails if the runtime context x is not unique. In Section 5.2.5, the substitution operation was calculated to recursively visit the full linear chain of the context. This suggests a solution for any non-unique context: we can actually traverse the context at runtime and create a fresh copy instead.

It is not immediately clear though how to implement such operation at runtime: the linear chains up to now are just a proof technique and we cannot actually visit the elements of the chain at runtime as we do not know which field in a chain element points to the next element. What we need to do is to explicitly annotate each constructor C^k (of arity k) in a context also with an index i corresponding to the field that points to the next element, as C^k_i. It turns out, we can actually do this efficiently while constructing the context – and we can do it systematically just by modifying our fold function to keep track of this *context path* at construction:

$$\begin{aligned}
(\!|\square|\!) &= \langle\rangle \\
(\!|C \ldots \square_i \ldots|\!) &= \mathsf{let}\ x\ =\ C_i \ldots \square_i \ldots \mathsf{in}\ \langle x,\ x@i\rangle \\
(\!|C \ldots \mathsf{K}_i \ldots|\!) &= \mathsf{let}\ \langle z, x@j\rangle\ =\ (\!|\mathsf{K}|\!)\ \mathsf{in}\ \langle C_i \ldots z \ldots,\ x@j\rangle \quad (\mathsf{K} \neq \square)
\end{aligned}$$

With such indices present at runtime, we can define non-unique substitution as:

$$(subapp) \quad H, [H']^{n+1}_x \mid \mathsf{subst}\ \langle x, y@i\rangle\ z \quad \cong \quad H, [H']^{n+1}_x \mid \mathsf{append}\ x\ z$$

where append follows the context path at runtime copying each element as we go, and eventually appending z at the hole:

$$H, x \mapsto^n C_i \ldots \square_i \ldots \mid \mathsf{append}\ x\ z \quad \longrightarrow_r \quad H, x \mapsto^n C_i \ldots \square_i \ldots \mid x.i\ \mathsf{as}\ z$$
$$H, x \mapsto^n C_i \ldots y_i \ldots \mid \mathsf{append}\ x\ z \quad \longrightarrow_r \quad H, x \mapsto^n C_i \ldots y_i \ldots \mid \mathsf{dup}\ y_i;\ x.i\ \mathsf{as}\ (\mathsf{append}\ y_i\ z)$$

We can show the context laws still hold for these definitions (see Appendix C.10). The append operation in particular can be implemented efficiently at runtime using a fast loop that updates the previous element at each iteration (essentially using manual TRMC!). In the Koka runtime system, it happens to be the case that there is already an 8-bit field index in the header of each object which is used for stackless freeing. We can thus use that field for context paths since if a context is freed it is fine to discard the context path anyways. The runtime cost of the hybrid technique is mostly due to an extra uniqueness check needed when doing context composition to see if we can safely substitute in-place (see also Appendix F). As we see in the benchmark section, this turns out to be quite fast in practice. Moreover, the Koka compiler uses static type information when possible to avoid this check if a function is guaranteed to be used only with a linear effect type.

5.5 Another Approach: Fall Back to General Evaluation Contexts

The hybrid solution is elegant and duplicates no code, but it depends on both having precise reference counts, and having extra bits in a header to track the linear chain index in objects. This is not always available in general. However, the precise reference count is only needed for contexts, and one may add limited reference counting just for contexts during the TRMC translation which may work in otherwise garbage-collected systems.

If there are no bits for tracking linear chains at runtime, another solution may be to instantiate two versions when doing TRMC translation: one that uses fast in-place updating (Section 5.2), and one slower one that uses the general CPS-style translation (Section 4.1) (which *is* safe to use with non-linear control). We then generate a wrapper function that picks either one to use depending on whether non-linear control is possible or not. For example, if the language has *shift*/*reset*, the presence of a *reset* frame would signify this. Similarly, in Koka, each handler has a flag that signifies if it may resume more than once and we can check efficiently at runtime if any such handler is in the current evaluation context. The Koka compiler currently uses this technique when using the JavaScript backend.

5.6 Improving Constructor Contexts

As remarked before, our constructor contexts K are still too restricted in practice as they do not allow evaluation before the tail call f. For example $C\ e_1\ (f\ e_2)\ w$ is not a K context, and thus our current definition would not even apply to `Cons(f(x),map(xx,f))` as `f(x)` is an expression.

In practice though, we can let-bind expressions upfront such that TRMC applies, e.g. we rewrite $C\ e_1\ (f\ e_2)\ w$ into let $x\ =\ e_1$ in $C\ x\ (f\ e_2)\ w$, or `Cons(f(x),map(xx,f))` into `let y = f(x) in Cons(y ,map(xx,f))`. In particular, we can define a more liberal version of contexts K that allows expressions before the hole:

$$\mathbb{K} ::= \square \mid C^k\ e_1\ \ldots\ e_{i-1}\ \mathbb{K}\ w_{i+1}\ \ldots\ w_k$$

This corresponds to the semantics of Koka which has a strict left-to-right evalution order. Other choices are possible too, for example, OCaml does not specify the evalution order of the arguments and we could use a more liberal definition of $\mathbb{K}$ that allows all fields to be expressions. We now use the $\mathbb{K}$ definition to define a *lifting* step before the TRMC translation algorithm that makes the evalution order explicit and independent of the TRMC translation as such. The lifting function $[\![\_]\!]$ is defined on recursive functions as:

$$[\![f\ x_1 \ldots x_n\ =\ e]\!] \quad = \quad f\ x_1 \ldots x_n\ =\ [\![e]\!]$$

$$
\begin{array}{lll}
[\![\mathsf{let}\ x\ =\ e'\ \mathsf{in}\ e]\!] & = \mathsf{let}\ x\ =\ e'\ \mathsf{in}\ [\![e]\!] & \\
[\![\mathsf{match}\ e'\ \{\ \overline{p_i \to e_i}\ \}]\!] & = \mathsf{match}\ e'\ \{\ \overline{p_i \to [\![e_i]\!]}\ \} & \\
[\![\mathbb{K}[f\ e_1\ \ldots\ e_n]]\!] & = \mathsf{T}[\mathsf{K}[f\ e_1\ \ldots\ e_n]] & \text{where } (\mathsf{T},\mathsf{K})\ =\ \mathsf{extract}(\mathbb{K}) \\
[\![e]\!] & = \mathsf{app}\ k\ e & \text{otherwise}
\end{array}
$$

The lifting algorithm is defined analogous to TRMC translation and in practice we can do both at the same time since during TRMC we can match on $\mathbb{K}$ contexts (instead of K) and use extract to extract the required T and K contexts when needed. The $\mathsf{extract} : \mathbb{K} \to (\mathsf{T}, \mathsf{K})$ function extracts a tail context T to explicitly let-bind expression fields, and a proper constructor context K for which TRMC applies:

$$
\begin{array}{l}
\mathsf{extract}(\square)\ =\ (\square, \square) \\
\mathsf{extract}(C^k\ e_1\ \ldots\ e_{i-1}\ \mathbb{K}\ w_{i+1}\ \ldots\ w_k) \\
\ \ =\ (\mathsf{let}\ x_1\ =\ e_1\ \mathsf{in} \ldots \mathsf{let}\ x_{i-1}\ =\ e_{i-1}\ \mathsf{in}\ \mathsf{T},\ C^k\ x_1\ \ldots\ x_{i-1}\ \mathsf{K}\ w_{i+1}\ \ldots\ w_k) \\
\ \ \text{where } (\mathsf{T},\mathsf{K})\ =\ \mathsf{extract}(\mathbb{K})
\end{array}
$$

As said, one advantage of defining lifting separately is that it abstracts from a particular evaluation order which is language dependent. A second advantage is that it removes some of the syntactic nature of TRMC. For example, suppose we have an expression like $\mathsf{let}\ x\ =\ C\ 1\ (f\ e)\ \mathsf{in}\ C\ x\ \mathit{Nil}$. Here, modulo-cons does not apply as the recursive call to f is in the body of the let binding. This is why the Koka compiler performs TRMC after a local simplification phase where x is inlined to $C\ (C\ 1\ (f\ e))\ \mathit{Nil}$, which is now amenable to TRMC. However, without the lifting phase, doing simplification can also make things worse – consider for example an expression like $\mathsf{let}\ x\ =\ \mathit{id}\ 1\ \mathsf{in}\ C\ x\ (f\ e)$ which simplifies to $C\ (\mathit{id}\ 1)\ (f\ e)$ which is not a K context. With lifting we resolve this tension and we can inline freely to maximise the TRMC opportunities.

6 BENCHMARKS

The Koka compiler has a full implementation the TRMC algorithm as described in this paper for constructor contexts (since v2.0.3, Aug 2020). We measure the impact of TRMC relative to other variants on various tests: the standard map function over a list (*map*), mapping over a balanced binary tree (*tmap*), balanced insertion in a red-black tree (*rbtree*), and finally the *knapsack* problem as shown in Section 5.3. Each test program scales the repetitions to process the same number of total elements (100 000 000) for each test size.

The *map* test repeatedly maps the increment function over a shared list of numbers from 1 to N, and sums the result list. This means that the `map` function repeatedly copies the original list and Perceus cannot apply reuse here (Lorenzen and Leijen, 2022). For example, the test for the standard (and TRMC) `map` function in Koka is written as:

```
fun map-std( xs : list<a>, f : a -> e b ) : e list<b>
  match xs
    Cons(x,xx) -> Cons(f(x),xx.map-std(f))
    Nil        -> Nil

fun test(n : int)
  val xs = list(1,n)
  val x  = fold-int(0, 100_000_000/max(n,1), 0) fn(i,acc)
             acc + xs.map-std(fn(x) x + 1).sum
  println("total: " ++ x.show)
```

For each test, we measured five different variants:

- *trmc*: the TRMC version which is exactly like the standard (*std*) version.
- *std*: the standard non tail recursive version. This is the same source as the *trmc* version but compiled with the `-fno-trmc` flag.
- *acc*: this is the accumulator style definition where the accumulated result list- or tree-visitor is reversed in the end.

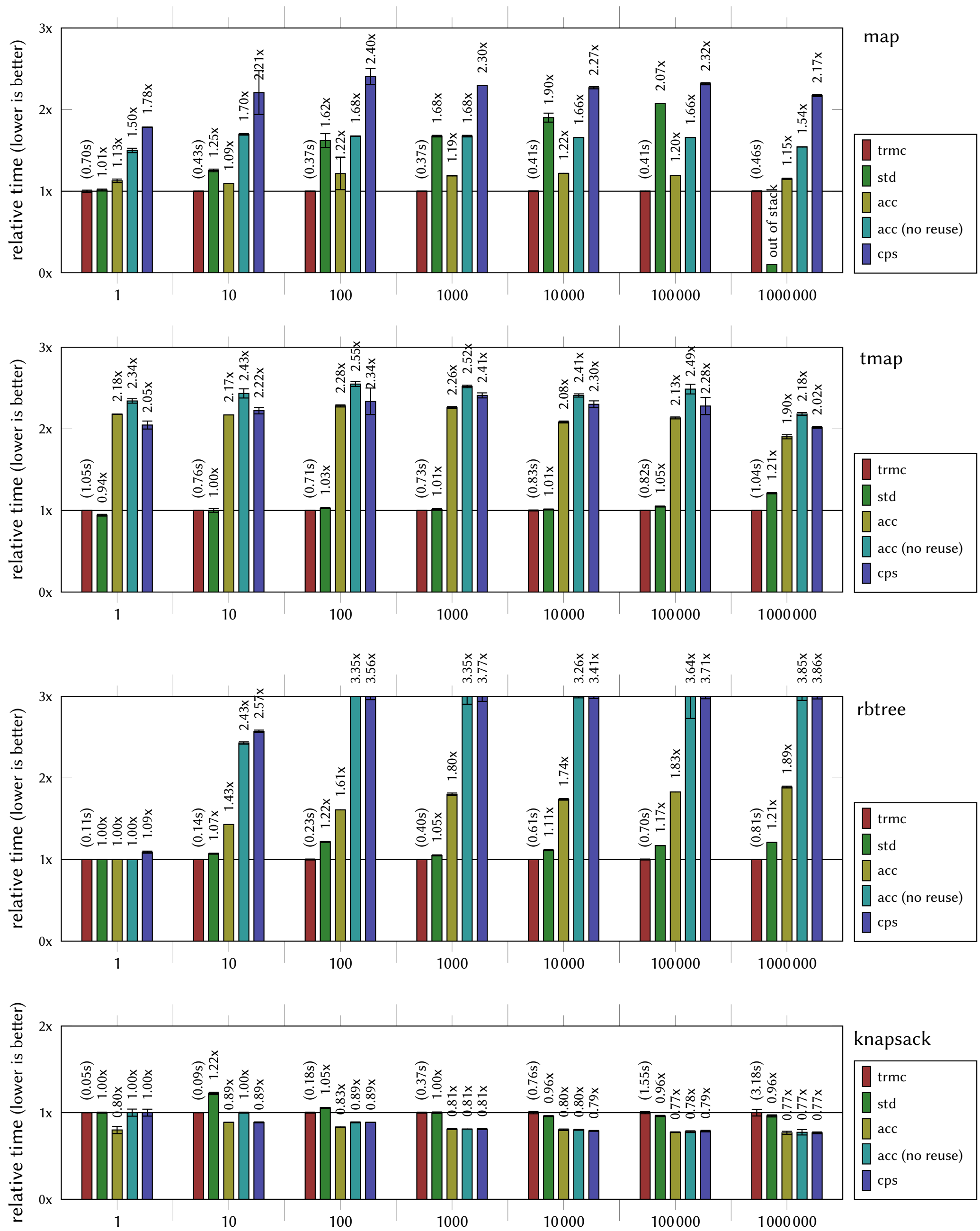


Fig. 3. Benchmarks on Ubuntu 20.04 (AMD 5950x), Koka v2.4.1-dev. The benchmarks are map over a list (*map*), map over a tree (*tmap*), balanced red-black tree insertion (*rbtree*), and the *knapsack* program that use non-linear control flow. Each workload is scaled to process the same number of total elements (usually 100 000 000). The tested variants are TRMC (*trmc*), the standard non tail recursive style (*std*), accumulator style (*acc*), accumulator style without Perceus reuse (*acc (no reuse)*), and finally CPS style (*cps*).

- *acc (no reuse)*: this is the accumulator style version but with Perceus reuse disabled for the accumulator. The performance of this variant may be more indicative for systems without reuse. Accumulator reuse is important as it allows the accumulated result to be reversed "in place".
- *cps*: the CPS style version with an explicit continuation function. This allocates a closure for every element that eventually allocates the result element for the final result.

The benchmark results are shown in Figure 3. For the *map* function we see that our TRMC translation is always faster than the alternatives for any size list. For a tree map (*tmap*) this is also the case, except for one-element trees where the standard *tmap* is slightly faster (6%). However, when we consider a slightly more realistic example of balanced insertion into a tree, TRMC is again as fast or faster in all cases. The *rbtree* benchmark is interesting as during traversal down to the insertion point, there a 2 recursive cases where TRMC applies, but also 2 recursive cases where TRMC does not apply. Here we see that it still helps to apply TRMC where possible as looping is apparently faster than a recursive call in this benchmark.

Finally, *knapsack* implements the example from 5.3 with a backtracking effect. Unfortunately, the TRMC variant, which uses the *hybrid* approach to copy the context on demand, is less fast than the alternatives. It is not *that* much slower though – about 25% at worst. The reason for this is that there is less sharing. For the accumulator version, at each choice point the current accumulated result is shared between each choice, building a tree of choices. At the end, many of these choices are just discarded (as the knapsack is too full), and only for valid solutions a result list is constructed (as a copy). However, for the *hybrid trmc* approach, we copy the context on demand *at each choice point*, and when we reach a point where the knapsack is too full the entire result is discarded, keeping only valid solutions. As such, the *trmc* variant copies more than the other approaches depending on how many of the generated solutions are eventually kept. Still, in Koka we prefer the hybrid approach to avoid code duplication.

7 RELATED WORK

Tail recursion modulo *cons* was a known technique in the LISP community as early as the 1970's. Risch (1973) describes the TRMc transformation in the context of REMREC system which also implemented the modulo associative operators instantiation described in Section 4.3. A more precise description of the TRMc transformation was given by Friedman and Wise (1975).

More recently, Bour et al. (2021) describle an implementation for OCaml which also explores various language design issues with TRMc. The implementation is based on *destination passing* style where the result is always directly written into the destination hole. This entails generating an initial unrolling of each function. For example, the `map` function is translated (in pseudo code) as:

```
fun map( xs, f )
  match xs
    Nil -> Nil
    Cons(x,xx) ->
      val y = f(x)
      val dst = Cons(y,□)
      map_dps( xx, f, dst@2 )
      dst
```

```
fun map_dps( xs, f, dst@i ) : ()
  match xs
    Nil -> dst.i := Nil
    Cons(x,xx) ->
      val y = f(x)
      val dst' = Cons(y,□)
      dst.i := dst'
      map_dps( xx, f, dst'@2 )
```

This can potentially be more efficient since there is only one extra argument for the destination address (instead of our representation as a Minamide tuple of the final result with the hole address) but it comes at the price of duplicating code. Note that the `map_dps` function returns just a unit value and is only called for its side effect. As such it seems quite different from our general TRMC based on context composition and application. However, the destination passing style may still be reconciled with our approach: with a Minamide tuple the first iteration always uses an "empty" tuple, while every subsequent iteration has a tuple with the fixed final result as its first element, where only the hole address (i.e. the destination) changes. Destination passing style uses this observation to specialize for each case, doing one unrolling for the first iteration (with the empty tuple), and then iterating with only the second hole address as the destination.

The algorithm rules by Bour et al. (2021) directly generate a destination passing style program. For example, the core translation rule for a constructor with a hole is:

$$\frac{n' = |I| + 1 \quad d'.n' \leftarrow \Box[U] \ \rightsquigarrow_{dps} \mathbb{T}[d_l.n_l \leftarrow \mathsf{K}_l]^l}{d.n \leftarrow \mathsf{K}[C((e_i)^{i \in I}, \Box, (e_j)^j)][U] \ \rightsquigarrow_{dps} \begin{array}{l} \mathsf{let}\ d' = C((e_i)^{i\in I}, \mathit{Hole}, (e_j)^j)\ \mathsf{in} \\ d.n \leftarrow \mathsf{K}[d']; \\ \mathbb{T}[d_l.n_l \leftarrow \mathsf{K}_l]^l \end{array}} \ [\textsc{dps-reify}]$$

Here a single rule does various tranformations that we treat as orthogonal, such as folding, extraction, instantiation of composition, and the actual TRMc transformation. Engels (2022) describes an implemention of TRMc for the Elm language while Lorenzen and Leijen (2022) describe how TRMc can be used to speed up balanced insertion into red-black trees. In some languages, other constructs take the place of TRMc. In logic languages, difference lists (Clark and Tärnlund, 1977) can be used for the same effect: they are usually presented as a pair (L, X) where X is a logic variable which is the last element of the list L. By setting X one can thus append to to L – quite similar to our constructor contexts. Pottier and Protzenko (2013) implement a type system inspired by separation logic, which allows the user to implement a safe version of in place updating TRMc through a mutable intermediate datatype. It is also well-known that lazy languages do not risk stack overflows when the recursion is guarded by a constructor allocation and thus such languages do not need TRMc to avoid stack overflow. Wadler (1984) demonstrates this principle for the listless machine.

Hughes (1986) considers the function `reverse` and shows how the fast version can be derived from the naive version by defining a new representation of lists. His function `rep` is equal to our ctx and his `abs` corresponds to app k [], and his correctness condition is exactly our (*appctx*) law. The idea of calculating programs from a specification has a long history and we refer the reader to early work by Bird (1984), Wand (1980), and Meertens (1986), and more recent work by Gibbons (2022) and Hutton (2021).

Defunctionalization has often been used to eliminate all higher-order calls and obtain a first-order version of a program. This technique was pioneered by Wand and Friedman (1978) who already describe a complete algorithm for LISP. Minamide et al. (1996) introduce special primitives `pack` and `open` (that correspond roughly to our *ctx* and app) and describe a type system for correct uses. Bell et al. (1997) and Tolmach and Oliva (1998) perform the conversion automatically at compile-time. Danvy and Nielsen (2001) propose to apply defunctionalization only to the closures of self-recursive calls, which should produce equal results as our approach in Section 4.2. However, they do not give an algorithm for this and the technique has so far mainly been used manually (Danvy and Goldberg, 2005; Gibbons, 2022). Sobel and Friedman (1998) propose to reuse the closures of a CPS transformed program for newly allocated constructors and show that this approach succeeds for all anamorphisms. However, by implementing reuse based on reference counts, we can reuse even original data for the accumulator and our approach is easily adopted to non-linear control.

8 CONCLUSION AND FUTURE WORK

In this paper we explored tail recursion modulo *context* and tried to bring the general principles out of the shadows of specific algorithms and into the light of equational reasoning. We have a full implementation of the modulo *cons* instantiation and look forward to explore future extensions to other instantiations as described in this paper.

REFERENCES

Jeffrey M. Bell, Françoise Bellegarde, and James Hook. "Type-Driven Defunctionalization," ICFP '97, . Association for Computing Machinery, New York, NY, USA, 25–37. 1997. doi:10.1145/258948.258953.

Richard S. Bird. "The Promotion and Accumulation Strategies in Transformational Programming." *ACM Transactions on Programming Languages and Systems* 6 (4): 487–504. Oct. 1984. doi:10.1145/1780.1781.

Joshua Bloch. *Effective Java*. Addison-Wesley Professional. 2008.

Frédéric Bour, Basile Clément, and Gabriel Scherer. "Tail Modulo Cons." *Journeées Francophones Des Langages Applicatifs (JFLA)*, April. Saint Médard d'Excideuil, France. Apr. 2021. `https://hal.inria.fr/hal-03146495/document`. hal-03146495.

Keith L Clark, and Sten-ke Tärnlund. "A First Order Theory of Data and Programs." In *IFIP Congress*, 939–944. 1977.

Olivier Danvy, and Mayer Goldberg. "There and Back Again." *Fundamenta Informaticae* 66 (4). IOS Press: 397–413. 2005.

Olivier Danvy, and Lasse R Nielsen. "Defunctionalization at Work." In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 162–174. 2001.

Damien Doligez, and Xavier Leroy. "A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML." In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages (POPL)*, 113–123. ACM press. Jan. 1993.

Jeroen Engels. "Tail Recursion, but modulo Cons." https://jfmengels.net/modulo-cons/. 2022. Accessed: 2022-06-06.

Daniel P. Friedman, and David S. Wise. *Unwinding Stylized Recursion into Iterations*. 19. Bloomingdale, Indiana. Dec. 1975. `https://legacy.cs.indiana.edu/ftp/techreports/TR19.pdf`.

Jeremy Gibbons. "Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity." *The Art, Science, and Engineering of Programming* 6 (November). Nov. 2022. Article 7.

Robert Harper. "15-150 Equational Reasoning Guide." 2012. `https://www.cs.cmu.edu/~15150/previous-semesters/2012-spring/resources/handouts/equational.pdf`. Notes for the 15-150 CMU Functional Programming course.

J. Hindley, and Jonathan Seldin. *Introduction to Combinators and Lambda-Calculus. Lambda-Calculus and Combinators, an Introduction*. Sep. 1986. doi:10.1017/CBO9780511809835.

Gérard P. Huet. "The Zipper." *Journal of Functional Programming* 7 (5): 549–554. 1997.

R John Muir Hughes. "A Novel Representation of Lists and Its Application to the Function "reverse."" *Information Processing Letters* 22 (3). Elsevier: 141–144. 1986.

Graham Hutton. "It's as Easy as 1, 2, 3." Jan. 2021. `https://www.cs.nott.ac.uk/~pszgmh/123.pdf`. Unpublished draft.

Daan Leijen. "Koka: Programming with Row Polymorphic Effect Types." In *MSFP'14, 5th Workshop on Mathematically Structured Functional Programming*. 2014. doi:10.4204/EPTCS.153.8.

Daan Leijen. "Type Directed Compilation of Row-Typed Algebraic Effects." In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 486–499. Paris, France. Jan. 2017. doi:10.1145/3009837.3009872.

Daan Leijen. "The Koka Language." 2021. `https://koka-lang.github.io`.

Anton Lorenzen, and Daan Leijen. "Reference Counting with Frame Limited Reuse." In *Proceedings of the 27th ACM SIGPLAN International Conference on Functional Programming (ICFP'2022)*. ICFP'22. Ljubljana, Slovenia. Sep. 2022.

Conor McBride. "The Derivative of a Regular Type Is Its Type of One-Hole Contexts." 2001. `http://strictlypositive.org/diff.pdf`. (Extended Abstract).

Lambert Meertens. "Algorithmics: Towards Programming as a Mathematical Activity." In *Mathematics and Computer Science*, 289–334. Jan. 1986.

Yasuhiko Minamide. "A Functional Representation of Data Structures with a Hole." In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 75–84. POPL '98. San Diego, California, USA. 1998. doi:10.1145/268946.268953.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. "Typed Closure Conversion." In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 271–283. 1996.

Chris Okasaki. *Purely Functional Data Structures*. Colombia University, New York. Jun. 1999.

Gordon D. Plotkin, and John Power. "Algebraic Operations and Generic Effects." *Applied Categorical Structures* 11 (1): 69–94. 2003. doi:10.1023/A:1023064908962.

Gordon D. Plotkin, and Matija Pretnar. "Handlers of Algebraic Effects." In *18th European Symposium on Programming Languages and Systems*, 80–94. ESOP'09. York, UK. Mar. 2009. doi:10.1007/978-3-642-00590-9_7.

François Pottier, and Nadji Gauthier. "Polymorphic Typed Defunctionalization." *ACM SIGPLAN Notices* 39 (1). ACM New York, NY, USA: 89–98. 2004.

François Pottier, and Jonathan Protzenko. "Programming with Permissions in Mezzo." In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, 173–184. ICFP '13. ACM, Boston, Massachusetts, USA. 2013. doi:10.1145/2500365.2500598.

Reinking, Xie, de Moura, and Leijen. "Perceus: Garbage Free Reference Counting with Reuse." In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 96–111. PLDI 2021. ACK, New York, NY, USA. 2021. doi:10.1145/3453483.3454032.

John C. Reynolds. “Definitional Interpreters for Higher-Order Programming Languages.” In *Proceedings of the ACM Annual Conference - Volume 2*, 717–740. ACM, Boston, Massachusetts, USA. 1972. doi:10.1145/800194.805852.

Tore Risch. “REMREC - A Program for Automatic Recursion Removal.” Inst. för Informationsbehandling, Uppsala Universitet. Nov. 1973. `https://user.it.uu.se/~torer/publ/remrec.pdf`.

Chung-chieh Shan. “A Static Simulation of Dynamic Delimited Control.” *Higher-Order and Symbolic Computation* 20 (4): 371–401. 2007. doi:10.1007/s10990-007-9010-4.

Jonathan Sobel, and Daniel P. Friedman. “Recycling Continuations.” In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, 251–260. ICFP ’98. Baltimore, Maryland, USA. 1998. doi:10.1145/289423.289452.

Andrew Tolmach, and Dino P Oliva. “From ML to Ada: Strongly-Typed Language Interoperability via Source Translation.” *Journal of Functional Programming* 8 (4). Cambridge University Press: 367–412. 1998.

Philip Wadler. “Listlessness Is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile-Time.” In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, 45–52. 1984.

Mitchell Wand. “Continuation-Based Program Transformation Strategies.” *Journal of the ACM* 27 (1): 164–180. Jan. 1980. doi:10.1145/322169.322183.

Mitchell Wand, and Daniel P Friedman. “Compiling Lambda-Expressions Using Continuations and Factorizations.” *Computer Languages* 3 (4). Pergamon: 241–263. 1978.

Andrew K. Wright, and Matthias Felleisen. “A Syntactic Approach to Type Soundness.” *Inf. Comput.* 115 (1): 38–94. Nov. 1994. doi:10.1006/inco.1994.1093.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. “Effect Handlers in Scope.” In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 1–12. Haskell ’14. Göthenburg, Sweden. 2014. doi:10.1145/2633357.2633358.

Ningning Xie, and Daan Leijen. “Effect Handlers in Haskell, Evidently.” In *Proceedings of the 2020 ACM SIGPLAN Symposium on Haskell*. Haskell’20. Jersey City, NJ. Aug. 2020.

Ningning Xie, and Daan Leijen. “Generized Evidence Passing for Effect Handlers – Efficient Compilation of Effect Handlers to C.” In *Proceedings of the 26th ACM SIGPLAN International Conference on Functional Programming (ICFP’2021)*. ICFP ’21. Virtual. Aug. 2021.

A FURTHER INSTANTIATIONS

A.1 Modulo Cons Products

Consider the partition function which partitions the elements of a list into two lists according to a given predicate function p:

$$\begin{array}{l} \mathit{partition}\ \mathit{xs}\ p\ =\ \mathsf{match}\ \mathit{xs}\ \{ \\ \quad \mathit{Nil} \to (\mathit{Nil}, \mathit{Nil}) \\ \quad \mathit{Cons}\ x\ \mathit{xx} \to \mathsf{let}\ \mathit{ok}\ =\ p\ x\ \mathsf{in}\ \mathsf{match}\ (\mathit{partition}\ \mathit{xx}\ p)\ \{ \\ \qquad (\mathit{ys}, \mathit{zs}) \to \mathsf{if}\ \mathit{ok}\ \mathsf{then}\ (\mathit{Cons}\ x\ \mathit{ys},\ \mathit{zs})\ \mathsf{else}\ (\mathit{ys},\ \mathit{Cons}\ x\ \mathit{zs}) \\ \}\ \} \end{array}$$

It turns out we accommodate such multiple result functions as well in our general TRMC framework. First we need to define a multi-hole context $\mathbb{T}$ that captures all tail positions:

$$\mathbb{T}\ ::=\ \square \mid \mathsf{let}\ x\ =\ e\ \mathsf{in}\ \mathbb{T} \mid \mathsf{match}\ e\ \{\ \overline{p_i\ \mapsto \mathbb{T}_i}\ \}$$

As noted by Bour et al. (2021), a tail context $\mathbb{T}$ is essentially the dual of an evaluation context. Using the multi-hole tail context, we can now define a (single hole) *cons product context* P as:

$$\mathsf{P}\ ::= \mathsf{match}\ \square\ \{\ (x, y) \to \mathbb{T}[\ \overline{(\mathsf{K}_1[x], \mathsf{K}_2[y])}\ )]\ \}$$

where $x, y \notin \mathsf{fv}(\mathbb{T})$ and $\mathbb{T}[\overline{v}]$ terminates **(a)**. This context captures any tail context where the product result is modified in a way that we can transform safely, i.e. where no side effects can happen. This allows us to represent a context directly as a product where we perform the matching upfront. We define our fold function as:

$$(\!|\mathsf{match}\ \square\ \{\ (x, y) \to \mathbb{T}[\overline{(\mathsf{K}_1[x], \mathsf{K}_2[y])}]\ \}|\!)\ =\ [\![\mathbb{T}[\overline{(\mathsf{K}_1, \mathsf{K}_2)}]]\!]$$

where

$$\begin{array}{lcl} [\![(\mathsf{K}_1, \mathsf{K}_2)]\!] & = & (\mathsf{ctx}\ \mathsf{K}_1,\ \mathsf{ctx}\ \mathsf{K}_2) \\ [\![\mathsf{match}\ e\ \{\ \overline{p_i \to e_i}\ \}]\!] & = & \mathsf{match}\ e\ \{\ \overline{p_i \to [\![e_i]\!]}\ \} \\ [\![\mathsf{let}\ x\ =\ e\ \mathsf{in}\ e']\!] & = & \mathsf{let}\ x\ =\ e\ \mathsf{in}\ [\![e']\!] \end{array}$$

Due to (a), we have $(\!|\mathsf{P}|\!)\ \cong\ (\mathsf{ctx}\ \mathsf{K}_1,\ \mathsf{ctx}\ \mathsf{K}_2)$ for some $\mathsf{K}_1, \mathsf{K}_2$ **(b)** and thus we can always represent a context as a tuple of constant constructor contexts ctx K at runtime. We can now define composition and application in terms of modulo *cons* composition and application (Section 5):

$$\begin{array}{llcl} (\mathit{qctx}) & \mathsf{ctx}\ \mathsf{P} & = & (\!|\mathsf{P}|\!) \\ (\mathit{qcomp}) & (k_1, k_2) \bullet (k_1', k_2') & = & (k_1 \bullet k_1', k_2 \bullet k_2') \\ (\mathit{qapp}) & \mathsf{app}\ (\mathsf{K}_1, \mathsf{K}_2)\ (e_1, e_2) & = & (\mathsf{app}\ k_1\ e_1, \mathsf{app}\ k_2\ e_2) \end{array}$$

Building on the laws for constructor contexts, we can show the context laws for product contexts are satisfied for this definition:

$$\begin{array}{cll} & \mathsf{app}\ ((k_1, k_2) \bullet (k_1', k_2'))\ (e_1, e_2) & \\ = & \mathsf{app}\ (k_1 \bullet k_1',\ k_2 \bullet k_2')\ (e_1, e_2) & \{\ (\mathit{qcomp})\ \} \\ = & (\mathsf{app}\ (k_1 \bullet k_1')\ e_1,\ \mathsf{app}\ (k_2 \bullet k_2')\ e_2) & \{\ (\mathit{qapp})\ \} \\ = & (\mathsf{app}\ k_1\ (\mathsf{app}\ k_1'\ e_1),\ \mathsf{app}\ k_2\ (\mathsf{app}\ k_2'\ e_2)) & \{\ (\mathit{kapp})\ \} \\ = & \mathsf{app}\ (k_1, k_2)\ (\mathsf{app}\ k_1'\ e_1,\ \mathsf{app}\ k_2'\ e_2) & \{\ (\mathit{qapp})\ \} \\ = & \mathsf{app}\ (k_1, k_2)\ (\mathsf{app}\ (k_1', k_2')\ (e_1, e_2)) & \{\ (\mathit{qapp})\ \} \end{array}$$

and

$$\begin{array}{cll} & \mathsf{app}\ (\mathsf{ctx}\ \mathsf{P})\ (e_1, e_2) & \\ = & \mathsf{app}\ ((\!|\mathsf{P}|\!))\ (e_1, e_2) & \{\ (\mathit{qctx})\ \} \\ \cong & \mathsf{app}\ (\mathsf{ctx}\ \mathsf{K}_1,\ \mathsf{ctx}\ \mathsf{K}_2)\ (e_1, e_2) & \{\ (b)\ \mathbf{1}\ \} \\ = & (\mathsf{app}\ (\mathsf{ctx}\ \mathsf{K}_1)\ e_1,\ \mathsf{app}\ (\mathsf{ctx}\ \mathsf{K}_2)\ e_2) & \{\ (\mathit{qapp})\ \} \\ = & (\mathsf{K}_1[e_1], \mathsf{K}_2[e_2]) & \{\ (\mathit{kapp})\ \} \\ \cong & \mathsf{P}[(e_1, e_2)] & \{\ (1)\ \mathit{and\ lemma}\ 3\ \} \end{array}$$

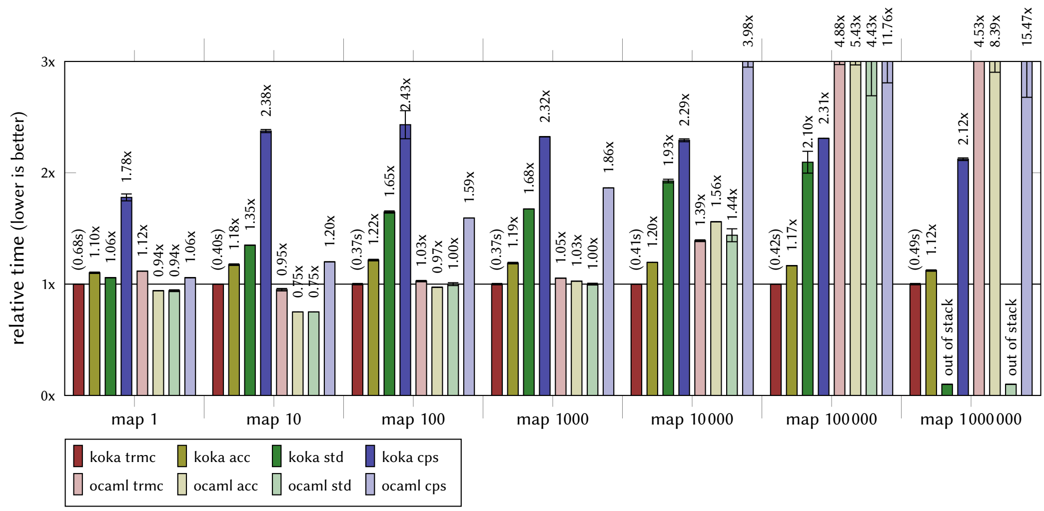


Fig. 4. Benchmarks on Ubuntu 20.04 (AMD 5950x), Koka v2.4.1-dev, OCaml 4.14.0. The benchmark repeatedly maps the increment function over a list of a given size and sums the result list. Each workload is scaled to process the same number of total elements (100 000 000). The tested variants of `map` are TRMC (trmc), accumulator style (acc), the standard non tail recursive style (std), and finally CPS style (cps).

□

where we assume the following lemma:

Lemma 3.
If $(\!|\mathsf{P}|\!) \;\cong\; (\mathsf{ctx}\ \mathsf{K}_1, \mathsf{ctx}\ \mathsf{K}_2)$ then $\mathsf{P}[(e_1, e_2)] \;\cong\; (\mathsf{K}_1[e_1], \mathsf{K}_2[e_2])$.

Using these definitions, our partition function satifies the P context and translates to:

$$
\begin{array}{l}
\mathit{partition'}\ \mathit{xs}\ p\ k \;=\; \mathsf{match}\ \mathit{xs}\ \{\\
\quad \mathit{Nil} \longrightarrow \mathsf{app}\ k\ (\mathit{Nil}, \mathit{Nil})\\
\quad \mathit{Cons}\ x\ \mathit{xx} \longrightarrow \mathsf{let}\ \mathit{ok} \;=\; p\ x\ \mathsf{in}\\
\qquad\qquad\qquad\quad \mathsf{let}\ k' \;=\; \mathsf{if}\ \mathit{ok}\ \mathsf{then}\ (\mathsf{ctx}\ \mathit{Cons}\ x\ \square,\ \mathsf{ctx}\ \square)\ \mathsf{else}\ (\mathsf{ctx}\ \square,\ \mathsf{ctx}\ \mathit{Cons}\ x\ \square)\ \mathsf{in}\\
\qquad\qquad\qquad\quad \mathit{partition'}\ \mathit{xx}\ p\ (k \bullet k')\\
\}
\end{array}
$$

After inlining the definitions and simplification we end up with:

$$
\begin{array}{l}
\mathit{partition'}\ \mathit{xs}\ p\ (k_1, k_2) \;=\; \mathsf{match}\ \mathit{xs}\ \{\\
\quad \mathit{Nil} \longrightarrow (\mathsf{app}\ k_1\ \mathit{Nil},\ \mathsf{app}\ k_2\ \mathit{Nil})\\
\quad \mathit{Cons}\ x\ \mathit{xx} \longrightarrow \mathsf{let}\ \mathit{ok} \;=\; p\ x\ \mathsf{in}\\
\qquad\qquad\qquad\quad \mathsf{let}\ (k_1', k_2') \;=\; \mathsf{if}\ \mathit{ok}\ \mathsf{then}\ (k_1 \bullet \mathsf{ctx}\ \mathit{Cons}\ x\ \square,\ k_2)\ \mathsf{else}\ (k_1,\ k_2 \bullet \mathsf{ctx}\ \mathit{Cons}\ x\ \square)\ \mathsf{in}\\
\qquad\qquad\qquad\quad \mathit{partition'}\ \mathit{xx}\ p\ (k_1', k_2')\\
\}
\end{array}
$$

where each result list is updated in-place using the modulo *cons* composition and application.

B FURTHER BENCHMARKS

Figure 4 shows benchmark results of the `map` benchmark. This time we included the results for OCaml 4.14.0 which has support for TRMc (Bour et al., 2021) using the `[@tail_mod_cons]` attribute.

For example, the TRMc `map` function is expressed as:

```
let[@tail_mod_cons] rec map_trmc xs f =
  match xs with
  | [] -> []
  | x :: xx -> let y = f x in y :: map_trmc xx f
```

Comparing across systems is always difficult since there are many different aspects, in particular the different memory management of both systems where Koka uses Perceus style reference counting (Reinking, Xie et al., 2021) and OCaml uses generational garbage collection, with a copying collector for the minor generation, and a mark-sweep collector for the major heap (Doligez and Leroy, 1993).

The results at least indicate that our approach, using Minamide style tuples of the final result object and a hole address, is competitive with the OCaml approach based on direct destination passing style. For our translation, the *trmc* translation is always as fast or faster as the alternatives, but unfortunately this is not the case in OCaml (yet) where it requires larger lists to become faster then the standard recursion.

OCaml is also faster for lists of size 10 where *std* is about 25% faster than Koka's *trmc*. We believe this is in particular due to memory management. For the micro benchmark, such small lists always fit in the minor heap with very fast bump allocation. Since in the benchmark the result is always immediately discarded no live data needs to be traced in the minor heap for GC – perfect! In contrast, Koka uses regular malloc/free with reference counting with the associated overheads. However, once the workload increases with larger lists, the overhead of garbage collection and copying to the major heap becomes larger, and in such situation Koka becomes (significantly) faster. Also, the time to process the 100M elements stays relatively stable for Koka (around 0.45s) no matter the sizes of the lists, while with GC we see that processing on larger lists takes much longer.

C PROOFS

C.1 Context Laws for Defunctionalized Contexts

$$\begin{array}{lll}
 & \text{app}\ (k_1 \bullet k_2)\ e & \\
= & \text{app}\ (k_1 \bullet \mathit{Hole})\ e & \{\ \mathit{assumption}\ \} \\
= & \text{app}\ k_1\ e & \{\ \mathit{def} \bullet\ \} \\
= & \text{app}\ k_1\ (\text{app}\ \mathit{Hole}\ e) & \{\ \mathit{def\ app}\ \} \\
= & \text{app}\ k_1\ (\text{app}\ k_2\ e) & \{\ \mathit{def}\ k_2\ \}
\end{array}$$

and case $k_2\ =\ \mathsf{A}_i\ x_1\ \ldots\ x_m\ k_3$

$$\begin{array}{lll}
 & \text{app}\ (k_1 \bullet k_2)\ e & \\
= & \text{app}\ (k_1 \bullet \mathsf{A}_i\ x_1\ \ldots\ x_m\ k_3)\ e & \{\ \mathit{assumption}\ \} \\
= & \text{app}\ (\mathsf{A}_i\ x_1\ \ldots\ x_m\ (k_1 \bullet k_3))\ e & \{\ \mathit{def} \circ\ \} \\
= & [\![\ \mathsf{E}_i[e \mid x_1,\ \ldots,\ x_m]\ ]\!]_{f,k} & \{\ \mathit{def\ app},\ k\ =\ k_1 \bullet k_3\ \} \\
= & \text{app}\ (k_1 \bullet k_3)\ (\mathsf{E}_i[e \mid x_1,\ \ldots,\ x_m]) & \{\ \mathit{spec}\ (b)\ \} \\
= & \text{app}\ k_1\ (\text{app}\ k_3\ (\mathsf{E}_i[e \mid x_1,\ \ldots,\ x_m])) & \{\ \mathit{inductive\ hypothesis}\ \} \\
= & \text{app}\ k_1\ (\text{app}\ (\mathsf{A}_i\ x_1\ \ldots\ x_m\ k_3)\ e) & \{\ \mathit{def\ app}\ \} \\
= & \text{app}\ k_1\ (\text{app}\ k_2\ e) & \{\ \mathit{def\ app}\ \}
\end{array}$$

For application we have:

$$\begin{array}{lll}
 & \text{app}\ (\text{ctx}\ \mathsf{E}_i)\ e & \\
= & \text{app}\ (\mathsf{A}_i\ x_1,\ \ldots\ x_m\ \mathit{Hole})\ e & \{\ \mathit{def\ ctx}\ \} \\
= & [\![\ \mathsf{E}_i[e \mid x_1,\ \ldots,\ x_m]\ ]\!]_{f,k} & \{\ \mathit{def\ app},\ k\ =\ \mathit{Hole}\ \} \\
= & \text{app}\ \mathit{Hole}\ (\mathsf{E}_i[e \mid x_1,\ \ldots\ x_m]) & \{\ \mathit{spec}\ (b)\ \} \\
= & \mathsf{E}_i[e \mid x_1,\ \ldots\ x_m] & \{\ \mathit{def\ app}\ \} \\
= & \mathsf{E}_i[e] &
\end{array}$$

$\square$

C.2 Context Laws for Right-biased-contexts

$$
\begin{array}{lll}
 & \mathsf{app}\ (k_1 \bullet k_2)\ e & \\
= & \mathsf{app}\ (k_2 \odot k_1)\ e & \{\ (\mathit{rcomp})\ \} \\
= & e \odot (k_2 \odot k_1) & \{\ (\mathit{rapp})\ \} \\
= & (e \odot k_2) \odot k_1 & \{\ \mathit{assoc.}\ \} \\
= & \mathsf{app}\ k_1\ (\mathsf{app}\ k_2\ e) & \{\ (\mathit{rapp})\ \}
\end{array}
$$

and for context application we have:

$$
\begin{array}{lll}
 & \mathsf{app}\ (\mathsf{ctx}\ \mathsf{A})\ e & \\
= & \mathsf{app}\ (\!|\mathsf{A}|\!)\ e & \{\ (\mathit{rctx})\ \} \\
= & e \odot (\!|\mathsf{A}|\!) & \{\ (\mathit{rapp})\ \}
\end{array}
$$

We proceed by induction over A.
Case $\mathsf{A} = \square$:

$$
\begin{array}{lll}
= & e \odot (\!|\square|\!) & \\
= & e \odot \mathit{unit} & \{\ \mathit{fold}\ \} \\
= & e & \{\ \mathit{unit}\ \} \\
= & \square[e] & \{\ \square\ \}
\end{array}
$$

and the case $\mathsf{A} = \mathsf{A}' \odot v$:

$$
\begin{array}{lll}
= & e \odot (\!|\mathsf{A}' \odot v|\!) & \\
= & e \odot ((\!|\mathsf{A}'|\!) \odot v) & \{\ \mathit{fold}\ \} \\
= & (e \odot (\!|\mathsf{A}'|\!)) \odot v & \{\ \mathit{assoc.}\ \} \\
= & \mathsf{A}'[e] \odot v & \{\ \mathit{induction\ hyp.}\ \} \\
= & \mathsf{A}[e] & \{\ \mathsf{A}\ \mathit{context}\ \}
\end{array}
$$

C.3 General Monoid Contexts

$$
\begin{array}{lll}
 & \mathsf{app}\ ((l_1,\ r_1) \bullet (l_2,\ r_2))\ e & \\
= & \mathsf{app}\ (l_1 \odot l_2,\ r_2 \odot r_1)\ e & \{\ (\mathit{acomp})\ \} \\
= & (l_1 \odot l_2) \odot e \odot (r_2 \odot r_1) & \{\ (\mathit{aapp})\ \} \\
= & (l_1 \odot (l_2 \odot e \odot r_2) \odot r_1) & \{\ \mathit{assoc.}\ \} \\
= & \mathsf{app}\ (l_1,\ r_1)\ (\mathsf{app}\ (l_2,\ r_2)\ e) & \{\ (\mathit{aapp})\ \}
\end{array}
$$

and

$$
\begin{array}{lll}
 & \mathsf{app}\ (\mathsf{ctx}\ \mathsf{A})\ e & \\
= & \mathsf{app}\ (\!|\mathsf{A}|\!)\ e & \{\ (\mathit{actx})\ \} \\
= & l \odot e \odot r & \{\ (\mathit{aapp}),\ \mathit{for}\ (l,\ r)\ =\ (\!|\mathsf{A}|\!)\ \}
\end{array}
$$

We proceed by induction over A: case $\mathsf{A} = \square$:

$$
\begin{array}{lll}
= & l \odot e \odot r & \{\ \mathit{for}\ (l,\ r)\ =\ (\!|\square|\!)\ \} \\
= & \mathit{unit} \odot e \odot \mathit{unit} & \{\ \mathit{fold}\ \} \\
= & e & \{\ \mathit{unit}\ \} \\
= & \square[e] & \{\ \square\ \}
\end{array}
$$

and $\mathsf{A} = v \odot \mathsf{A}'$:

$$
\begin{array}{lll}
= & l \odot e \odot r & \{\ \mathit{for}\ (l,\ r)\ =\ (\!|v \odot \mathsf{A}'|\!)\ \} \\
= & (v \odot l) \odot e \odot r & \{\ \mathit{fold},\ \mathit{for}\ (l,\ r)\ =\ (\!|\mathsf{A}'|\!)\ \} \\
= & v \odot (l \odot e \odot r) & \{\ \mathit{assoc.},\ \mathit{for}\ (l,\ r)\ =\ (\!|\mathsf{A}'|\!)\ \} \\
= & v \odot \mathsf{A}'[e] & \{\ \mathit{induction\ hyp.},\ \mathit{for}\ (l,\ r)\ =\ (\!|\mathsf{A}'|\!)\ \} \\
= & \mathsf{A}[e] & \{\ \mathsf{A}\ \mathit{context}\ \}
\end{array}
$$

and $\mathsf{A} = \mathsf{A}' \odot v$:

$$
\begin{array}{lll}
= & l \odot e \odot r & \{\ \mathit{for}\ (l,\ r)\ =\ (\!|\mathsf{A}' \odot v|\!)\ \} \\
= & l \odot e \odot (r \odot v) & \{\ \mathit{fold},\ \mathit{for}\ (l,\ r)\ =\ (\!|\mathsf{A}'|\!)\ \} \\
= & (l \odot e \odot r) \odot v & \{\ \mathit{assoc.},\ \mathit{for}\ (l,\ r)\ =\ (\!|\mathsf{A}'|\!)\ \} \\
= & \mathsf{A}'[e] \odot v & \{\ \mathit{induction\ hyp.},\ \mathit{for}\ (l,\ r)\ =\ (\!|\mathsf{A}'|\!)\ \} \\
= & \mathsf{A}[e] & \{\ \mathsf{A}\ \mathit{context}\ \}
\end{array}
$$

$\square$

C.4 Context Laws for Exponent Contexts

We prove the composition law by induction on k_2:

$$\begin{array}{cll}
 & \mathsf{app}\ (k_1 \bullet k_2)\ e & \\
= & \mathsf{app}\ (k_1 + k_2)\ e & \\
= & \mathsf{app}\ k_1\ e & \{\ \textit{case}\ k_2 = 0\ \} \\
= & \mathsf{app}\ k_1\ (\mathsf{app}\ 0\ e) & \{\ (\textit{xapp})\ \} \\
= & \mathsf{app}\ k_1\ (\mathsf{app}\ k_2\ e) & \{\ k_2 = 0\ \}
\end{array}$$

and

$$\begin{array}{cll}
 & \mathsf{app}\ (k_1 \bullet k_2)\ e & \\
= & \mathsf{app}\ (k_1 + (k' + 1))\ e & \{\ \textit{case}\ k_2 = k' + 1\ \} \\
= & \mathsf{app}\ ((k_1 + k') + 1)\ e & \{\ \textit{assoc.}\ \} \\
= & \mathsf{app}\ (k_1 + k')\ (g\ e) & \{\ (\textit{xapp})\ \} \\
= & \mathsf{app}\ k_1\ (\mathsf{app}\ k'\ (g\ e)) & \{\ \textit{inductive hyp.}\ \} \\
= & \mathsf{app}\ k_1\ (\mathsf{app}\ (k' + 1)\ e) & \{\ (\textit{xapp})\ \} \\
= & \mathsf{app}\ k_1\ (\mathsf{app}\ k_2\ e) & \{\ k_2 = k' + 1\ \}
\end{array}$$

Appliction can be derived as:

$$\begin{array}{cll}
 & \mathsf{app}\ (\mathsf{ctx}\ \mathsf{A})\ e & \\
= & \mathsf{app}\ \llparenthesis \mathsf{A} \rrparenthesis\ e & \{\ (\textit{xctx})\ \}
\end{array}$$

We proceed by induction over A: case $\mathsf{A} = \square$:

$$\begin{array}{cll}
= & \mathsf{app}\ \llparenthesis \square \rrparenthesis\ e & \\
= & \mathsf{app}\ 0\ e & \{\ \textit{fold}\ \} \\
= & e & \{\ (\textit{xapp})\ \} \\
= & \square[e] & \{\ \square\ \}
\end{array}$$

and $\mathsf{A} = g\ \mathsf{A}'$:

$$\begin{array}{cll}
= & \mathsf{app}\ \llparenthesis g\ \mathsf{A}' \rrparenthesis\ e & \\
= & \mathsf{app}\ (\llparenthesis \mathsf{A}' \rrparenthesis + 1)\ e & \{\ \textit{fold}\ \} \\
= & \mathsf{app}\ \llparenthesis \mathsf{A}' \rrparenthesis\ (g\ e) & \{\ (\textit{xapp})\ \} \\
= & \mathsf{A}'[g\ e] & \{\ \textit{induction hyp.}\ \} \\
= & \mathsf{A}[e] & \{\ \mathsf{A}\ \textit{context}\ \}
\end{array}$$

$\square$

C.5 Constructor Contexts

Composition:

$$\begin{array}{cll}
 & \mathsf{app}\ (k_1 \bullet k_2)\ e & \\
= & \mathsf{app}\ (k_1[k_2])\ e & \{\ (\textit{kcomp})\ \} \\
= & (k_1[k_2])[e] & \{\ (\textit{kapp})\ \} \\
= & k_1[k_2[e]] & \{\ \textit{contexts}\ \} \\
= & k_1[\mathsf{app}\ k_2\ e] & \{\ (\textit{kapp})\ \} \\
= & \mathsf{app}\ k_1\ (\mathsf{app}\ k_2\ e) & \{\ (\textit{kapp})\ \}
\end{array}$$

and application:

$$\begin{array}{cll}
 & \mathsf{app}\ (\mathsf{ctx}\ \mathsf{K})\ e & \\
= & \mathsf{app}\ \mathsf{K}\ e & \{\ (\textit{kctx})\ \} \\
= & \mathsf{K}[e] & \{\ (\textit{kapp})\ \}
\end{array}$$

$\square$

$$\frac{x : \tau \in \Gamma}{\Gamma\ ;\ \varnothing \vdash_{\mathsf{M}} x : \tau}\ [\textsc{var}] \qquad \frac{\Gamma|_{\mathsf{N}} \uplus \{x : \tau_1\}\ ;\ \varnothing \vdash_{\mathsf{M}} M : \tau_2}{\Gamma\ ;\ \varnothing \vdash_{\mathsf{M}} \lambda x : \tau_1 .\ M : \tau_1 \to \tau_2}\ [\textsc{abs}]$$

$$\frac{}{\Gamma\ ;\ x : \tau \vdash_{\mathsf{M}} x : \tau}\ [\textsc{hle}] \qquad \frac{\Gamma\ ;\ x : \tau_1 \vdash_{\mathsf{M}} M : \tau_2}{\Gamma\ ;\ \varnothing \vdash_{\mathsf{M}} \hat{\lambda} x : \tau_1 .\ M : (\tau_1,\ \tau_2)\ \mathit{hfun}}\ [\textsc{hfun}]$$

$$\frac{\Gamma_1\ ;\ \varnothing \vdash_{\mathsf{M}} M_1 : \tau_1 \to \tau_2 \qquad \Gamma_2\ ;\ \varnothing \vdash_{\mathsf{M}} M_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2\ ;\ \varnothing \vdash_{\mathsf{M}} M_1\ M_2 : \tau_2}\ [\textsc{app}]$$

$$\frac{\Gamma_1\ ;\ \varnothing \vdash_{\mathsf{M}} M_1 : (\tau_1,\ \tau_2)\ \mathit{hfun} \qquad \Gamma_2\ ;\ H \vdash_{\mathsf{M}} M_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2\ ;\ H \vdash_{\mathsf{M}} \mathit{happ}\ M_1\ M_2 : \tau_2}\ [\textsc{happ}]$$

$$\frac{\Gamma_i\ ;\ H_i \vdash_{\mathsf{M}} M_i : \tau_i \qquad C^k : \tau_1 \to \ldots \to \tau_k \to \tau}{\uplus_i\, \Gamma_i\ ;\ \oplus_i\, H_i \vdash_{\mathsf{M}} C^k\ M_1 \ldots M_k : \tau}\ [\textsc{cons}] \qquad \frac{C^k : \tau \in \Gamma}{\Gamma \vdash_{\mathsf{M}} C^k : \tau}\ [\textsc{con}]$$

$$\frac{\Gamma_1\ ;\ \varnothing \vdash_{\mathsf{M}} M : \tau_1 \qquad \Gamma_2\ ;\ \varnothing \vdash_{\mathsf{PAT}} p_i : \tau_1 \mapsto M_i : \tau_2}{\Gamma_1 \uplus \Gamma_2\ ;\ \varnothing \vdash_{\mathsf{M}} \mathsf{match}\ M\ \{\ \overline{p_i \mapsto M_i}\ \} : \tau_2}\ [\textsc{match}] \qquad \frac{f : \tau \in \Gamma}{\Gamma \vdash_{\mathsf{M}} f : \tau}\ [\textsc{fun}]$$

$$\frac{\Gamma, f : \tau\ ;\ \varnothing \vdash_{\mathsf{M}} \lambda x.\ e : \tau}{\Gamma\ ;\ \varnothing \vdash_{\mathsf{M}} \mathsf{fun}\ f = \lambda x.\ e : \tau}\ [\textsc{fundecl}] \qquad \frac{\begin{array}{c}\Gamma\ ;\ \varnothing \vdash_{\mathsf{M}} C^k : \tau_1 \to \ldots \to \tau_k \to \tau \\ \Gamma, x_1 : \tau_1, \ldots, x_k : \tau_k\ ;\ \varnothing \vdash M_i : \tau'\end{array}}{\Gamma\ ;\ \varnothing \vdash_{\mathsf{PAT}} C^k\ x_1 \ldots x_k : \tau \mapsto e_i : \tau'}\ [\textsc{pat}]$$

$$\frac{\Gamma_1\ ;\ \varnothing \vdash_{\mathsf{M}} M_1 : \tau_1 \qquad \Gamma_2, x : \tau_1\ ;\ \varnothing \vdash_{\mathsf{M}} M_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2\ ;\ \varnothing \vdash_{\mathsf{M}} \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \tau_2}\ [\textsc{let}]$$

Fig. 5. Minamide's type system adapted to our language

C.6 Constructor Contexts and Minamide

The hole calculus is restricted by a linear type discipline where the contexts $\mathit{ctx}\ \alpha \equiv \mathit{hfun}\ \alpha\ \alpha$ have a linear type. This is what enables an efficient in-place update implementation while still having a pure functional interface. For our needs, we need to check separately that the translation ensures that all uses of a context k are indeed linear. Type judgements in Minamide's system (Minamide, 1998, fig. 4) are denoted as $\Gamma\ ;\ H \vdash_{\mathsf{M}} e : \tau$ where Γ is the normal type environment, and H the linear one containing at most one linear value. The type environment Γ can still contain linear values with a linear type but only pass those to one of the premises. The environment restricted to non-linear values is denoted at $\Gamma|_{\mathsf{N}}$. We can now show that our translation can be typed in Minamide's system:

Lemma 4. (*TRMC uses contexts linearly*)
If $\Gamma|_{\mathsf{N}}\ ;\ \varnothing \vdash_{\mathsf{M}} \mathsf{fun}\ f = \lambda xs.\ e : \tau_1 \to \ldots \to \tau_n \to \tau$ and k fresh
then $\Gamma|_{\mathsf{N}}, f\ ;\ \varnothing \vdash_{\mathsf{M}} \mathsf{fun}\ \hat{f} = \lambda xs.\ \lambda k.\ [\![e]\!]_{f,k} : \tau_1 \to \ldots \to \tau_n \to ((\tau,\ \tau)\ \mathit{hfun}) \to \tau$.

To show this, we need a variant of the general replacement lemma (Wright and Felleisen, 1994, Lemma 4.2; Hindley and Seldin, 1986, Lemma 11.18) to reason about linear substitution in an evaluation context:

Lemma 5. (*Linear replacement*)
If $\Gamma|_{\mathsf{N}}\ ;\ \varnothing \vdash_{\mathsf{M}} \mathsf{K}[e] : \tau$ for a constructor context K then there is a sub-deduction $\Gamma|_{\mathsf{N}}\ ;\ \varnothing \vdash_{\mathsf{M}} e : \tau'$

at the hole and $\Gamma|_\mathsf{N}\ ;\ x : \tau' \vdash_\mathsf{M} \mathsf{K}[x] : \tau$.

Proof. By induction over the constructor context K.

Case $\square$.

$$
\begin{array}{ll}
\Gamma|_\mathsf{N}\ ;\ \varnothing \vdash_\mathsf{M} \square[e] : \tau & \{\ \textit{assumption}\ \} \\
\Gamma|_\mathsf{N}\ ;\ \varnothing \vdash_\mathsf{M} e : \tau & \{\ \textit{subject reduction}\ \} \\
\Gamma|_\mathsf{N}\ ;\ x : \tau \vdash_\mathsf{M} x : \tau & \{\ [\textit{hle}]\ \} \\
\Gamma|_\mathsf{N}\ ;\ x : \tau \vdash_\mathsf{M} \square[x] : \tau' & \{\ \textit{definition}\ \} \\
\Gamma|_\mathsf{N}\ ;\ x : \tau \vdash_\mathsf{M} \mathsf{E}[x] : \tau' & \{\ \textit{definition}\ \}
\end{array}
$$

Case $C^k\ w_1 \ldots \mathsf{K}' \ldots w_k$.

$$
\begin{array}{ll}
\Gamma|_\mathsf{N}\ ;\ \varnothing \vdash_\mathsf{M} C^k\ w_1 \ldots \mathsf{K}'[e] \ldots w_k : \tau & \{\ \textit{assumption}\ \} \\
\Gamma|_\mathsf{N}\ ;\ \varnothing \vdash_\mathsf{M} w_i : \tau_i \quad \text{for } i \neq j & \{\ [\textit{cons}]\ \textit{and nonlinearity}\ \} \\
\Gamma|_\mathsf{N}\ ;\ \varnothing \vdash_\mathsf{M} \mathsf{K}'[e] : \tau_j & \{\ [\textit{cons}]\ \} \\
\Gamma|_\mathsf{N}\ ;\ x : \tau' \vdash_\mathsf{M} \mathsf{K}'[x] : \tau_j & \{\ \textit{inductive hypothesis}\ \} \\
\Gamma|_\mathsf{N}\ ;\ x : \tau' \vdash_\mathsf{M} C^k\ w_1 \ldots \mathsf{K}'[x] \ldots w_k : \tau & \{\ [\textit{cons}]\ \}
\end{array}
$$

Again we see that our maximal context is an evaluation context as we would not be able to derive the Lemma for contexts under lambda's for example (as the linear type environment is not propagated under lambda's).

Proof. (*Of Theorem 4*) By the FUNDECL and ABS rules we obtain:

$$
\begin{array}{ll}
\Gamma_1 = \Gamma|_\mathsf{N}, f : \tau_1 \to \ldots \to \tau_n \to \tau,\ x_1 : \tau_1, \ldots, x_n : \tau_n & \\
\Gamma_1\ ;\ \varnothing \vdash_\mathsf{M} e : \tau & \{\ \textit{inductive property}\ \}
\end{array}
$$

By the FUNDECL and ABS rules, we need to derive:

$$
\begin{array}{l}
\Gamma_2 = \Gamma|_\mathsf{N},\ f : \tau_1 \to \ldots \to \tau_n \to \tau,\ \hat{f} : \tau_1 \to \ldots \to \tau_n \to ((\tau, \tau)\ \textit{hfun}) \to \tau,\ x_1 : \tau_1, \ldots, x_n : \tau_n \\
\Gamma_2,\ k : ((\tau, \tau)\ \textit{hfun})\ ;\ \varnothing \vdash_\mathsf{M} [\![e]\!]_{f,k} : \tau
\end{array}
$$

In particular, we have $\Gamma_1 \subseteq \Gamma_2$. We proceed by induction over the translation function while maintaining the inductive property.

Case (*base*).

$$[\![e]\!]_{f,k} = \mathsf{app}\ k\ e = \textit{happ}\ k\ e$$

$$
\begin{array}{ll}
k : (\tau, \tau)\ \textit{hfun}\ ;\ \varnothing \vdash_\mathsf{M} k : (\tau, \tau)\ \textit{hfun} & \{\ [\textit{hle}]\ \} \\
\Gamma_1\ ;\ \varnothing \vdash_\mathsf{M} e : \tau & \{\ \textit{assumption}\ \} \\
\Gamma_2\ ;\ \varnothing \vdash_\mathsf{M} e : \tau & \{\ \textit{weakening}\ \} \\
\Gamma_2,\ k : (\tau, \tau)\ \textit{hfun}\ ;\ \varnothing \vdash_\mathsf{M} \textit{happ}\ k\ e & \{\ [\textit{happ}]\ \}
\end{array}
$$

Case (*tail*), $e = \mathsf{K}[f\ e_1 \ldots e_n]$.

$$[\![e]\!]_{f,k} = \hat{f}\ e_1 \ldots e_n\ (k \bullet \mathsf{ctx}\ \mathsf{K}) = \hat{f}\ e_1 \ldots e_n\ (\textit{hcomp}\ k\ (\hat{\lambda} x.\ \mathsf{K}[x]))$$

$$
\begin{array}{ll}
\Gamma_1\ ;\ \varnothing \vdash_\mathsf{M} \mathsf{K}[f\ e_1 \ldots e_n] : \tau & \{\ \textit{assumption}\ \} \\
\Gamma_2\ ;\ \varnothing \vdash_\mathsf{M} \mathsf{K}[f\ e_1 \ldots e_n] : \tau & \{\ \textit{weakening}\ \} \\
\Gamma_2\ ;\ x : \tau' \vdash_\mathsf{M} \mathsf{K}[x] : \tau & \{\ \textit{linear replacement with nonlinearity of}\ \Gamma_2\ \} \\
\Gamma_2\ ;\ \varnothing \vdash_\mathsf{M} \hat{\lambda} x.\ \mathsf{K}[x] : (\tau, \tau)\ \textit{hfun} & \{\ [\textit{hfun}]\ \} \\
\Gamma_2,\ k : (\tau, \tau)\ \textit{hfun}\ ;\ \varnothing \vdash_\mathsf{M} \textit{hcomp}\ k\ (\hat{\lambda} x.\ \mathsf{K}[x]) : (\tau, \tau)\ \textit{hfun} & \{\ \textit{hcomp},\ [\textit{happ}],\ [\textit{hfun}]\ \} \\
\Gamma_2\ ;\ \varnothing \vdash_\mathsf{M} f\ e_1 \ldots e_n : \tau' & \{\ \textit{linear replacement with nonlinearity of}\ \Gamma_2\ \} \\
\Gamma_2\ ;\ \varnothing \vdash_\mathsf{M} e_i : \tau_i & \{\ [\textit{app}]\ \} \\
\Gamma_2,\ k : (\tau, \tau)\ \textit{hfun}\ ;\ \varnothing \vdash_\mathsf{M} \hat{f}\ e_1 \ldots e_n\ (\textit{hcomp}\ k\ (\hat{\lambda} x.\ \mathsf{K}[x])) & \{\ [\textit{app}]\ \}
\end{array}
$$

Case (*let*), $e = \text{let } x = e_1 \text{ in } e_2$.

$[\![e]\!]_{f,k} = \text{let } x = e_1 \text{ in } [\![e_2]\!]_{f,k}$

$$
\begin{array}{ll}
\Gamma_1 \;\; ; \varnothing \vdash_{\mathsf{M}} \text{let } x = e_1 \text{ in } e_2 : \tau & \{ \textit{ assumption } \} \\
\Gamma_1 \;\; ; \varnothing \vdash_{\mathsf{M}} e_1 : \tau_1 & \{ [\textit{let}] \} \\
\Gamma_2 \;\; ; \varnothing \vdash_{\mathsf{M}} e_1 : \tau_1 & \{ \textit{ weakening } \} \\
\Gamma_1, \; x : \tau_1 ; \varnothing \vdash_{\mathsf{M}} e_2 : \tau & \{ [\textit{let}] \} \\
\Gamma_2, \; k : (\tau, \tau) \; \textit{hfun}, \; x : \tau_1 ; \varnothing \vdash_{\mathsf{M}} [\![e_2]\!]_{f,k} : \tau & \{ \textit{ inductive hypothesis } \} \\
\Gamma_2, \; k : (\tau, \tau) \; \textit{hfun}, \; ; \varnothing \vdash_{\mathsf{M}} \text{let } x = e_1 \text{ in } [\![e_2]\!]_{f,k} : \tau & \{ [\textit{let}] \}
\end{array}
$$

Case (*match*), $e = \text{match } e_1 \{ p_i \mapsto e_i \}$.

$[\![e]\!]_{f,k} = \text{match } e_1 \{ p_i \mapsto [\![e_i]\!]_{f,k} \}$

$$
\begin{array}{ll}
\Gamma_1 \;\; ; \varnothing \vdash_{\mathsf{M}} \text{match } e_1 \{ p_i \mapsto e_i \} : \tau & \{ \textit{ assumption } \} \\
\Gamma_1 \;\; ; \varnothing \vdash_{\mathsf{M}} e_1 : \tau' & \{ [\textit{match}] \} \\
\Gamma_2 \;\; ; \varnothing \vdash_{\mathsf{M}} e_1 : \tau' & \{ \textit{ weakening } \} \\
\Gamma_1 \;\; ; \varnothing \vdash_{\mathsf{PAT}} p_i \mapsto e_i : \tau & \{ [\textit{match}] \} \\
\Gamma_1 \;\; ; \varnothing \vdash_{\mathsf{M}} C^k : \tau_1 \to \ldots \to \tau_k \to \tau' & \{ [\textit{pat}] \} \\
\Gamma_1, \; x_1 : \tau_1, \ldots, x_k : \tau_k ; \varnothing \vdash_{\mathsf{M}} e_i : \tau & \{ [\textit{pat}] \} \\
\Gamma_2, \; k : (\tau, \tau) \; \textit{hfun}, \; x_1 : \tau_1, \ldots, x_k : \tau_k ; \varnothing \vdash_{\mathsf{M}} [\![e_i]\!]_{f,k} : \tau & \{ \textit{ inductive hypothesis } \} \\
\Gamma_2, \; k : (\tau, \tau) \; \textit{hfun} ; \varnothing \vdash_{\mathsf{PAT}} p_i \mapsto [\![e_i]\!]_{f,k} : \tau & \{ [\textit{pat}] \} \\
\Gamma_2, \; k : (\tau, \tau) \; \textit{hfun} ; \varnothing \vdash_{\mathsf{M}} \text{match } e_1 \{ p_i \mapsto [\![e_i]\!]_{f,k} \} : \tau & \{ [\textit{match}] \}
\end{array}
$$

□

C.7 Contexts Form Linear Chains

Proof. (*Of Lemma 2*) By induction on the shape of K:

Case $C \ldots \square_i \ldots$:

$$
\begin{array}{lll}
 & H \mid C \ldots \square_i \ldots & \\
\longrightarrow_{\mathsf{r}}^{*} & H, \; x \mapsto^1 C \ldots \square_i \ldots \mid x & \{ (\textit{con}_r) \} \\
= & H, \; [x \mapsto^1 C \ldots \square_i \ldots]^1_x \mid x & \{ \textit{ linear chain } \}
\end{array}
$$

Case $C \ldots \mathsf{K}'[C' \ldots \square_i \ldots] \ldots$

$$
\begin{array}{lll}
 & H \mid (\!| C \ldots \mathsf{K}'[C' \ldots \square_i \ldots] \ldots |\!) & \\
\longrightarrow_{\mathsf{r}}^{*} & H, \; [H', y \mapsto^1 C' \ldots \square_i \ldots]^1_{x'} \mid \langle C \ldots x' \ldots \rangle & \{ \textit{ induction hyp. } \} \\
\longrightarrow_{\mathsf{r}} & H, \; x \mapsto^1 C \ldots x' \ldots, \; [H', \; y \mapsto^1 C' \ldots \square_i \ldots]^1_{x'} \mid x & \{ (\textit{con}_r) \} \\
= & H, \; [x \mapsto^1 C \ldots x' \ldots, \; H', \; y \mapsto^1 C' \ldots \square_i \ldots]^1_x \mid x & \{ \textit{ linear chain } \}
\end{array}
$$

□

C.8 Deriving Constructor Context Fold

Given the specification:

$$(\textit{foldspec}) \quad H \mid (\!| \mathsf{K}[C \ldots \square_i \ldots] |\!) \;\cong\; H \mid \text{let } x = \mathsf{K}[C \ldots \square_i \ldots] \text{ in } \langle x, [x]@i \rangle$$

we can calculate the fold using induction over the shape of K. In the case that $\mathsf{K} = \Box$, we derive:

$$
\begin{array}{lll}
 & H \mid \llparenthesis C \ldots \Box_i \ldots \rrparenthesis & \\
\cong & H \mid \mathsf{let}\ x = C \ldots \Box_i \ldots\ \mathsf{in}\ \langle x, [x]@i\rangle & \{\ \textit{specification}\ \} \\
\cong & H, x \mapsto^1 C \ldots \Box_i \ldots \mid \langle x, [x]@i\rangle & \{\ (\textit{let}_r),\ (\textit{con}_r),\ \mathbf{1}\ \} \\
= & H, [x \mapsto^1 C \ldots \Box_i \ldots]^1_x \mid \langle x, [x]@i\rangle & \{\ \textit{linear chain}\ \} \\
= & H, [x \mapsto^1 C \ldots \Box_i \ldots]^1_x \mid \langle x, x@i\rangle & \{\ \textit{def.}\ \} \\
\cong & H \mid \mathsf{let}\ x = C \ldots \Box_i \ldots\ \mathsf{in}\ \langle x, x@i\rangle & \{\ (\textit{let}_r),\ (\textit{con}_r),\ \mathbf{1}\ \}
\end{array}
$$

and otherwise, K has the form $C' \ldots \mathsf{K}' \ldots$ where $\llparenthesis \mathsf{K}'[C \ldots \Box_i \ldots] \rrparenthesis = \mathsf{let}\ x = \mathsf{K}'[C \ldots \Box_i \ldots]\ \mathsf{in}\ \langle x, [x]@i\rangle$ (by induction):

$$
\begin{array}{lll}
 & H \mid \llparenthesis C' \ldots \mathsf{K}'[C \ldots \Box_i \ldots] \ldots \rrparenthesis & \\
\cong & H \mid \mathsf{let}\ x = C' \ldots \mathsf{K}'[C \ldots \Box_i \ldots] \ldots\ \mathsf{in}\ \langle x, [x]@i\rangle & \{\ \textit{specification}\ \} \\
\cong & H \mid \mathsf{let}\ z = \mathsf{K}'[C \ldots \Box_i \ldots]\ \mathsf{in}\ \mathsf{let}\ x = C \ldots z \ldots\ \mathsf{in}\ \langle x, [x]@i\rangle & \{\ (\textit{let}_r)\ \} \\
\cong & H \mid \mathsf{let}\ \langle z, [z]@i\rangle = \llparenthesis \mathsf{K}'[C \ldots \Box_i \ldots] \rrparenthesis\ \mathsf{in}\ \mathsf{let}\ x = C \ldots z \ldots\ \mathsf{in}\ \langle x, [x]@i\rangle & \{\ \textit{calculate}\ \} \\
\cong & H,\ [H', y \mapsto^1 C \ldots \Box_i \ldots]^1_z,\ x \mapsto^1 C \ldots z \ldots \mid \langle x, [x.i]\rangle & \{\ (\textit{let}_r),\ \textit{lemma}\ 2,\ \mathbf{1}\ \} \\
= & H,\ [x \mapsto^1 C \ldots z \ldots,\ [H', y \mapsto^1 C \ldots \Box_i \ldots]^1_z]^1_x, \mid \langle x, [x]@i\rangle & \{\ \textit{linear chain}\ \} \\
= & H,\ [x \mapsto^1 C \ldots z \ldots,\ [H', y \mapsto^1 C \ldots \Box_i \ldots]^1_z]^1_x, \mid \langle x, y@i\rangle & \{\ \textit{def.}\ \} \\
\cong & H \mid \mathsf{let}\ \langle z, y@i\rangle = \llparenthesis \mathsf{K}'[C \ldots \Box_i \ldots] \rrparenthesis\ \mathsf{in}\ \langle C \ldots z \ldots,\ y@i\rangle & \{\ (\textit{let}_r),\ (\textit{con}_r)\ (1)\ \}
\end{array}
$$

C.9 Deriving Constructor Context Composition

We can calculate for a $\mathsf{K}_1, \mathsf{K}_2 \neq \Box$:

$$
\begin{array}{lll}
 & H \mid \mathsf{app}\ (\mathsf{ctx}\ \mathsf{K}_1 \bullet \mathsf{ctx}\ \mathsf{K}_2)\ e & \\
\cong & H \mid \mathsf{app}\ (\mathsf{let}\ x_1 = \mathsf{K}_1[\Box]\ \mathsf{in}\ \langle x_1, [x_1]@i\rangle) \bullet (\mathsf{ctx}\ \mathsf{K}_2)\ e & \{\ \textit{fold specification},\ \mathsf{K}_1 \neq \Box\ \} \\
\cong & H,\ [H_1, y_1 \mapsto^1 C_1 \ldots \Box_i \ldots]^1_{x_1} \mid \mathsf{app}\ (\langle x_1, [x_1]@i\rangle \bullet \mathsf{ctx}\ \mathsf{K}_2)\ e & \{\ \textit{lemma}\ 2,\ \mathbf{1}\ \} \\
\cong & H,\ [H_1, y_1 \mapsto^1 C_1 \ldots \Box_i \ldots]^1_{x_1},\ [H_2, y_2 \mapsto^1 C_2 \ldots \Box_j \ldots]^1_{x_2} & \\
 & \mid \mathsf{app}\ (\langle x_1, [x_1]@i\rangle \bullet \langle x_2, [x_2]@j\rangle)\ e & \{\ \textit{fold specification},\ \mathsf{K}_2 \neq \Box,\ \textit{lemma}\ 2, \\
= & H,\ [H_1, y_1 \mapsto^1 C_1 \ldots \Box_i \ldots]^1_{x_1},\ [H_2, y_2 \mapsto^1 C_2 \ldots \Box_j \ldots]^1_{x_2} & \\
 & \mid \mathsf{app}\ (\langle x_1, y_1@i\rangle \bullet \langle x_2, y_2@j\rangle)\ e & \{\ \textit{def.}\ \} \\
= & H,\ [H_1, y_1 \mapsto^1 C_1 \ldots \Box_i \ldots]^1_{x_1},\ [H_2, y_2 \mapsto^1 C_2 \ldots \Box_j \ldots]^1_{x_2} & \\
 & \mid \mathsf{app}\ \langle \mathsf{app}\ \langle x_1, y_1@i\rangle,\ y_2@j\rangle\ e & \{\ \textit{calculate}\ \} \\
= & H,\ [H_1, y_1 \mapsto^1 C_1 \ldots x_2 \ldots]^1_{x_1},\ [H_2, y_2 \mapsto^1 C_2 \ldots \Box_j \ldots]^1_{x_2} & \\
 & \mid \mathsf{app}\ \langle x_1, y_2@j\rangle)\ e & \{\ (\textit{uapp})\ \} \\
= & H,\ [H_1, y_1 \mapsto^1 C_1 \ldots x_2 \ldots]^1_{x_1},\ [H_2, y_2 \mapsto^1 C_2 \ldots \Box_j \ldots]^1_{x_2},\ z \mapsto^1 v & \\
 & \mid \mathsf{app}\ \langle x_1, y_2@j\rangle\ z & \{\ e\ \textit{terminating},\ \mathbf{3}\ \} \\
\cong & H,\ [H_1, y_1 \mapsto^1 C_1 \ldots \Box_i \ldots]^1_{x_1},\ [H_2, y_2 \mapsto^1 C_2 \ldots z \ldots]^1_{x_2},\ z \mapsto^1 v & \\
 & \mid \mathsf{app}\ \langle x_1, y_1@i\rangle\ x_2 & \{\ (\textit{app})\ \} \\
\cong & H,\ [H_1, y_1 \mapsto^1 C_1 \ldots \Box_i \ldots]^1_{x_1},\ [H_2, y_2 \mapsto^1 C_2 \ldots \Box_j \ldots]^1_{x_2},\ z \mapsto^1 v & \\
 & \mid \mathsf{app}\ \langle x_1, y_1@i\rangle\ (\mathsf{app}\ \langle x_2, y_2@j\rangle\ z) & \{\ (\textit{app})\ \} \\
\cong & H,\ [H_1, y_1 \mapsto^1 C_1 \ldots \Box_i \ldots]^1_{x_1},\ [H_2, y_2 \mapsto^1 C_2 \ldots z \ldots]^1_{x_2} & \\
 & \mid \mathsf{app}\ \langle x_1, y_1@i\rangle\ (\mathsf{app}\ \langle x_2, y_2.j)\ e & \{\ (3)\ \} \\
\cong & H,\ [H_1, y_1 \mapsto^1 C_1 \ldots \Box_i \ldots]^1_{x_1} \mid \mathsf{app}\ \langle x_1, y_1@i\rangle\ (\mathsf{app}\ \mathsf{ctx}\ \mathsf{K}_2)\ e & \{\ (2)\ \} \\
\cong & H \mid \mathsf{app}\ (\mathsf{ctx}\ \mathsf{K}_1)\ (\mathsf{app}\ (\mathsf{ctx}\ \mathsf{K}_2)\ e) & \{\ (1)\ \}
\end{array}
$$

and thus define composition as:

$$(\textit{ucomp})\ H \mid \langle x_1, y_1@i\rangle \bullet \langle x_2, y_2@j\rangle \quad \longrightarrow_{\mathrm{r}} \quad H \mid \langle \mathsf{app}\ \langle x_1, y_1@i\rangle\ x_2,\ y_2@j\rangle$$

In case the context is empty, we can calculate immediately:

$$
\begin{array}{lll}
 & H \mid \mathsf{app}\ (\mathsf{ctx}\ \Box)\ e & \\
= & H \mid \mathsf{app}\ (\!|\Box|\!)\ e & \{\ \textit{def.}\ \} \\
\cong & H \mid \mathsf{app}\ \langle\rangle\ e & \{\ \textit{fold specification}\ \} \\
\cong & H \mid e & \{\ \textit{calculate}\ \} \\
= & H \mid \Box[e] & \{\ \textit{contex}\ \}
\end{array}
$$

For the empty contexts we can calculate for application:

$$
\begin{array}{lll}
 & \mathsf{app}\ (\mathsf{ctx}\ \Box \bullet \mathsf{ctx}\ \mathsf{K}_2)\ e & \\
= & \mathsf{app}\ ((\!|\Box|\!) \bullet \mathsf{ctx}\ \mathsf{K}_2)\ e & \{\ \textit{def.}\ \} \\
\cong & \mathsf{app}\ (\langle\rangle \bullet \mathsf{ctx}\ \mathsf{K}_2)\ e & \{\ \textit{fold specification}\ \} \\
\cong & \mathsf{app}\ (\mathsf{ctx}\ \mathsf{K}_2)\ e & \{\ \textit{calculate}\ \} \\
\cong & \mathsf{K}[e] & \{\ (\textit{appctx})\ \} \\
\cong & \Box[\mathsf{K}[e]] & \{\ \textit{contexts}\ \}
\end{array}
$$

and similarly for $\mathsf{K}_2 = \Box$ (but note that in our translation we never have $k \bullet \mathsf{ctx}\ \Box$).

C.10 Soundness of the Hybrid Approach

We need to show the context laws still hold for the hybrid approach.

At runtime, a context K is always a linear chain resulting from the fold or composition. We write $H \mid \hat{\mathsf{K}}$ for a non-empty context $[H', y \mapsto^m C \ldots \Box_i \ldots]^n_x \mid \langle x, y@i\rangle$ if we have $H_0 \mid (\!|\mathsf{K}|\!) \cong H_0,\ [H', y \mapsto^m C \ldots$

Application:

$$
\begin{array}{lll}
 & H \mid \mathsf{app}\ \hat{\mathsf{K}}\ e & \\
= & H,\ [H', y \mapsto^m C \ldots \Box_i \ldots]^{n+1}_x \mid \mathsf{app}\ \langle x, y@i\rangle\ e & \{\ (\mathrm{A}),\ \mathbf{1}\ \} \\
\cong & H,\ z \mapsto^1 v,\ [H', y \mapsto^m C_i \ldots \Box_i \ldots]^{n+1}_x \mid \mathsf{app}\ \langle x, y@i\rangle\ z & \{\ \textit{e is terminating}\ \mathbf{2}\ \} \\
= & H,\ z \mapsto^1 v,\ [H', y \mapsto^m C_i \ldots \Box_i \ldots]^{n+1}_x \mid \mathsf{append}\ x\ z & \{\ \textit{calculate}\ \}
\end{array}
$$

Now proceed by induction on H'. $H' = \Box$:

$$
\begin{array}{lll}
 & H,\ z \mapsto^1 v,\ [y \mapsto^{n+1} C_i \ldots \Box_i \ldots]^{n+1}_y \mid \mathsf{append}\ y\ z & \{\ \textit{singleton}\ \} \\
\cong & H,\ z \mapsto^1 v,\ [y \mapsto^{n+1} C_i \ldots \Box_i \ldots]^{n+1}_y \mid y.i\ \mathsf{as}\ z & \{\ \textit{calculate}\ \} \\
\cong & H,\ z \mapsto^1 v,\ [y \mapsto^n C_i \ldots \Box_i \ldots]^n_y,\ [x' \mapsto^1 C_i \ldots z \ldots]^1_{x'} \mid x' & \{\ (\textit{as})\ \} \\
\cong & H,\ z \mapsto^1 v,\ [y \mapsto^n C_i \ldots \Box_i \ldots]^n_y \mid \hat{\mathsf{K}}[z] & \{\ (1)\ \} \\
\cong & H,\ [y \mapsto^n C_i \ldots \Box_i \ldots]^n_y \mid \hat{\mathsf{K}}[e] & \{\ (2)\ \}
\end{array}
$$

and

$$
\begin{array}{lll}
& H,\ z \mapsto^1 v,\ [x \mapsto^{n+1} C' \ldots y_i \ldots, [H_1]^1_y]^{n+1}_x & \\
& \mid \text{append } x\ z & \\
\cong & H,\ z \mapsto^1 v,\ [x \mapsto^{n+1} C' \ldots y_i \ldots, [H_1]^1_y]^{n+1}_x & \\
& \mid \text{dup } y_i;\ x.i \text{ as } (\text{append } y_i\ z) & \{\ (\mathit{append})\ \} \\
\cong & H,\ z \mapsto^1 v,\ [x \mapsto^{n+1} C' \ldots y_i \ldots, [H_1]^2_y]^{n+1}_x & \\
& \mid x.i \text{ as } (\text{append } y\ z) & \\
\cong & H,\ z \mapsto^1 v,\ [x \mapsto^{n+1} C' \ldots y_i \ldots, [H_1]^1_y]^{n+1}_x,\ [H_2]^1_{y'} & \\
& \mid x.i \text{ as } y' & \{\ \mathit{induction\ hyp.}\ \} \\
\cong & H,\ z \mapsto^1 v,\ [x \mapsto^{n} C' \ldots y_i \ldots, [H_1]^1_y]^{n}_x,\ [x' \mapsto^1 C' \ldots y'_i \ldots,\ [H'']^1_{y'}]^1_{x'} & \\
& \mid x' & \{\ (\mathit{as})\ \} \\
\cong & H,\ z \mapsto^1 v,\ [x \mapsto^{n} C' \ldots y_i \ldots, [H_1]^1_y]^{n}_x & \\
& \mid C' \ldots \hat{\mathsf{K}}'[z] \ldots & \\
\cong & H,\ z \mapsto^1 v,\ [x \mapsto^{n} C' \ldots y_i \ldots, [H_1]^1_y]^{n}_x & \\
& \mid \hat{\mathsf{K}}[z] & \\
\cong & H,\ [x \mapsto^{n} C' \ldots y_i \ldots, [H_1]^1_y]^{n}_x & \\
& \mid \hat{\mathsf{K}}[e] & \{\ (2)\ \}
\end{array}
$$

D TYPING DEFUNCTIONALIZED CONTEXTS

In general, defunctionalized programs can not always be typed in polymorphic type systems like ML or System F. As Pottier and Gauthier (2004) show, the problem arises in the definition of app, as the type of the second argument may depend on the first. For the CPS-transform this is no problem:

```
fun plusOne(x : int)
  1 + x

fun consOne(xs : list<int>)
  Cons(1, xs)

fun app(k : a -> b, x : a) : b
  k(x)

app(plusOne, 1)
app(consOne, [])
```

But when we defunctionalize it, we can no longer type app:

```
type defun<a>
  PlusOne
  ConsOne

fun app(k : defun<a>, x : a) : a
  match k
    PlusOne -> 1 + x // type error, can not unify a with int
    ConsOne -> Cons(1, x) // type error, can not unify a with list<int>

app(PlusOne, 1)
app(ConsOne, [])
```

Pottier and Gauthier (2004) propose an easy fix: Just use GADTs! In pseudo-code:

```
type defun<a>
  PlusOne : defun<int>
  ConsOne : defun<list<int>>
```

This indeed fixes the problem. However, our approach can even work without GADTs, as we restrict the type of the hole as `a` in `ctx<a>`. While this can not handle polymorphic recursion, it is general enough for all examples in the paper – and ensures that our translation can be typed in System F. In this section, we want to show the last claim in formal.

Types

τ	::=	α	(type variable)
	|	$\mathsf{T}^k\ \tau_1 \ldots \tau_k$	(fully applied type constructors)
	|	$\tau \to \tau$	(functions)
	|	$\forall \alpha.\ \tau$	(abstraction)

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.\ e : \sigma \to \tau}\ [\text{LAM}] \qquad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ [\text{VAR}]$$

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau}\ [\text{APP}] \qquad \frac{C^k : \sigma \in \Gamma}{\Gamma \vdash C^k : \sigma}\ [\text{CON}]$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2}\ [\text{LET}] \qquad \frac{f : \sigma \in \Gamma}{\Gamma \vdash f : \sigma}\ [\text{FUN}]$$

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash_{\text{PAT}} p_i : \sigma \mapsto e_i : \tau}{\Gamma \vdash \mathsf{match}\ e\ \{\ \overline{p_i \mapsto e_i}\ \} : \tau}\ [\text{MATCH}] \qquad \frac{\Gamma,\ \alpha\ \mathsf{type} \vdash e : \sigma}{\Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\ \sigma}\ [\text{GEN}]$$

$$\frac{\begin{array}{c}\Gamma \vdash C^k : \sigma_1 \to \ldots \to \sigma_k \to \sigma \\ \Gamma, x_1 : \sigma_1, \ldots, x_k : \sigma_k \vdash e_i : \tau\end{array}}{\Gamma \vdash_{\text{PAT}} C^k\ x_1 \ldots x_k : \sigma \mapsto e_i : \tau}\ [\text{PAT}] \qquad \frac{\Gamma \vdash e : \forall\alpha.\ \sigma}{\Gamma \vdash e\ \tau : \sigma[\alpha{:=}\tau]}\ [\text{INST}]$$

$$\frac{\overline{\alpha}\ \mathsf{type},\ \tau\ \mathsf{type},\ f : \forall\overline{\alpha}.\ \tau \vdash \lambda x.\ e : \tau \quad \overline{\alpha} = \mathsf{ftv}(\tau)}{\Gamma \vdash \mathsf{fun}\ f = \Lambda\overline{\alpha}.\ \lambda x.\ e : \forall\overline{\alpha}.\ \tau}\ [\text{FUNDECL}]$$

Fig. 6. Typing rules for System F in our language

First, let us define the `accum` type. Formally, we introduce a new type constructor A with a parameter for each type variable α bound at the top-level in the definition of f. The constructor symbols get the type $\mathsf{A}_i : \forall\overline{\alpha}.\ \tau_1 \to \ldots \to \tau_k \to \mathsf{A}\ \overline{\alpha}$, where τ_i is the type of the ith free variable. As the context transformation does not descend into $\Lambda\alpha.\ e$ terms, the free type variables of τ_i are contained in $\overline{\alpha}$. In the `map` example, this yields the accumulator:

```
type accum<a, b>
  Hole
  A1(x : b, k : accum<a, b>)
```

We now show that the transformed program is typeable. For convenience, we redefine $\hat{f}$ to take the continuation k as the first parameter.

Lemma 6.
Let Γ be a context that contains a type τ, a list of types $\tau_1 \ldots \tau_n$, a list of type variables $\overline{\alpha}$ and variables $f : \forall\overline{\alpha}.\ \tau_1 \to \ldots \to \tau_n \to \tau, \hat{f} : \forall\overline{\alpha}.\ \mathsf{A}\ \overline{\alpha} \to \tau_1 \to \ldots \to \tau_n \to \tau, k : \mathsf{A}\ \overline{\alpha}$ and $\mathsf{app} : \forall\overline{\alpha}.\ \mathsf{A}\ \overline{\alpha} \to \tau \to \tau$.

Let e be an expression with $\Gamma \vdash e : \tau$. Then $\Gamma \vdash [\![\ e\ ]\!]_{f,k} : \tau$.

Proof by induction on e.

Case $\Gamma \vdash (\mathsf{let}\ x = e' : \sigma\ \mathsf{in}\ e) : \tau$.

$$
\begin{array}{ll}
\Gamma,\ x : \sigma \vdash e : \tau & \{\ [let]\ \} \\
\Gamma,\ x : \sigma \vdash [\![\ e\ ]\!]_{f,k} : \tau & \{\ \textit{inductive hypothesis}\ \} \\
\Gamma \vdash \mathsf{let}\ x = e'\ \mathsf{in}\ [\![\ e\ ]\!]_{f,k} : \tau & \{\ [let]\ \} \\
\Gamma \vdash [\![\ \mathsf{let}\ x = e'\ \mathsf{in}\ e\ ]\!]_{f,k} : \tau & \{\ (tlet)\ \}
\end{array}
$$

Case $\Gamma \vdash \mathsf{match}\ (e' : \sigma)\ \{\ \overline{p_i \mapsto e_i}\ \} : \tau$.

$$
\begin{array}{ll}
\Gamma \vdash_{\mathrm{PAT}} p_i : \sigma \mapsto e_i : \tau & \{\ [match]\ \} \\
\Gamma,\ x_1 : \sigma_1, \ldots, x_k : s_k \vdash e_i : \tau & \{\ [pat]\ \} \\
\Gamma,\ x_1 : \sigma_1, \ldots, x_k : s_k \vdash [\![\ e_i\ ]\!]_{f,k} : \tau & \{\ \textit{inductive hypothesis}\ \} \\
\Gamma \vdash_{\mathrm{PAT}} p_i : \sigma \mapsto [\![\ e_i\ ]\!]_{f,k} : \tau & \{\ [pat]\ \} \\
\Gamma \vdash \mathsf{match}\ (e' : \sigma)\ \{\ \overline{p_i \mapsto [\![\ e_i\ ]\!]_{f,k}}\ \} : \tau & \{\ [match]\ \} \\
\Gamma \vdash [\![\ \mathsf{match}\ (e' : \sigma)\ \{\ \overline{p_i \mapsto e_i}\ \}\ ]\!]_{f,k} : \tau & \{\ (tmatch)\ \}
\end{array}
$$

Case $\mathsf{E}[f\ \overline{b}\ e_1 \ldots e_n]$, with $f\ \overline{b}\ e_1 \ldots e_n : \tau$. We write $\overline{b}$ for type variables that are potentially distinct from $\overline{\alpha}$.

$$
\begin{array}{ll}
\Gamma \vdash f\ \overline{b}\ e_1 \ldots e_n : \tau & \{\ \textit{by assumption}\ \} \\
\Gamma \vdash \hat{f}\ \overline{b}\ k\ e_1 \ldots e_n : \tau & \{\ \textit{by assumption}\ \} \\
\Gamma \vdash k : \mathsf{A}\ \overline{b} & \{\ \textit{by } [app]\ \} \\
\Gamma \vdash \mathsf{A}_i\ x_1 \ldots x_m\ k : \mathsf{A}\ \overline{b} & \{\ \textit{for } x_1 \ldots x_m \textit{ the free variables in } \mathsf{E}\ \} \\
\Gamma \vdash \hat{f}\ \overline{b}\ (\mathsf{A}_i\ x_1 \ldots x_m\ k)\ e_1 \ldots e_n : \tau & \{\ \textit{follows}\ \} \\
\Gamma \vdash [\![\ \mathsf{E}[f\ \overline{b}\ e_1 \ldots e_n]\ ]\!]_{f,k} : \tau & \{\ (tail)\ \}
\end{array}
$$

Case else.

$$
\begin{array}{ll}
\Gamma \vdash \mathsf{app}\ \overline{\alpha} : \mathsf{A}\ \overline{\alpha} \to \tau \to \tau & \{\ \textit{by assumption}\ \} \\
\Gamma \vdash k : \mathsf{A}\ \overline{\alpha} & \{\ \textit{by assumption}\ \} \\
\Gamma \vdash e : \tau & \{\ \textit{by assumption}\ \} \\
\Gamma \vdash \mathsf{app}\ \overline{\alpha}\ k\ e : \tau & \{\ [app]\ \} \\
\Gamma \vdash [\![\ e\ ]\!]_{f,k} : \tau & \{\ (base)\ \}
\end{array}
$$

Lemma 7.
Let Γ be a context that contains the variable $\mathsf{app} : \forall \overline{\alpha}.\ \mathsf{A}\ \overline{\alpha} \to \tau \to \tau$.
If $\Gamma \vdash \mathsf{fun}\ f = \Lambda\overline{\alpha}.\ \lambda xs.\ e : \forall \overline{\alpha}.\ \tau_1 \to \ldots \to \tau_n \to \tau$
then $\Gamma, f \vdash \mathsf{fun}\ \hat{f} = \Lambda\overline{\alpha}.\ \lambda k.\ \lambda xs.\ [\![\ e\ ]\!]_{f,k} : \forall \overline{\alpha}.\ \mathsf{A}\ \overline{\alpha} \to \tau_1 \to \ldots \to \tau_n \to \tau$.

Directly by lemma 6, LAM and FUNDECL.

Lemma 8.
Let $\Gamma \vdash \mathsf{fun}\ f = \Lambda\overline{\alpha}.\ \lambda x.\ e : \forall \overline{\alpha}.\ \tau$. Then $\Gamma \vdash \mathsf{fun}\ \mathsf{app} = \Lambda\overline{\alpha}.\ \lambda k.\ \lambda x.\ e' : \forall \overline{\alpha}.\ \mathsf{A}\ \overline{\alpha} \to \tau \to \tau$ with e' as defined above.

We prove $\mathsf{app}\ k\ x : \tau$ for the individual cases of k. The lemma follows by the MATCH, LAM and FUNDECL rules.

Case $k = \textit{Hole}$. Obvious as $x : \tau$.

Case $k = \mathsf{A}_i\ x_1 \ldots x_m\ k'$. We need to show that $[\![\ \mathsf{E}_i[\ x \mid x_1 \ldots x_m\ ]\ ]\!]_{f,k} : \tau$. By lemma 6, it suffices to show $\mathsf{E}_i[\ x \mid x_1 \ldots x_m\ ] : \tau$. However, this follows from $x : \tau$ and the requirement in (tail) that $\mathsf{E}[f\ e] : \tau$.

E POLYMORPHIC RECURSION

In this paper we have limited ourselves to recursive functions where each recursive call has the same return type. However, there are some functions where the recursive call might have a different return type due to polymorphic recursion. For example, Okasaki (1999) presents the following random access list:

```
type seq<a>
  Nil
  Zero(s : seq<(a, a)>)
  One(x : a, s : seq<(a, a)>)

fun cons(x : a, s : seq<a>) : seq<a>
  match s
    Nil -> One(x, Nil)
    Zero(ps) -> One(x, ps)
    One(y, ps) -> Zero(cons((x, y), ps))
```

Here the recursive call instantiates `a` with `(a,a)`, and the hole in `Zero(□)` has type `seq<(a,a)>`. As a consequence, performing the translation for polymorphically recursive code can lead to code that is not typeable is System F (see discussion in section D), but as with defunctionalization, one can regain typability with GADTs (Pottier and Gauthier, 2004). Even though Koka has an intermediate core representation based on System F, the application and composition functions are primitives and can be typed without needing extensions (and Koka transforms the above function without problems).

F AN EXAMPLE OF THE GENERATED CODE

As an example of the code generation of our TRMC scheme we consider the `map` function from our benchmarks where the `map` function is specialized by the compiler for the increment function as:

```
fun map_trmc'( xs : list<int32>, k : ctx<int32> ) : list<int32>
  match xs
    Nil -> app k Nil
    Cons x xx -> val y = x+1 in map_trmc'(xx, comp(k,y))

fun map_trmc( xs : list<int32> ) : list<int32>
  map_trmc'( xs, Ctx invalid null)
```

Here the `Ctx` constructor is the Minamide tuple as a value type containing the final result and hole address. For efficiency we represent the empty tuple with a `null` address for the hole. Eventually, such value type is passed in registers, and the generated code for arm64 becomes:

```
map_trmc':
  ...                           ; setup
  mov x21, x2                   ; x21 is the hole address of the tuple
  mov x19, x1                   ; x19 the final result part of the tuple
  cmp x0, #5                    ; is it Nil?
  b.ne  LBB3_5                  ; if not, goto to Cons branch
  ...
LBB3_5:                         ; Cons branch
  mov x20, x3                   ; set up loop variables in registers
  mov x23, #x100000000          ; used for fast int32 arithmetic
  mov w24, #x020202             ; Cons header: total fields=2, context path index=2, tag=2, rc=0
  mov w25, #1
LBB3_6:                         ; tail call entry
  ldp x26, x22, [x0, #8]        ; load pair: x = x26 and xx = x22
  ldr w8, [x0, #4]              ; load ref count in w8
  cbnz  w8, LBB3_10             ; if not unique, goto slower copying path
LBB3_7:
  add x8, x23, x26, lsl #31     ; increment x from/to a boxed int32 representation
  asr x8, x8, #31
  orr x8, x8, #0x1
  stp x24, x8, [x0]             ; store pair in-place: the header and the incremented x
  mov x8, x0
  str x25, [x8, #16]!           ; set the tail to invalid (1) for now (not really needed)
  cbz x21, LBB3_16              ; if this an empty tuple (hole==NULL), goto slow path
  str x0, [x21]                 ; else store our Cons result into the current hole
LBB3_9:
  mov x0, x22                   ; continue with the tail (x22)
  mov x21, x8                   ; and set x21 to the new hole
  cmp x22, #5                   ; is it a Nil?
  b.ne  LBB3_6                  ; if not, make a tail call
  b LBB3_2                      ; otherwise return
  ...

map_trmc:
  mov x3, x1                    ; set up the empty Minamide tuple
  mov w1, #1                    ; final result is invalid for now (1)
  mov x2, #0                    ; with the initial hole==NULL
  b map_trmc'                   ; and jump
```