JFP, 63 pages, 2025.
doi:10.1017/xxxxx

Tail Recursion Modulo Context: An Equational Approach (extended version)

DAAN LEIJEN
Microsoft Research, USA e-mail: daan@microsoft.com

ANTON LORENZEN
University of Edinburgh, UK e-mail: anton.lorenzen@ed.ac.uk

Abstract

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 further instantiations not so commonly associated with TRMC, such as *defunctionalized* evaluation contexts, *monoids*, *semirings*, *exponents* and *fields*. 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, Clément, and Scherer [2021] in the context of OCaml. A prototypical example of a function that can be transformed this way is `map`, which applies a function to every element of a list:

```
fun map( xs : list<a>, f : a -> b ) : 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 [Bour et al. 2021; Friedman and Wise 1975; Risch 1973], 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 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 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 instantiations not so commonly associated with TRMC that arise naturally in our generic approach, namely modulo *defunctionalized* evaluation contexts, modulo *monoids*, modulo *semirings*, and modulo *exponents*.
- In Section 6 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 exactly the implementation as used in our 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` [Danvy and Filinski 1990; Shan 2007; Sitaram and Felleisen 1990], or general algebraic effect handlers [Plotkin and Power 2003; Plotkin and Pretnar 2009], where one can resume more than once. 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 7 we show two novel solutions to this: The general approach generates two versions for each TRMc translation and chooses at runtime the appropriate version depending on whether 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 9).

This paper is the extended version of Leijen and Lorenzen [2023]. We make the following contributions over the conference version:

- We extend the TRMC algorithm to ensure that (when instantiated to general evaluation contexts) it can optimize all recursive calls that are not under a lambda (Section 3). In contrast, the algorithm presented in the conference paper could only achieve this if the source program was in *A-normal form* [Flanagan et al. 1993]. Our new algorithm extends the previous algorithm to perform the necessary A-normalizations on-demand.
- We describe a method for composing context instantiations (Section 5). This is especially useful for programs where fast instantiations (like semiring contexts) are not quite good enough to make the program tail-recursive. In that case, we can use the fast instantiation where it applies and use a slower instantiation like defunctionalized contexts for the rest. We use this insight to derive a tail-recursive evaluator for an arithmetic expression evaluator on fields.
- We include a detailed description of Koka's implementation of constructor contexts. We discuss a snippet of the assembly code generated by the Koka and explain the optimizations that make the implementation efficient (Section 7.2). Furthermore, we describe in detail another implementation strategy proposed by Lorenzen et al. [2024], which does not rely on reference counting and contrast it with our implementation.
- Constructor contexts were special in the conference version of this paper, since they were the only contexts for which the transformation could not be done manually. However, Lorenzen et al. [2024] realized that the hybrid approach can be used to make constructor contexts first-class values. This gives programmers the ability to make their functions tail-recursive manually. We include several practical examples of programming with first-class constructor contexts, where it would be hard to achieve a tail-recursive version fully automatically, but a manual solution is evident.

The new content in this version supersedes several sections of the conference paper. We no longer include "Improving Constructor Contexts" (which is now covered by the extended algorithm in Section 3), "Modulo Cons Products" (which can be achieved more easily using first-class constructor contexts), and "Fall Back to General Evaluation Contexts" (which is less efficient than the implementation proposed by Lorenzen et al. [2024]).

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:

```
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 we have to evaluate `f(x)` before allocating the closure `fn(ys) Cons(y,ys)` since `f` may have an observable (side) effect. The function `id` is the identity function, and `apply` and `compose` regular function application and composition:

```
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
```

All our 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). Note that every function arrow in Koka has three arguments where the type `a -> e b` denotes a function from type `a` to `b` with potential (side) effects `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`.

We would like to stress though that the described techniques are not restricted to Koka as such, and apply generally to any strict programming language (and particular instances can already be found in various compilers, including GCC, see Section 4.6). Some techniques, like the hybrid approach in Section 7.1 may require particular runtime support (like precise reference counts) but this is again independent of the particular language.

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 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 6), 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.

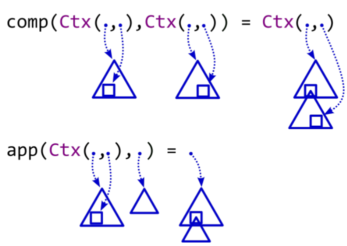
2.2 Linear Continuation Style

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 that we end up with a 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 a 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 App. 7.2 in the supplement). Section 9 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 6), its extension to non-linear control (Section 7), and finally conclude with benchmarks (Section 9) and related work. Proofs and further benchmarks can be found in the supplementary technical report [Leijen and Lorenzen 2022].

3 Calculating Tail-Recursion-Modulo-Context

In order to reason precisely 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 $\square$. We write $\mathsf{E}[v]$ for the substitution $\mathsf{E}[\square := v]$ (which binds tighter than function application). 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 $e_1 \longrightarrow e_2$ together with the (*step*) rule to reduce in any evaluation context $\mathsf{E}[e_1] \longmapsto \mathsf{E}[e_2]$ (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 $e_1 \;\cong\; e_2$ if e_1 and e_2 are *extensionally equivalent*: either $e_1 \longmapsto^* v$ and $e_2 \longmapsto^* v$, or both $e_1\Uparrow$ and $e_2\Uparrow$. During reasoning, we often use the rule that when e_2 is terminating, then $(\lambda x.\, e_1)\, e_2 \;\cong\; e_1[x{:=}e_2]$.

3.1 Abstract Contexts

Before we start calculating our general TRMC transformation, we first define *abstract contexts* as an abstract type $ctx\,\tau$ in our calculus. There are three context operations: creation (as ctx), application (as app), and composition (as $(\bullet)$). 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:

Expressions:

$$
\begin{array}{lllllll}
e & ::= & v & \text{(value)} & v & ::= & x, y & \text{(variables)} \\
 & | & e\,e & \text{(application)} & & | & f & \text{(recursive functions)} \\
 & | & \mathsf{let}\, x = e\, \mathsf{in}\, e & \text{(let binding)} & & | & \lambda x.\, e & \text{(functions)} \\
 & | & \mathsf{match}\, e\, \{\, \overline{p_i \mapsto e_i}\,\} & \text{(matching, } i \geqslant 1) & & | & C^k\, v_1 \ldots v_n & \text{(constructor of arity } k, \\
 & & & & & & & \;\text{with } k \geqslant 0 \text{ and } n \leqslant k) \\
p & ::= & C^k\, x_1 \ldots x_k & \text{(pattern)} & F & ::= & \{\overline{f_i = \lambda x.\, e_i}\,\} & \text{(recursive definitions)}
\end{array}
$$

Syntax:

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

Evaluation Contexts:

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

Tail Contexts:

$$
\mathsf{T} \;::=\; \Box \mid e\, \mathsf{T} \mid \mathsf{let}\, x = e\, \mathsf{in}\, \mathsf{T} \mid \mathsf{match}\, e\, \{\, \overline{p_i \mapsto \mathsf{T}_i}\,\} \quad \text{(tail context)}
$$

Expression Contexts (= Tail Context + Evaluation Context):

$$
\mathsf{X} \;::=\; \Box \mid \mathsf{X}\, e \mid e\, \mathsf{X} \mid \mathsf{let}\, x = \mathsf{X}\, \mathsf{in}\, e \mid \mathsf{let}\, x = e\, \mathsf{in}\, \mathsf{X} \mid \mathsf{match}\, \mathsf{X}\, \{\, \overline{p_i \mapsto e_i}\,\} \mid \mathsf{match}\, e\, \{\, \overline{p_i \mapsto \mathsf{X}_i}\,\}
$$

Operational Semantics:

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

$$
\frac{e_1 \longrightarrow e_2}{\mathsf{E}[e_1] \longmapsto \mathsf{E}[e_2]}\;[\textsc{step}]
$$

Fig. 1. Syntax and operational semantics.

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

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 context, we need to show the context laws are satisfied. In such case, 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 (*appcomp*) it follows directly that $(app\ (k_1 \bullet k_2)\ e)\Uparrow$ 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. The algorithm is calculated assuming an abstract context where the context laws hold. Eventually, the algorithm needs to be instantiated in the compiler to particular contexts (like constructor contexts), with a particular implementation of context application and composition. We show many such instantiations in Sections 4 and 6.

For clarity, we use single parameter functions for proofs and derivations (but of course the results extend straightforwardly to multiple parameter functions). Now consider a function $f\,x = e_f$ with its TRMC transformed version denoted as f':

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

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: $\mathsf{exec}\,k\,(\mathit{eval}\,e) \cong \mathit{eval}'\,e\,k$. Following that approach, we use $\mathsf{app}\,k\,(f\,e) \cong f'\,e\,k$ **(a)** as our initial specification. This seems a good start since it implies:

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

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

Unfortunately, the specification is not yet specific enough to calculate with 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):

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

(and if $e\Uparrow$, then $\mathsf{app}\,k\,(f\,e)\Uparrow$ and $f'\,e\,k\Uparrow$ 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 improved specification directly gives us a trivial solution for the translation as:

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

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 $e = \mathsf{E}[f\; e_1]$. We can then calculate:

$$\begin{array}{lll}
 & \mathsf{app}\; k\; \mathsf{E}[f\; e_1] & \\
= & \mathsf{app}\; k\; (\mathsf{app}\; (\mathsf{ctx}\; \mathsf{E})\; (f\; e_1)) & \{\; (appctx)\; \} \\
= & \mathsf{app}\; (k \bullet \mathsf{ctx}\; \mathsf{E})\; (f\; e_1) & \{\; (appcomp)\; \} \\
\cong & f'\; e_1\; (k \bullet \mathsf{ctx}\; \mathsf{E}) & \{\; specification\; (a)\; \} \\
= & [\![\mathsf{E}[f\; e_1]]\!]_{f,k} & \{\; define\; \}
\end{array}$$

which leads to the following set of equations:

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

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 constrained by some particular $(\star)$ condition (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 instantiation 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 4.2).

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 $\mathsf{let}\; x = e_0\; \mathsf{in}\; e_1$ where $e_0 \neq \mathsf{E}[f\; e']$ (so it does not overlap with E contexts):

$$\begin{array}{lll}
 & \mathsf{app}\; k\; (\mathsf{let}\; x = e_0\; \mathsf{in}\; e_1) & \\
\cong & \mathsf{app}\; k\; e_1\, [x{:=}e_0] & \{\; (let),\; e_0\; is\; terminating\; \} \\
= & (\mathsf{app}\; k\; e_1)\, [x{:=}e_0] & \{\; x \notin \mathsf{fv}(k)\; \} \\
\cong & \mathsf{let}\; x = e_0\; \mathsf{in}\; \mathsf{app}\; k\; e_1 & \{\; e_0\; is\; terminating\; \} \\
\cong & \mathsf{let}\; x = e_0\; \mathsf{in}\; [\![e_1]\!]_{f,k} & \{\; specification\; \} \\
= & [\![\mathsf{let}\; x = e_0\; \mathsf{in}\; e_1]\!]_{f,k} & \{\; define\; \}
\end{array}$$

(and if $e_0\Uparrow$, then also $\mathsf{app}\; k\; (\mathsf{let}\; x = e_0\; \mathsf{in}\; e_1)\Uparrow$ and $[\![\mathsf{let}\; x = e_0\; \mathsf{in}\; e_1]\!]_{f,k}\Uparrow$).

Unfortunately, this rule is still too restrictive in general as it does not apply when the let-statement is itself under a context E. For example, we might encounter an expression like:

$$\mathsf{let}\; x = (\mathsf{let}\; y = e_0\; \mathsf{in}\; f\; x\; y)\; \mathsf{in}\; e_1$$

Here, the recursive call is under the let-binding of x (as $\mathsf{E}[\mathsf{let}\; y = e_0\; \mathsf{in}\; f\; x\; y]$) but the $y = e_0$ binding prevents the recursive call $f\; x\; y$ to be the focus of the evaluation context. This situation occurs whenever an expression is not in *A-normal form* [Flanagan et al. 1993], and these cannot be optimized by the rules outlined so far (and neither by the rules as presented in earlier work [Leijen and Lorenzen 2023]). Instead, we need to consider the general case where the let-binding appears under a context E. Assuming that the variables bound in let-bindings and matches are fresh, we can calculate:

$$
\begin{array}{lll}
 & \mathsf{app}\,k\,\mathsf{E}[\mathsf{let}\,x = e_0\,\mathsf{in}\,e_1] & \\
\cong & \mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,(\mathsf{let}\,x = e_0\,\mathsf{in}\,e_1)) & \{\ (\mathit{appctx})\ \} \\
\cong & \mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,e_1[x{:=}e_0]) & \{\ (\mathit{let}),\ e_0\ \mathit{is\ terminating}\ \} \\
= & \mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,e_1)[x{:=}e_0] & \{\ x \notin \mathsf{fv}(\mathsf{E})\ \} \\
= & (\mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,e_1))[x{:=}e_0] & \{\ x \notin \mathsf{fv}(k)\ \} \\
\cong & \mathsf{let}\,x = e_0\,\mathsf{in}\,\mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,e_1) & \{\ e_0\ \mathit{is\ terminating}\ \} \\
\cong & \mathsf{let}\,x = e_0\,\mathsf{in}\,\mathsf{app}\,k\,\mathsf{E}[e_1] & \{\ (\mathit{appctx})\ \} \\
\cong & \mathsf{let}\,x = e_0\,\mathsf{in}\,[\![\ \mathsf{E}[e_1]\ ]\!]_{f,k} & \{\ (\mathit{specification})\ \} \\
= & [\![\ \mathsf{E}[\mathsf{let}\,x = e_0\,\mathsf{in}\,e_1]\ ]\!]_{f,k} & \{\ \mathit{define}\ \}
\end{array}
$$

(and if $e_0\Uparrow$, then also $\mathsf{app}\,k\,\mathsf{E}[\mathsf{let}\,x = e_0\,\mathsf{in}\,e_1]\Uparrow$ and $[\![\ \mathsf{E}[\mathsf{let}\,x = e_0\,\mathsf{in}\,e_1]\ ]\!]\Uparrow$). Effectively we have lifted out the let-binding from the evaluation context. We can do the same for matches:

$$
\begin{array}{lll}
 & \mathsf{app}\,k\,\mathsf{E}[\mathsf{match}\,e_0\,\{\,\overline{p_i \to e_i}\,\}] & \\
\cong & \mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,(\mathsf{match}\,e_0\,\{\,\overline{p_i \to e_i}\,\})) & \{\ (\mathit{appctx})\ \} \\
\cong & \mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,e_i[x_1{:=}v_1,\ldots,x_n{:=}v_n]) & \{\ p_i = C_i\,x_1\ldots x_n,\ e_0 \cong C_i\,v_1\ldots v_n,\ \mathbf{1}\ \} \\
= & \mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,e_i)[x_1{:=}v_1,\ldots,x_n{:=}v_n] & \{\ x_j \notin \mathsf{fv}(\mathsf{E})\ \} \\
= & (\mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,e_i))[x_1{:=}v_1,\ldots,x_n{:=}v_n] & \{\ x_j \notin \mathsf{fv}(k)\ \} \\
\cong & \mathsf{match}\,e_0\,\{\ \overline{p_i \to \mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,e_i)}\ \} & \{\ (1),\ e_0\ \mathit{is\ terminating}\ \} \\
\cong & \mathsf{match}\,e_0\,\{\ \overline{p_i \to \mathsf{app}\,k\,\mathsf{E}[e_i]}\ \} & \{\ (\mathit{appctx})\ \} \\
\cong & \mathsf{match}\,e_0\,\{\ \overline{p_i \to [\![\ \mathsf{E}[e_i]\ ]\!]_{f,k}}\ \} & \{\ (\mathit{specification})\ \} \\
= & [\![\ \mathsf{E}[\mathsf{match}\,e_0\,\{\,\overline{p_i \to e_i}\,\}]\ ]\!]_{f,k} & \{\ \mathit{define}\ \}
\end{array}
$$

(and if $e_0\Uparrow$, then also $\mathsf{app}\,k\,\mathsf{E}[\mathsf{match}\,e_0\,\{\ldots\}]\Uparrow$ and $[\![\ \mathsf{E}[\mathsf{match}\,e_0\,\{\ldots\}]\ ]\!]\Uparrow$).

This form of specification essentially performs A-normalization whenever necessary to create further opportunities to match on tail-recursive calls. Our presentation follows the approach of Maurer et al. [2017], who describe the positions in a term that occur last in an evaluation order as *tail contexts* T. They show that A-normalization can be achieved by commuting the E and T contexts whenever possible. This is exactly the approach taken here, where we commute single let-bindings and matches under E contexts to the front of the term. A potential drawback of the match normalization is that it duplicates the evaluation context E in each of the branches. Maurer et al. [2017] also show how *join points* can be used to avoid code duplication in such case.

This leaves one last expression form to consider: the application of a function to an argument. Using the intuition of commuting tail contexts, we might define $[\![\ \mathsf{E}[e_0\,e_1]\ ]\!]_{f,k} = e_0\,[\![\ \mathsf{E}[e_1]\ ]\!]_{f,k}$. However, while e_0 can now be evaluated early, the application itself depends on the result of our transformation. Thus, we need to be a bit more careful and instead calculate:

$$
\begin{array}{lll}
 & \mathsf{app}\,k\,\mathsf{E}[e_0\,e_1] & \\
\cong & \mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,(e_0\,e_1)) & \{\ (\mathit{appctx})\ \} \\
\cong & \mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,(\mathsf{let}\,g = e_0\,\mathsf{in}\,g\,e_1)) & \{\ (\mathit{let}),\ \mathit{for\ fresh}\ g\ \} \\
\cong & \mathsf{let}\,g = e_0\,\mathsf{in}\,\mathsf{app}\,k\,(\mathsf{app}\,(\mathsf{ctx}\,\mathsf{E})\,(g\,e_1)) & \{\ \overline{\alpha}\ \mathit{before}\ \} \\
\cong & \mathsf{let}\,g = e_0\,\mathsf{in}\,\mathsf{app}\,k\,\mathsf{E}[g\,e_1] & \{\ (\mathit{appctx})\ \} \\
\cong & \mathsf{let}\,g = e_0\,\mathsf{in}\,[\![\ \mathsf{E}[g\,e_1]\ ]\!]_{f,k} & \{\ (\mathit{specification})\ \} \\
= & [\![\ \mathsf{E}[e_0\,e_1]\ ]\!]_{f,k} & \{\ \mathit{define}\ \}
\end{array}
$$

$$
\begin{array}{llll}
(\textit{tlet}) & [\![\ \mathsf{E}[\mathsf{let}\, x = e_0 \,\mathsf{in}\, e]\ ]\!]_{f,k} & = \mathsf{let}\, x = e_0 \,\mathsf{in}\, [\![\ \mathsf{E}[e]\ ]\!]_{f,k} & \\
(\textit{tmatch}) & [\![\ \mathsf{E}[\mathsf{match}\, e_0\, \{\, \overline{p_i \to e_i}\, \}]\ ]\!]_{f,k} & = \mathsf{match}\, e_0\, \{\, \overline{p_i \to [\![\ \mathsf{E}[e_i]\ ]\!]_{f,k}}\ \} & \\
(\textit{tapp}) & [\![\ \mathsf{E}[e_0\, e]\ ]\!]_{f,k} & = \mathsf{let}\, g = e_0 \,\mathsf{in}\, [\![\ \mathsf{E}[g\, e]\ ]\!]_{f,k} & \text{with } g \text{ fresh} \\
(\textit{tail}) & [\![\ \mathsf{E}[f\, e_1 \ldots e_n]\ ]\!]_{f,k} & = f'\, e_1 \ldots e_n\, (k \bullet (\mathsf{ctx}\, \mathsf{E})) & \text{iff } (\star) \\
(\textit{base}) & [\![e]\!]_{f,k} & = \mathsf{app}\, k\, e & \text{otherwise}
\end{array}
$$

where $e_0,\ e_1,\ \ldots,\ e_n \neq \mathsf{X}[f\, e'_1 \ldots e'_n]$

Fig. 2. Calculated algorithm for general selective tail recursion modulo context transformation. It is parameterized by the $(\star)$ condition, and the composition $(\bullet)$ and application (app) operations

(and if $e_0 \Uparrow$, then also $\mathsf{app}\, k\, \mathsf{E}[e_0\, e_1] \Uparrow$ and $[\![\ \mathsf{E}[e_0\, e_1]\ ]\!] \Uparrow$).

In this general form, we need to strengthen our requirement that $e_0 \neq \mathsf{E}[f\, e']$ to ensure that it does not overlap with our newly calculated rules. We write $e_0 \neq \mathsf{X}[f\, e_1 \ldots e_n]$ to mean that e_0 can not have a recursive call under an *expression context* X. The expression context $\mathsf{X}[e]$ matches all possible expressions that contain e, unless e occurs exclusively under lambdas in $\mathsf{X}[e]$.

3.3 The Tail-Recursion-Modulo-Contexts Algorithm

Figure 2 shows all five 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 $ctx\, \alpha$, 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.

Thanks to the changes in this extended version, we can now prove that the TRMC algorithm exhaustively optimizes recursive calls:

Theorem 1. (*Matching all recursive calls*)
For any transformed expression $e' = [\![e]\!]_{f,k}$ with $(\star)$ unconstrained, we have $e' \neq \mathsf{X}[f\, e_1 \ldots e_n]$.

There are two types of recursive calls that can not be optimized by our algorithm: First, we do not optimize recursive calls inside lambdas. This is a necessary restriction, since it is impossible in general to push the accumulated context k under a lambda. Second, our algorithm will only optimize the *first* recursive call(s) in the evaluation order. If those are followed by further recursive calls, the evaluation context E stored as $\mathsf{ctx}\, \mathsf{E}$ in the (*tail*) rule may still contain unoptimized recursive calls. We will revisit this problem in Section 4.2.

To see our algorithm in action, let's consider the *map* function again:

$$
\begin{array}{l}
map\, xs\, f \ = \ \mathsf{match}\, xs\, \{\ Nil \to Nil \\
\qquad\qquad\qquad\qquad\qquad\ \ Cons\, x\, xx \to Cons\, (f\, x)\, (map\, xx\, f)\ \}
\end{array}
$$

When translating this function, we first use the (*tmatch*) rule with $\mathsf{E} = \square$ to descend into the branches of the match. In the *Nil* branch, the (*base*) rule applies. In the *Cons* branch, we use the (*tapp*) rule (again with $\mathsf{E} = \square$) to bind the call to $f\, x$. We then use the (*tail*) rule to optimize the recursive call to `map xx f`:

$$\begin{aligned}
&map'\,xs\,f\,k \;=\; \mathsf{match}\,xs\,\{\; Nil \rightarrow \mathsf{app}\,k\,Nil\\
&\qquad\qquad\qquad\qquad Cons\,x\,xx \rightarrow \mathsf{let}\,c \;=\; Cons\;(f\;x)\;\mathsf{in}\,map'\,xx\,f\;(k \bullet (\mathsf{ctx}\;(c\,\square)))\;\}
\end{aligned}$$

However, for Constructor Contexts (Section 6), it is useful to keep the *Cons* constructor in the context passed to *map*. In our pratical implementation, we therefore modify the (*tapp*) rule slightly to extract the arguments instead of the entire partially applied function. Our final transformation for *map* is then:

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

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 $ctx\,\alpha \;=\; \alpha \rightarrow \alpha$:

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

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:

$$\begin{aligned}
&map\,xs\,f \;=\; \mathsf{match}\,xs\,\{\; Nil \rightarrow Nil\\
&\qquad\qquad\qquad Cons\,x\,xx \rightarrow \mathsf{let}\,y \;=\; f\,x\;\mathsf{in}\;Cons\,y\;(map\,xx\,f)\;\}
\end{aligned}$$

which translates to:

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

and which the compiler can further simplify into:

$$\begin{aligned} map'\, xs\, f\, k \;=\; \mathsf{match}\, xs \,\{\; & Nil \rightarrow k\, Nil \\ & Cons\, x\, xx \rightarrow \mathsf{let}\, y = f\, x\, \mathsf{in}\, map'\, xx\, f\, (\lambda x.\, k\, (Cons\, y\, x))\;\} \end{aligned}$$

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.

4.2 Nested Translation of Modulo Evaluation Contexts

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 $f'\, e\, (k \bullet \mathsf{ctx}\, \mathsf{E})$, 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 $f'\, e\, (\mathsf{ctx}\, \square)$ 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.

4.3 Modulo Defunctionalized Evaluation Contexts

In order to better understand the shapes that evaluation contexts can take, we want to consider the *defunctionalization* [Danvy and Nielsen 2001; Reynolds 1972] 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:

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

Just as we saw in Section 4.2, we need to use the translated evaluation context in the definition of app translate nested calls. The context laws now follow by induction – see App. B.1 in the supplement for the derivations. Applying this instantiation to the `map` function, we obtain:

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

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:

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

As we can see, using the modulo defunctionalized evaluation 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 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 1991]: 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]).

4.3.1 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. This can happen automatically in languages with reuse analysis such as Koka [Lorenzen and Leijen 2022], Lean [Ullrich and de Moura 2019], or OPAL [Didrich et al. 1994]. In particular, in the `app` function, the match:

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

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 6.1), 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 [Lorenzen et al. 2023; Reinking, Xie et al. 2021].

4.4 Modulo Associative Operator Contexts

In the previous instantiations we considered general evaluation 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 $\odot : \tau \rightarrow \tau \rightarrow \tau$ 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:

$$length\,xs = \mathsf{match}\,xs\,\{\,Cons\,x\,xx \rightarrow 1 + length\,xx;\ Nil \rightarrow 0\,\}$$

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:

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

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:

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

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

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

and for context application we have:

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

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:

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

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

$$k \bullet (\mathsf{ctx}\ (1\ +\ \square))\ =\ k\ +\ (\mathsf{ctx}\ (1\ +\ \square))\ =\ k\ +\ (\!|1 + \square|\!)\ =\ k\ +\ 1$$

such that we end up with:

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

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

4.4.1 Using Right Biased Contexts

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

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

with the fold defined as:

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

We can now compose in the opposite order:

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

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

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

First, our TRMC algorithm transforms it into:

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

and with our instantiated context, this simplifies to:

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

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

4.5 Modulo Monoid Contexts

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

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

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:

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

which we compose as:

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

We can again show that the context laws hold for this definition (see App. B.3 in the supplement).

4.6 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]:

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

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 separately (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:

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

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

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

We can show the context laws hold for these definitions:

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

and

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

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:

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

which further simplifies to:

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

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. Indeed, it turns out that GCC implements this optimization [Dvořák 2004] for integers and floating point numbers (if `-fast-math` is enabled to allow the assumption of associativity). This implementation specifically creates two local accumulators for addition and multiplication, and uses a direct while loop to compile the tail recursive calls.

4.7 Modulo Exponent Contexts

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

$$\mathsf{E}\ ::= \square\ |\ g\;\mathsf{E}$$

If we use a defunctionalized evaluation context from Section 4.3 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:

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

We can define the primitive operations as:

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

where $\mathsf{app}\ k\ e$ applies the function g to its argument k times. See App. B.4 in the supplement 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 $\mathsf{app}\ k\ (g\ e)$ by $[\![g\ e]\!]_{f,k}$ at compile time (as shown in Section 4.2). 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:

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

5 Context Composition

While the contexts we have defined so far are useful when they apply, they can fall short if they only match *some* of the recursive calls. This makes them fragile when a new recursive call is added to a function, as the context may no longer apply. In this section, we remove this restriction by showing how fast but restricted contexts can be composed with slower more general ones. We have not implemented this feature in Koka, but we hope to do so in the future.

5.1 A Basic Expression Evaluator

To motivate the composition of contexts, we consider a basic arithmetic expression evaluator in the style of Hutton [2021]:

```
type expr
  Lit(lit : int)
  Add(e1 : expr, e2 : expr)

fun eval(e)
  match e
    Add(e1, e2) -> eval(e1) + eval(e2)
    Lit(n)      -> n
```

The `+` suggests the use of a monoid context. However, this does not apply directly, since we have two recursive calls to `eval` instead of just one. The best we can do is to ignore the first recursive call and treat it as a regular value. Then we would obtain:

```
fun eval(e, addacc)
  match e
    Add(e1, e2) -> eval(e2, eval(e1, addacc))
    Lit(n)      -> addacc + n
```

However, we have not quite achieved a tail-recursive version yet. Like Hutton [2021], we can achieve this by using defunctionalized evaluation contexts:

```
type accum
  Hole
  AddL(k : accum, e : expr)
  AddR(addacc : int, k : accum)

fun eval(e, acc)
  match e
    Add(e1, e2) -> eval(e1, AddL(acc, e2))
    Lit(n)      -> app(acc, n)

fun app(acc, result)
  match acc
    Hole -> result
    AddL(k, e) -> eval(e, AddR(result, k))
    AddR(addacc, k) -> app(k, addacc + result)
```

This version is now tail-recursive, but it is also more complex than the original version and involves the allocation of `AddL` and `AddR` constructors. In particular, the `AddR` constructor seems superfluous, as it corresponds to the context `app(k,addacc + eval(e2))`, which we optimized using the monoid contexts earlier. In this section, we want to combine the two approaches to obtain a more efficient version, where we use *both* an accumulator and a monoid context:

```
type accum
  Hole
  AddL(k : accum, e : expr)

fun eval(e, acc, addacc)
  match e
    Add(e1, e2) -> eval(e1, AddL(acc, e2), addacc)
    Lit(n)      -> app(acc, addacc + n)

fun app(acc, result)
  match acc
    Hole -> result
    AddL(k, e) -> eval(e, k, result)
```

This version has the best of both worlds: it is fully tail-recursive and only needs to allocate a defunctionalized continuation for the left recursive call (where we need to keep track of the expression `e2`), while the right recursive call is efficiently handled by the monoid context.

5.2 Swapping Contexts

To achieve this transformation in general, we need to be able to compose two contexts. For two contexts E_1 and E_2, we can define their product context, which consists of tuples of the two contexts. We can apply a product context to an expression by applying each context in turn:

$$(\mathit{papp}) \quad \mathsf{app}\,(l,\ r)\,e \;=\; \mathsf{app}\,l\,(\mathsf{app}\,r\,e)$$

But how would we compose two product contexts? We would like to turn a composition of tuples into a tuple of compositions as $(l_1,\ r_1) \bullet (l_2,\ r_2) \;=\; (l_1 \bullet l_2,\ r_1 \bullet r_2)$. We can try to calculate this directly:

$$\begin{array}{lll}
 & \mathsf{app}\,((l_1,\ r_1) \bullet (l_2,\ r_2))\,e & \\
= & \mathsf{app}\,(l_1,\ r_1)\,(\mathsf{app}\,(l_2,\ r_2)\,e) & \{\ (\mathit{appcomp})\ \} \\
= & \mathsf{app}\,l_1\,(\mathsf{app}\,r_1\,(\mathsf{app}\,l_2\,(\mathsf{app}\,r_2\,e))) & \{\ (\mathit{papp})\ \}
\end{array}$$

. . . but now we are stuck. Here, l_1, l_2 belong to the context E_1 and r_1, r_2 to E_2. In order to make progress, we have to *swap* the inner contexts r_1 and l_2. But this is not always going to be possible! Instead, we need to parameterize the product context with a swap operation:

$$(\mathit{appswap}) \quad \mathsf{app}\,l\,(\mathsf{app}\,r\,e) \;=\; \mathsf{app}\,r'\,(\mathsf{app}\,l'\,e) \quad \mathsf{where}\ (l',\ r') \;=\; \mathsf{swap}(l,\ r)$$

If the contexts are connected in this sense, we can continue to calculate their composition:

$$\begin{array}{lll}
 & \mathsf{app}\,l_1\,(\mathsf{app}\,r_1\,(\mathsf{app}\,l_2\,(\mathsf{app}\,r_2\,e))) & \\
= & \mathsf{app}\,l_1\,(\mathsf{app}\,l_2'\,(\mathsf{app}\,r_1'\,(\mathsf{app}\,r_2\,e))) & \{\ \mathit{swap}(r_1,\ l_2) \;=\; (l_2',\ r_1')\ \} \\
= & \mathsf{app}\,(l_1 \bullet l_2')\,(\mathsf{app}\,(r_1' \bullet r_2)\,e) & \{\ (\mathit{appcomp})\ \} \\
= & \mathsf{app}\,(l_1 \bullet l_2',\ r_1' \bullet r_2)\,e & \{\ (\mathit{papp})\ \}
\end{array}$$

This gives us a definition for product contexts: we can fold any context $\mathsf{E} \;=\; \mathsf{E}_1 \mid \mathsf{E}_2$ by composing the folds of E_1 and E_2:

$$\begin{array}{ll}
\llparenthesis \square \rrparenthesis & = (\mathsf{ctx}\,\square,\ \mathsf{ctx}\,\square) \\
\llparenthesis \mathsf{E}_1\,[e] \rrparenthesis & = (\mathsf{ctx}\,\mathsf{E}_1,\ \mathsf{ctx}\,\square) \bullet \llparenthesis e \rrparenthesis \\
\llparenthesis \mathsf{E}_2\,[e] \rrparenthesis & = (\mathsf{ctx}\,\square,\ \mathsf{ctx}\,\mathsf{E}_2) \bullet \llparenthesis e \rrparenthesis
\end{array}$$

$$\begin{array}{lll}
(\mathit{pctx}) & \mathsf{ctx}\,\mathsf{E} & = (\llparenthesis \mathsf{E} \rrparenthesis) \\
(\mathit{pcomp}) & (l_1, r_1) \bullet (l_2, r_2) & = (l_1 \bullet l_2',\ r_1' \bullet r_2) \quad \mathsf{where}\ (l_2',\ r_1') \;=\; \mathsf{swap}(r_1,\ l_2) \\
(\mathit{papp}) & \mathsf{app}\,(k_1,\ k_2)\,e & = \mathsf{app}\,k_1\,(\mathsf{app}\,k_2\,e)
\end{array}$$

With this definition in hand, we can now derive several contexts from the previous section from the more basic contexts we defined earlier.

5.2.1 Modulo Monoid Contexts

We motivated the Modulo Monoid Contexts in Section 4.5 as the composition of a left-biased and a right-biased context. In fact, we can now *derive* this context as the product context of a left-biased and right-biased contexts, with $\mathit{swap}(l,\ r) \;=\; (r,\ l)$. This follows the swap law since:

$$\begin{array}{ll}
 & \mathsf{app}\,l\,(\mathsf{app}\,r\,e) \\
= & l \odot (e \odot r) \\
= & (l \odot e) \odot r \\
= & \mathsf{app}\,r\,(\mathsf{app}\,l\,e)
\end{array}$$

With this, we get exactly the previous definition of (acomp) of Modulo Monoid Contexts.

5.2.2 Modulo Semiring Contexts

Similarly, we can derive the semiring context (Section 4.6) as the composition of two left-biased contexts for its addition (l) and multiplication (r). Here, the swap operation is given by $swap(r,\ l) = (r * l,\ r)$:

$$
\begin{aligned}
& \mathsf{app}_*\, r\, (\mathsf{app}_+\, l\, e) \\
= \;& r * (l + e) \\
= \;& (r * l) + (r * e) \\
= \;& \mathsf{app}_+\, (r * l)\, (\mathsf{app}_*\, r\, e)
\end{aligned}
$$

With this, we get exactly the previous definition of (*scomp*) and it our new context corresponds to the semiring contexts we defined earlier. For this definition to work, it is important though that the left-biased context for the addition is in the first component of the tuple with multiplication in the second. That allows us to define a swap operation that uses the distributivity of the semiring to swap the contexts. We could not define a swap operation if multiplication is in the first component, since this would require us to move an addition under a multiplication, which is only possible if the semiring has multiplicative inverses.

5.3 Composing (Defunctionalized) Evaluation Contexts

(Defunctionalized) evaluation contexts are the only contexts introduced in the last section that can reliably make all recursive calls tail-recursive. For this reason they are particularly attractive for composition with other contexts, that lead to faster code in practice but only apply in more limited cases. Thankfully, this is easily possible, since we can swap an arbitrary context r with a general evaluation context l by storing it in a closure:

$$swap\,(r,\ l) = ((\lambda x.\, \mathsf{app}\, r\, x) \bullet l,\ \mathsf{ctx}\, \square)$$

We can verify that this definition fulfills the swap law:

$$
\begin{aligned}
& \mathsf{app}\, r\, (\mathsf{app}\, l\, e) \\
= \;& (\lambda x.\, \mathsf{app}\, r\, x)\, (\mathsf{app}\, l\, e) && \{\ \textit{eta expansion}\ \} \\
= \;& \mathsf{app}\, (\lambda x.\, \mathsf{app}\, r\, x)\, (\mathsf{app}\, l\, e) && \{\ (\textit{eapp})\ \} \\
= \;& \mathsf{app}\, ((\lambda x.\, \mathsf{app}\, r\, x) \bullet l)\, e && \{\ (\textit{appcomp})\ \} \\
= \;& \mathsf{app}\, ((\lambda x.\, \mathsf{app}\, r\, x) \bullet l)\, \square[e] \\
= \;& \mathsf{app}\, ((\lambda x.\, \mathsf{app}\, r\, x) \bullet l)\, (\mathsf{app}\, (\mathsf{ctx}\, \square)\, e) && \{\ (\textit{appctx})\ \}
\end{aligned}
$$

The same approach can also be used for defunctionalized evaluation contexts. Analogous to creating a fresh closure, we could create a special constructor to store an application of the other context. However, to avoid allocations and to enable a nested translation (Section 4.2), we integrate the restricted context into the constructors.

We define the extended 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 *and the inner context* k'. The compiler then generates an app function where we interpret A_i by evaluating E_i with the stored free variables:

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

Then we can define the swap operation as:

$$swap\,(k_1,\ \mathsf{H}) = (\mathsf{H},\ k_1)$$
$$swap\,(k_1,\ \mathsf{A}_i\,x_1\ldots x_m\,k'\,k_2) = (\mathsf{A}_i\,x_1\ldots x_m\,(k_1 \bullet k')\,k_2,\ \mathsf{ctx}\,\square)$$

This definition again fulfills the swap law. Case H:

$$\begin{array}{lll} & \mathsf{app}\,r\,(\mathsf{app}\,\mathsf{H}\,e) & \\ = & \mathsf{app}\,r\,e & \{\ (dapp)\ \} \\ = & \mathsf{app}\,\mathsf{H}\,(\mathsf{app}\,r\,e) & \{\ (dapp)\ \} \end{array}$$

Case $\mathsf{A}_i\,x_1\ldots x_m\,k'\,k_2$:

$$\begin{array}{lll} & \mathsf{app}\,r\,(\mathsf{app}\,(\mathsf{A}_i\,x_1\ldots x_m\,r'\,k_2)\,e) & \\ = & \mathsf{app}\,r\,([\![\mathsf{E}_i[e,\ x_1\ldots x_m]]\!]_{f\,(r',\,k_2)}) & \{\ (dapp)\ \} \\ = & \mathsf{app}\,r\,(\mathsf{app}\,(r',\ k_2)\,(\mathsf{E}_i[e,\ x_1\ldots x_m])) & \{\ specification\,(b)\ \} \\ = & \mathsf{app}\,r\,(\mathsf{app}\,r'\,(\mathsf{app}\,k_2\,(\mathsf{E}_i[e,\ x_1\ldots x_m]))) & \{\ (papp)\ \} \\ = & \mathsf{app}\,(r \bullet r')\,(\mathsf{app}\,k_2\,(\mathsf{E}_i[e,\ x_1\ldots x_m])) & \{\ (appcomp)\ \} \\ = & \mathsf{app}\,(r \bullet r',\ k_2)\,(\mathsf{E}_i[e,\ x_1\ldots x_m]) & \{\ (papp)\ \} \\ = & [\![\mathsf{E}_i[e,\ x_1\ldots x_m]]\!]_{f\,(k_2,\,r \bullet r')} & \{\ specification\,(b)\ \} \\ = & \mathsf{app}\,(\mathsf{A}_i\,x_1\ldots x_m\,(r \bullet r')\,k_2)\,e & \{\ (dapp)\ \} \\ = & \mathsf{app}\,(\mathsf{A}_i\,x_1\ldots x_m\,(r \bullet r')\,k_2)\,\square[e] & \\ = & \mathsf{app}\,(\mathsf{A}_i\,x_1\ldots x_m\,(r \bullet r')\,k_2)\,(\mathsf{app}\,(\mathsf{ctx}\,\square)\,e) & \{\ (appctx)\ \} \end{array}$$

5.4 Extending the Expression Evaluator

We can use this insight to derive a tail-recursive expression evaluator which supports multiplication as well, where we compose a defunctionalized evaluation context with a semiring context. First, we add a new constructor `Mul(e1,e2)` to our expression datatype which encodes the multiplication `eval(e1) * eval(e2)`. We then create a datatype `accum` which stores the defunctionalized evaluation contexts when descending into the first expression `e1`. These constructors contain both the second expression `e2` and the semiring context `(a , m)`. When descending into `e1`, we store the current semiring context in the constructor and continue with the semiring context $\mathsf{ctx}\,\square = (0,\ 1)$:

$$\begin{array}{lll} & \mathsf{app}\,(acc,\ (a,\ m))\,(\mathsf{eval}\,e_1 + \mathsf{eval}\,e_2) & \\ = & \mathsf{eval}'\,((acc,\ (a,\ m)) \bullet (\mathsf{AddL}\,0\,1\,\mathsf{Hole}\,e_2,\ (0,\ 1)))\,e_1 & \{\ (tail)\ \} \\ = & \mathsf{eval}'\,(acc \bullet \mathsf{AddL}\,a\,m\,\mathsf{Hole}\,e_2,\ (0,\ 1) \bullet (0,\ 1))\,e_1 & \{\ (pcomp)\ \} \\ = & \mathsf{eval}'\,(\mathsf{AddL}\,a\,m\,acc\,e_2,\ (0,\ 1))\,e_1 & \{\ (dcomp)\ \} \end{array}$$

This calculation directly follows the recipe for composing with defunctionalized evaluation contexts and can thus be derived algorithmically. Our full implementation becomes:

```
type expr
  Lit(lit : int)
  Add(e1 : expr, e2 : expr)
  Mul(e1 : expr, e2 : expr)

type accum
  Hole
  AddL(a : int, m : int, k : accum, e : expr)
  MulL(a : int, m : int, k : accum, e : expr)
```

```
fun eval(e, acc, a, m)
  match e
    Add(e1, e2) -> eval(e1, AddL(a, m, acc, e2), 0, 1)
    Mul(e1, e2) -> eval(e1, MulL(a, m, acc, e2), 0, 1)
    Lit(n)      -> app(acc, a + m * n)

fun app(acc, n)
  match acc
    Hole -> n
    AddL(a, m, k, e) -> eval(e, k, a + m * n, m)
    MulL(a, m, k, e) -> eval(e, k, a, m * n)
```

In contrast, a translation using just defunctionalized evaluation contexts would require two more constructors `AddR` and `MulR` (just as in the basic example). Unfortunately though, now that we store the semiring context in `accum`, our constructors carry a few more elements than the constructors of `expr`. In a language like Koka, which can reuse constructors of equal size [Lorenzen and Leijen 2022; Reinking, Xie et al. 2021], it would be preferable to obtain constructors of the same size as `expr`, since we could then hope to avoid the allocation of `AddL` and `MulL` by reusing the memory of `Add` and `Mul`. This is often possible when using non-composed defunctionalized evaluation contexts and Lorenzen et al. [2023] show that it is guaranteed to work if the original function has the shape of a map or fold (like `eval`). Alas, the same is not true for composed contexts, since we need to store the additional semiring context. However, using composed contexts like here can still avoid allocations in languages that lack reuse analysis.

Finally, we can reduce the number of elements stored in the constructors and obtain a more natural version of the evaluator by using the distributivity law to push the semiring context into the expression. For the `Add` case we calculate:

$$
\begin{array}{lll}
 & \mathsf{app}\;(\mathit{acc},\;(a,\,m))\;(\mathsf{eval}\,e_1\;+\;\mathsf{eval}\,e_2) & \\
= & \mathsf{app}\,\mathit{acc}\;(\mathsf{app}\;(a,\,m)\;(\mathsf{eval}\,e_1\;+\;\mathsf{eval}\,e_2)) & \{\;(\mathit{papp})\;\} \\
= & \mathsf{app}\,\mathit{acc}\;(a\;+\;m\;*\;(\mathsf{eval}\,e_1\;+\;\mathsf{eval}\,e_2)) & \{\;(\mathit{sapp})\;\} \\
= & \mathsf{app}\,\mathit{acc}\;(a\;+\;m\;*\;\mathsf{eval}\,e_1\;+\;m\;*\;\mathsf{eval}\,e_2) & \{\;\mathit{distributivity}\;\}
\end{array}
$$

At this point, the recursive call to $\mathsf{eval}\,e_1$ is under a semiring context (a, m) and an evaluation context $\square\;+\;m\;*\;\mathsf{eval}\,e_2$. We thus have to store an extra `m` in our `AddL` constructor:

$$
\begin{array}{lll}
 & \mathsf{eval}'\;((\mathit{acc},\;(0,1))\bullet(\mathsf{AddL}\,0\,1\,\mathsf{Hole}\,m\,e_2,\;(a,m)))\;e_1 & \{\;(\mathit{tail})\;\} \\
= & \mathsf{eval}'\;(\mathit{acc}\bullet\mathsf{AddL}\,0\,1\,\mathsf{Hole}\,m\,e_2,\;(0,1)\bullet(a,\,m))\;e_1 & \{\;\mathit{swap}\;\} \\
= & \mathsf{eval}'\;(\mathsf{AddL}\,0\,1\,\mathit{acc}\,m\,e_2,\;(a,\,m))\;e_1 & \{\;(\mathit{dcomp})\;\mathit{and}\;(\mathit{scomp})\;\}
\end{array}
$$

It turns out that in this version, the semiring context stored in the accumulated datatype is always going to be $(0, 1)$, so we can simplify the definition by omitting it. We thus obtain the implementation:

```
type accum
  Hole
  AddL(m : int, k : accum, e : expr)
  MulL(a : int, k : accum, e : expr)

fun eval(e, acc, a, m)
  match e
    Add(e1, e2) -> eval(e1, AddL(m, acc, e2), a, m)
    Mul(e1, e2) -> eval(e1, MulL(a, acc, e2), 0, m)
    Lit(n)      -> app(acc, a + m * n)
```

```
fun app(acc, n)
  match acc
    Hole -> n
    AddL(m, k, e) -> eval(e, k, n, m)
    MulL(a, k, e) -> eval(e, k, a, n)
```

This version is slightly more efficient than the previous one, but the constructors are still too big for reuse analysis to apply. Furthermore, it is unclear whether we can derive this algorithmically as well. Instead, we will stick with the more general version that can be derived directly from the composition of contexts and extend it to derive an evaluator that can also handle subtraction and division.

To extend the expression evaluator to support division, we might try to add a new context for division and show how to compose it with the semiring context. However, this is not straightforward, since $(a + \square)^{-1}$ can not be simplified to $a' + \square^{-1}$ for any other a': the inverse of the sum depends on the $\square$, which is not yet known, and there is no general rule for exchanging the inverse operation with addition. Instead, we need to use an idea from the theory of continued fractions.

5.5 Aside: Continued Fractions

Continued fractions are a representation of the rational (or real) numbers that arises from the Euclidean algorithm. They consist of a sequence of nested additions and fractions with numerator 1. For example, we can calculate the continued fraction of 4.24 as:

$$4.24 = 4 + \tfrac{24}{100} = 4 + \cfrac{1}{\frac{100}{24}} = 4 + \cfrac{1}{4 + \cfrac{4}{24}} = 4 + \cfrac{1}{4 + \cfrac{1}{6}} = 4 + \cfrac{1}{4 + \cfrac{1}{\cfrac{6}{1}}} = 4 + \cfrac{1}{4 + \cfrac{1}{5 + \cfrac{1}{1}}}$$

We can write such a (long-form) continued fraction (with the final '1' left implicit) as $[4, 4, 5]$. Then we can compute its floating point representation with a simple recursive algorithm:

```
fun frac(xs)
  match xs
    Nil -> 1
    Cons a xx -> a + 1 / frac(xx)
```

This algorithm is not tail-recursive and it might be quite difficult to make tail recursive without further insight (and without resorting to general evaluation contexts). However, it is well-known that continued fractions can be calculated by their *convergents*, which is a sequence h_n, k_n with $\mathrm{frac}([a_0, \ldots, a_n]) = h_n \,/\, k_n$. The convergents start with $h_{-2} = 0$, $h_{-1} = 1$, $k_{-2} = 1$, $k_{-1} = 0$ and are further calculated by:

$$\frac{h_n}{k_n} = \frac{a_n * h_{n-1} + h_{n-2}}{a_n * k_{n-1} + k_{n-2}}$$

This gives us a tail-recursive algorithm to calculate the continued fraction (where we write `h1` for h_{n-1}, `h2` for h_{n-2} and equivalent for `k1` and `k2`):

```
fun frac'(xs, h1, k1, h2, k2)
  match xs
    Nil -> (1 * h1 + h2) / (1 * k1 + k2)
    Cons a xx -> frac'(xx, a * h1 + h2, h1, a * k1 + k2, k1)

fun frac(xs)
  frac'(xs, 1, 0, 0, 1)
```

It turns out that we can use the same idea to define a general context that applies to arbitrary sequences of addition, multiplication, and inverses.

5.6 *Modulo Fields Context*

Using the insight from continued fractions, we can define general *field contexts* that support not only addition and multiplication but also additive and multiplicative inverses. We define the context F as:

$$F ::= \square \mid a + F \mid m * F \mid F^{-1}$$

It turns out that we use the same convergent representation as for continued fractions, where we keep four numbers:

$$\frac{x * h_1 + h_2}{x * k_1 + k_2}$$

and define the fold operation as:

$$(\!|\square|\!) = \frac{x * 1 + 0}{x * 0 + 1} \qquad (\!|m * F|\!) = \frac{x * m + 0}{x * 0 + 1} \bullet (\!|F|\!)$$

$$(\!|a + F|\!) = \frac{x * 1 + a}{x * 0 + 1} \bullet (\!|F|\!) \qquad (\!|F^{-1}|\!) = \frac{x * 0 + 1}{x * 1 + 0} \bullet (\!|F|\!)$$

We can apply a field context to an expression by substituting the expression for x. Similarly, we can compose two field contexts by substituting the second context into the first context and simplifying the expression. Our context is defined as:

$$
\begin{array}{lll}
(\mathit{fctx}) & \mathsf{ctx}\, F & = (\!|F|\!) \\
(\mathit{fcomp}) & \dfrac{x * h_1 + h_2}{x * k_1 + k_2} \bullet \dfrac{y * h_1' + h_2'}{y * k_1' + k_2'} & = \dfrac{y * (h_1' * h_1 + k_1' * h_2) + (h_2' * h_1 + k_2' * h_2)}{y * (h_1' * k_1 + k_1' * k_2) + (h_2' * k_1 + k_2' * k_2)} \\
(\mathit{fapp}) & \mathsf{app}\, \dfrac{x * h_1 + h_2}{x * k_1 + k_2}\, e & = \dfrac{e * h_1 + h_2}{e * k_1 + k_2}
\end{array}
$$

and the context laws hold.

5.7 *An Advanced Expression Evaluator*

Using the field contexts, we can extend our expression evaluator to support arbitrary field operations. Our implementation arises directly from the obvious expression evaluator which folds the expression into a rational number:

```
fun eval(e : expr) : rat
  match e
    Add(e1, e2) -> eval(e1) + eval(e2)
    Mul(e1, e2) -> eval(e1) * eval(e2)
    Neg(e1)     -> from-int(-1) * eval(e1)
    Inv(e1)     -> from-int(1) / eval(e1)
    Lit(n)      -> from-int(n)
```

We can directly define the field context as a datatype, where we define `empty()`, `add(a)`, `mul(a)`, `inv()` to correspond to the fold operations:

```
type fctx
  Fctx( h1 : rat, h2 : rat, k1 : rat, k2 : rat )

fun fctx/app(f : fctx, r : rat) : rat
  (f.h1 * r + f.h2) / (f.k1 * r + f.k2)

fun fctx/add(a : rat) : fctx
  Fctx( from-int(1), a, from-int(0), from-int(1) )

...
```

Then we use the TRMC algorithm with the composition of defunctionalized contexts and field contexts to obtain a tail-recursive version that uses a field context for the field operations and a defunctionalized context for the recursive calls that leave an expression to be evaluated:

```
type accum
  Hole
  AddL(f : fctx, k : accum, e : expr)
  MulL(f : fctx, k : accum, e : expr)

fun fctx/eval(e : expr, acc : accum, f : fctx)
  match e
    Add(e1, e2) -> eval(e1, AddL(f, acc, e2), empty())
    Mul(e1, e2) -> eval(e1, MulL(f, acc, e2), empty())
    Neg(e1)     -> eval(e1, acc, comp(f, mul(-1)))
    Inv(e1)     -> eval(e1, acc, comp(f, inv()))
    Lit(n)      -> app(acc, app(f, from-int(n)))

fun fctx/app(acc : accum, r : rat)
  match acc
    Hole -> r
    AddL(f, k, e) -> eval(e, k, comp(f, add(r)))
    MulL(f, k, e) -> eval(e, k, comp(f, mul(r)))
```

The final derived program is actually quite sophisticated and fully tail-recursive. We believe that deriving this algorithm manually would be non-trivial. Moreover, it only allocates a small amount of memory while descending the left-spine. In contrast, a simple application of defunctionalized contexts without field contexts would require us to allocate a constructor even in the `Neg` and `Inv` cases, which would be less efficient.

6 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:

$$\mathsf{K} \quad ::= \quad \square \mid C^k\, v_1 \ldots \mathsf{K} \ldots v_k$$

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:

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

Similar to general evaluation contexts (Section 4.1), the context laws hold trivially for such definition (App. B.5 in the supplement) – and just as with general evaluation contexts, the *map* function 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.

6.1 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]:

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

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:

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

and application:

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

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 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 2. (*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 $\Gamma|_{\mathsf{N}}, f\,;\,\varnothing \vdash_{\mathsf{M}} \mathsf{fun} f'\ =\ \lambda x_1 \ldots x_n.\, \lambda k.\, [\![e]\!]_{f,k}\ :\ \tau_1 \to \ldots \to \tau_n \to ((\tau,\ \tau)\ \mathit{hfun}) \to \tau$.

To show this, we need a variant of the general replacement lemma [Hindley and Seldin 1986, Lemma 11.18; Wright and Felleisen 1994, Lemma 4.2] 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 App. B.6 in the supplement, which also contains the full type rules adapted to our calculus.

6.2 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 ($\square$). 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 [Lorenzen and Leijen 2022; Reinking, Xie et al. 2021]. 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 $H \mid e_1 \;\longmapsto_r\; H' \mid e_2$, which reduces a heap H and an expression e to a new heap H' and expression e_2 [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.

6.2.1 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:

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

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

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

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:

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

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 $\Box$, we can further refine this to:

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

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

6.2.2 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 $[H]^n_x$:

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

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 (*linearcons*) we can derive that we can append a unique linear chain as well:

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

6.2.3 Contexts as a Linear Chain

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

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

since we can always arrange any K to have this form by let-binding the values v. 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 (App. B.7 in the supplement).

6.2.4 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 $\mathsf{ctx}\,\mathsf{K} = (\!|\mathsf{K}|\!)$. For an empty context we use an empty tuple ($(\!|\square|\!) = \langle\rangle$), but otherwise we can specify the fold as:

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

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 App. B.8 in the supplement), where we get following definition for $(\!|\_|\!)$:

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

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:

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

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

6.2.5 *Updating a Context*

Before we can define in-place application, we need an in-place substitution operation $\mathsf{subst}\,\langle x, y@i\rangle\,z$ 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:

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

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

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

and for the extension we have:

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

This leads to the following inductive definition of subst:

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

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 instead using the tuple address $y@i$ directly to update the hole in-place. In essence, due to the uniqueness 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.

6.2.6 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 $\mathsf{K} \neq \square$):

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

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 App. B.9 in the supplement. 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 App. B.9 in the supplement):

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

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 3 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!

7 Modulo Constructor Contexts: 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* [Danvy and Filinski 1990; Shan 2007; Sitaram and Felleisen 1990], or in general with algebraic effect handlers [Plotkin and Power 2003; Plotkin and Pretnar 2009], whenever a continuation or handler resumption can be invoked more than once. Note that if only single-shot continuations or resumptions are allowed (as in OCaml [Dolan et al. 2015] for example), the control flow is still always linear and the TRMc transformation still sound. Since the Koka language relies foundationally on general 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]:

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

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)
       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 2020 2021]. 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.

7.1 Dynamic Copying via Reference Counting

Our context composition is defined in terms of context application, which in turn relies on on the in-place substitution (Section 6.2.5):

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

This is the operation that eventually fails if the runtime context x is not unique. In Section 6.2.5, 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:

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

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

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

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

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

We can show the context laws still hold for these definitions (see App. B.10 in the supplement). 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 App. 7.2 in the supplement). 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.

7.2 Efficient Code Generation

As an example of the code generation of our TRMC scheme we consider the `map` function from our benchmarks in Section 9. The `map` function is specialized by the compiler for the increment function, and after the TRMC transformation we have (something like):

```
fun map_trmc'( xs : list<int32>, k : ctx<list<int32>> ) : list<int32>
  match xs
    Nil -> app k Nil
    Cons(x,xx) ->
      val y = x+1
      val c = Cons2(y,□)
      map_trmc'(xx, comp(k, Ctx(c,c@2)))

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 (defined in the `std/core/types` module). For efficiency we represent the empty tuple with a `null` address for the hole. Eventually, such value type is passed in registers (`x19` and `x21`), 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,ctx 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
```

Note in particular how the header for the `Cons` node in the context is set as `mov w24, #x020202` where, from left-to-right, we initialize the tag (`0x02`), the context path field (`0x02`) and the total number of fields (also `0x02`). As such, maintaining context paths comes for free since it is done as part of header initialization. Also we see the reuse of Perceus reference counting [Lorenzen and Leijen 2022; Reinking, Xie et al. 2021] in action, where the `Cons` node that is matched (in `x0`) is reused for the context `Cons` node (also in `x0`). Since the effect inferred for the specialized map function is `total` the check for uniqueness of the context is removed as it the context is guaranteed to be used lineraly.

7.3 First-class Constructor Contexts

Now that we can dynamically copy constructor contexts along the context paths efficiently at runtime (Section 7.1), it actually allows us to expose constructor contexts as first-class values in the language [Lorenzen et al. 2024]. This abstraction can safely encapsulate the limited form of mutation necessary to implement a minamide tuple, while still having a purely functional interface.

In the Koka language, it is now possible to define a constructor context manually using the `ctx` K expression with a single hole denoted by an underscore `_`. For example, we can write a list constructor context as `ctx Cons(1,_)` or a binary tree constructor context as `ctx Node (Node(Leaf,1,Leaf),2,_)`. The composition operation ($\bullet$) is written as `(++)` while application is written as `(++.)`. For example, the expression `(ctx Cons(1,_)) ++ (ctx Cons(2,_)) ++. Nil` evaluates to `(ctx Cons(1,Cons(2,_))) ++. Nil` and then to `[1,2]`. Using first-class constructor contexts, we can implement the TRMC transformation of the `map` function directly in Koka as well:

```
fun map-trmc'( xs : list<a>, f : a -> b, acc : ctx<list<b>> ) : list<b>
  match xs
    Cons(x,xx) -> map-trmc'( xx, f, acc ++ ctx Cons(f(x),_) )
    Nil        -> acc ++. Nil

fun map-trmc( xs : list<a>, f : a -> b ) : list<b>
  map-trmc'( xs, ctx _ )
```

(and the TRMc transformation becomes a source-to-source transformation). A `ctx` K expression is compiled using the fold function $(\!|K|\!)$ as shown in Section 7.1 such that each constructor context has a context path at runtime. As shown in Section 7.2, the Koka compiler compiles a context like `ctx Node(Node(Node(Leaf,1,Leaf),2,_),5,Leaf)` internally into a Minamide tuple:

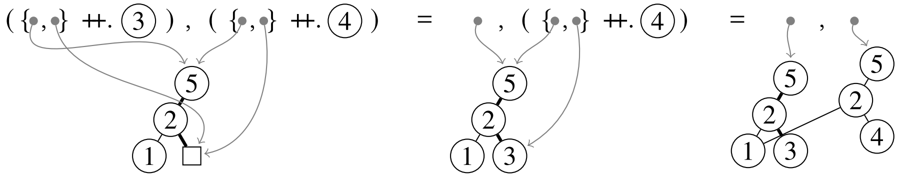
```
val x = Node3(Node(Leaf,1,Leaf),2,hole) in Ctx(Node1(x,5,Leaf), x@3)
```

where each constructor along the context path is annotated with a child index (1 and 3).

When we compose or apply a context we can now copy shared contexts only when needed. If the contexts happen to be used linearly, then all operations execute in constant time, just as in Minamide's approach; but we now have full functional semantics and any subsequent substitutions on the same context work correctly (but will take linear time in the length of the context path). The expression `val c = ctx Cons(1,_) in (c ++. [2] , c ++. [3])`, where the context `c` is shared, evaluates correctly to `([1,2],[1,3])`.

7.4 Dynamic Copying without Reference Counting

Lorenzen et al. [2024] show that it is possible to support first-class constructor contexts even in languages without precise reference counts. Their proposed implementation (also suggested by Gabriel Scherer) uses a special distinguished value for a runtime hole □ that is never used by any other object. A substitution now first checks the value at the hole: if it is a □ value, the hole is substituted for the first time and we just overwrite the hole in-place (in constant time). However, any subsequent substitution on the same context will find some object instead of □. At this point, we first dynamically copy the context path (in linear time) and then update the copy in-place.



The illustration above (due to Lorenzen et al. [2024]) shows a more complex example of a shared tree context that is applied to two separate nodes. The runtime context path is denoted here by bold edges. The intermediate state is interesting as it is both a valid tree, but also a part of the tree is shared with the remaining context, where the hole points to a regular node now. When that context is applied, only the context path (node 5 and 2) is copied first where all other nodes stay shared (in this case, only node 1).

However, it turns out that this simple approach is not sound without further restrictions. For general first-class contexts, the second context can be arbitrary (instead of always a constant `ctx` in the TRMC case), the context composition operation `c1 ++ c2` needs an extra check in order to avoid creating cycles: we check if `c2` has an already overwritten hole or if the hole in `c2` is at the same address as in `c1`. In either case, `c2` is copied along the context path.

Figure 3 shows a partial implemention in C code of how one can implement constructor contexts in a runtime for languages without precise reference counting. We assume that `HOLE` is the distinguished value for unfilled holes (□). When we compose two contexts we need to ensure we can handle shared contexts as well where we copy a context along the context path if needed (using `ctx_copy`).

In the application and composition functions, the check `(A)` sees if the hole in `c1` is already overwritten (where `*c1.hole != HOLE`). In that case we copy `c1` along the context path as shown in Section 7.1 to maintain referential transparency.

However, in the composition operation we also need to do a similar check for `c2` as well in order to avoid cycles: the second check `(B)` checks if `c2` has an already overwritten hole, but also if the hole in `c2` is the same as in `c1`. In either case, `c2` is copied along the context path. Effectively, both checks ensure that the new context that is returned always ends with a single fresh `HOLE`. Let's consider some examples of shared contexts. A basic example is a simple shared context, as in:

```
val c = ctx Cons(1,_) in (c ++. [2], c ++. [3])
```

which evaluates to `([1,2],[1,3])`. Here, during the second application, check `(A)` ensures the shared context `c` is copied such that the list `[1,2]` stays unaffected.

```
struct ctx_t {              // a Minamide context
  heap_block_t*   root;
  heap_block_t**  hole;
};

struct ctx_t ctx_copy( struct ctx_t c ) {
  struct ctx_t d = { .root = c.root, .hole = c.hole };
  if( c.root == NULL ) return d;
  heap_block_t** prev = &(c.root);
  heap_block_t** next = &(d.root);

  while( prev != c.hole ) {
    *next = heap_block_copy( *prev );
    prev = (*prev)->children + ((*prev)->ctx_path);
    next = (*next)->children + ((*next)->ctx_path);
  }
  d.hole = next;
  return d;
}

// (++.) : cctx<a,b> -> b -> a
heap_block_t*  ctx_apply( struct ctx_t c1, heap_block_t* x )
{
  // is c1 an empty context?
  if (c1.root == NULL) return x;

  // copy c1 ?
  struct ctx_t  d1 = (*c1.hole != HOLE ?  ctx_copy(c1) : c1);                    // (A)

  *d1.hole = x;
  return d1.root;
}

// (++) : cctx<a,b> -> cctx<b,c> -> cctx<a,c>
struct ctx_t  ctx_compose( struct ctx_t c1, struct ctx_t c2 )
{
  // is c1 or c2 an empty context?
  if (c1.root == NULL) return c2;
  if (c2.root == NULL) return c1;

  // copy c1 ?
  struct ctx_t  d1 = (*c1.hole != HOLE  ?  ctx_copy(c1) : c1 );                  // (A)

  // copy c2 ? (needed to avoid cycles)
  struct ctx_t  d2 = ((*c2.hole != HOLE || c1.hole == c2.hole)
                     ? ctx_copy(c2) : c2 );                                      // (B)

  *d1.hole = d2.root;
  d1.hole  = d2.hole;
  return d1;
}
```

Fig. 3. Implementing constructor composition and application in the runtime system (for languages without precise reference counts).

A more tricky example is composing a context with itself:

```
val c = ctx Cons(1,_) in (c ++ c) ++. [2]
```

which evaluates to `[1,1,2]`. The check `(B)` here copies the appended `c` (since `c1.hole == c2.hole`). In this example the potential for a cycle is immediate, but generally it can be obscured with a shared context inside another. Consider:

```
val c1 = ctx Cons(1,_)
val c2 = ctx Cons(2,_)
val c3 = ctx Cons(3,_)
val c  = c1 ++ c2 ++ c3 in  (c ++ c2) ++. [4]
```

which evaluates to `[1,2,3,2,4]`. The check `(B)` again copies the appended `c2` in `c ++ c2` (since `*c2.hole != HOLE`).

Note that the (B) check in composition is sufficient to avoid cycles. In order to create a cycle in the context path, either `c1` must be in the context path of `c2` (I), or the `c2` in the context path of `c1` (II). For case (I), if `c1` is at the end of `c2`, then their holes are at the same address where `c1.hole == c2.hole`. Otherwise, if `c1` is not at the end, then `*c1.hole != HOLE` and we have copied `c1` already due to check (A). For case (II) the argument is similar: if `c2` is at the end of `c1` we again have `c1.hole == c2.hole`, and otherwise `*c2.hole != HOLE`.

The implementation using precise reference counting is not very different from the one without reference counting. The main difference is in the checks (A) and (B), which become:

```
// copy c1 ?
struct ctx_t  d1 = (!is_unique(c1.root) ?  ctx_copy(c1) : c1 );              // (A)

// copy c2 ? (needed to maintain ctx paths where each node beside the root is unique)
struct ctx_t  d2 = (!is_unique(c2.root) ?  ctx_copy(c2) : c2 );              // (B)
```

This is the implementation that is used in the Koka runtime system. The (B) check here is required to maintain the invariant that context paths always form *unique chains* (Section 7.1). From this property it follows directly that no cycles can occur in the context path.

7.5 Runtime Behaviour

Interestingly, the two implementations, with or without precise reference counting,do differ in their runtime performance characteristics, which are dual to each other in terms of space and time.

7.5.1 Time

The implementation without reference counting only copies on demand when the hole is already filled, whereas our earlier implementation with reference counts copies whenever the context is found to be not unique upon filling the hole. This can be a problem, if the context is later discarded without being used. Consider the knapsack program, which in its last iteration may call itself on a one-element list `[x]` with `x = w`. For this special case, the code reduces to:

```
fun knapsack'(x : int, k : ctx<list<int>>) : <nondet,div> list<int>
  val v = if flip() then x else fail() in k ++. Cons(v, [])
```

This computation is run twice, where the first run successfully returns `k ++. (Cons(x, []))` but the second run fails. The reference counting-based implementation has to copy `k` in the first run, since its reference count is not one (due to `k` being captured for the second run). In contrast, assuming that the hole in `k` is not yet filled, the new implementation can simply fill the hole of `k` with `Cons(x, [])` in the first run without copying. Since `k` is discarded in the second run, no copying is needed at all. We will come back to this point in Section 9, where we see that the reference counting implementation in Koka does not perform well in a backtracking search, presumably due to this issue.

7.5.2 Space

The implementation without precise reference counts can use more space though than the one based on reference counting. This can occur when a context accidentally holds on to values that have been written into its hole. Consider an earlier state of the knapsack program, where it may process a list `vs = Cons(v, vv)` with `v > w`. Then we can simplify the code to:

```
if flip()
  then knapsack(w - v, Cons(v, vv), k ++ Cons(v, _))
  else val v' = select(vv) in knapsack(w - v', Cons(v, vv), k ++ Cons(v, _))
```

Following the `flip()`, we first try to use `v` as our element. But since `v > w`, this computation fails and we backtrack. However, our new algorithm may have written `Cons(v, _)` into the hole of `k`. This value is now garbage, but this may not be obvious to a garbage collector or reference counting scheme, since `k` is still live. Only when backtracking to the second run do we copy `k` and discard the old value.

In contrast, the implementation based on reference counting would have copied (and discarded) `k` in the first run already. Unlike the new implementation, it is *garbage-free* [Reinking, Xie et al. 2021] and guarantees that no space is used for values that are no longer needed. For this reason, we prefer the implementation via reference counting in Koka, using the other implementation for GC-based languages.

8 Programming with First-class Constructor Contexts

First-class constructor contexts turn out to be a powerful feature, and they allow us to write many programs by hand that would be hard to generate automatically from a general TRMC transformation. In this section we explore some of these programs, all of which can be written in Koka.

8.1 Modulo Cons Products

The `partition` function calls a predicate on each element of a list and appends it to one of two piles depending on the result:

```
fun partition(p, xs)
  match xs
    Nil -> (Nil, Nil)
    Cons(x, xx) ->
      val (yes, no) = partition(p, xx)
      if p(x)
        then (Cons(x, yes), no)
        else (yes, Cons(x, no))
```

The recursive call to `partition` is followed by a pattern match on the resulting tuple, an if-statement and finally the constructor application. This does not fit the TRMc transformation directly, but it also might not seem too different – and indeed this function was suggested as fruitful target for an expanded TRMC translation both by Bour et al. [2021] and the conference version of this paper.

However, in order to make this function tail recursive, the `p(x)` call would have to be moved *before* the recursive call. That can be done by a compiler if `p` is pure, but what if `p` may perform side-effects? Thus, even an extended TRMc transformation could only apply if the user first rewrote their code to:

```
fun partition(p, xs)
  match xs
    Nil -> (Nil, Nil)
    Cons(x, xx) ->
      val ok = p(x)
      val (yes, no) = partition(p, xx)
      if ok
        then (Cons(x, yes), no)
        else (yes, Cons(x, no))
```

The conference version of this paper describes a transformation that recognizes that the pattern-match on the returned tuple is mirrored in the creation of a new tuple and looks for constructor contexts inside the created tuple.

However, it may not be worth implementing such specific transformation as we can easily rewrite it manually using two explicit first-class constructor contexts for `yes` and `no`:

```
fun partition(p, xs, yes, no)
  match xs
    Nil -> (yes ++. Nil, no ++. Nil)
    Cons(x, xx) ->
      if p(x)
        then partition(p, xx, yes ++ ctx Cons(x, _), no)
        else partition(p, xx, yes, no ++ ctx Cons(x, _))
```

The resulting code is clearer than the version with an explicit `ok` variable, and arguably even clearer than the original version. . . and even being more efficient. For this reason, we now recommend that programmers use first-class constructor contexts directly for examples like this.

8.2 *Difference Lists*

Another example of future work described by Bour et al. [2021] is the `flatten` function. This function calls itself recursively and passes the result to the `append` function on lists:

```
fun append(xs : list<a>, ys : list<a>) : list<a>
  match xs
    Nil -> ys
    Cons(x, xx) -> Cons(x, append(xx, ys))

fun flatten(xss : list<list<a>>) : list<a>
  match xss
    Nil -> Nil
    Cons(xs, xss) -> append(xs, flatten(xss))
```

While `append` is tail recursive modulo cons, `flatten` is not. However, `append` is just a sequence of constructor applications ending in the second argument, and we can easily rewrite it using a first-class constructor context returned from `append` (i.e. a *difference* list):

```
fun append(acc : ctx<list<a>>, xs : list<a>) : ctx<list<a>>
  match xs
    Nil -> acc
    Cons(x, xs) -> append(acc ++ ctx Cons(x, _), xs)

fun flatten-acc(acc : ctx<list<a>>, xss : list<list<a>>) : list<a>
  match xss
    Nil -> acc ++. Nil
    Cons(xs, xss) -> flatten-acc(append(acc, xs), xss)

fun flatten( xss : list<list<a>> ) : list<a>
  flatten-acc(ctx _, xss)
```

8.3 Composing Constructor Contexts

Another example which illustrates the usefulness of first-class conntexts that can be stored in data structures, is the composition of constructor contexts with defunctionalized evaluation contexts. While constructor contexts naturally apply to the map over a list, they do not apply directly to a map over trees:

```
type tree<a>
  Leaf
  Bin(l : tree<a>, a : a, r : tree<a>)

fun tmap(t, f)
  match t
    Bin(l, x, r) -> Bin(tmap(l, f), f(x), tmap(r, f))
    Leaf -> Leaf
```

Here, the first recursive call to `tmap` is not in a constructor context and thus the TRMc transformation alone is not enough to make this tail recursive. However, instead of resorting to full defunctionalized evaluation contexts, we can use them only for descending into the left child and keep using constructor contexts to descend into the right branch:

```
type accum<a,b>
  Hole
  Accum(acc : accum<a,b>, top : ctx<tree<b>>, x : a, r : tree<a>)

fun tmap-acc(t, f, acc, top)
  match t
    Bin(l, x, r)    -> tmap-acc(l, f, Accum(acc, top, x, r), ctx _)
    Leaf            -> tmap-app(f, top ++. Leaf, acc)
```

```
fun tmap-app(f, l, acc)
  match acc
    Hole -> l
    Accum(acc, top, x, r) -> tmap-acc(r, f, acc, top ++ ctx Bin(l, f(x), _))
```

This function immediately follows from the technique described in Section 5.3. It extends the `acc` accumulator whenever it goes into the left subtree, and extends the `top` accumulator whenever it goes into the right subtree. While a version using only defunctionalized evaluation contexts corresponds to pointer reversal [Schorr and Waite 1967], this version reverses only the pointers going to the right child, but leaves the pointers to the left child intact.

8.4 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>
  Empty
  Zero(        s : seq<(a, a)> )
  One ( x : a, s : seq<(a, a)> )

fun cons(x : a, s : seq<a>) : seq<a>
  match s
    Empty      -> One(x, Empty)
    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)>`. It turns out that for polymorphically recursive code, performing the translation can lead to code that is not typeable is System F. This issue is well-known for defunctionalized evaluation contexts, where GADTs are required to regain typability [Pottier and Gauthier 2004]. Analogously, we give two type parameters to first-class constructor contexts `cctx<a,b>` where `a` corresponds to the type of the root and `b` to the type of the hole. Our primitive operations have the general types:

```
alias ctx<a> = cctx<a,a>

fun (++)( c1 : cctx<a,b>, c2 : cctx<b,c> ) : cctx<a,c>
fun (++.)( c : cctx<a,b>, x : b) : a
```

It turns out that this encapsulates the necessary type information to type the result of the translation for polymorphic recursion. Even though Koka has an intermediate core representation based on System F, the application and composition functions are primitives and Koka transforms the above function without problems. Our `cons` function is translated to:

```
fun cons(x : a, s : seq<a>, acc : cctx<seq<b>, seq<a>>) : seq<b>
  match s
    Empty      -> acc ++. One(x, Empty)
    Zero(ps)   -> acc ++. One(x, ps)
    One(y, ps) -> cons((x, y), ps, acc ++ ctx Zero(_))
```

9 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 7. 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.
- *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 4. 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 7 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

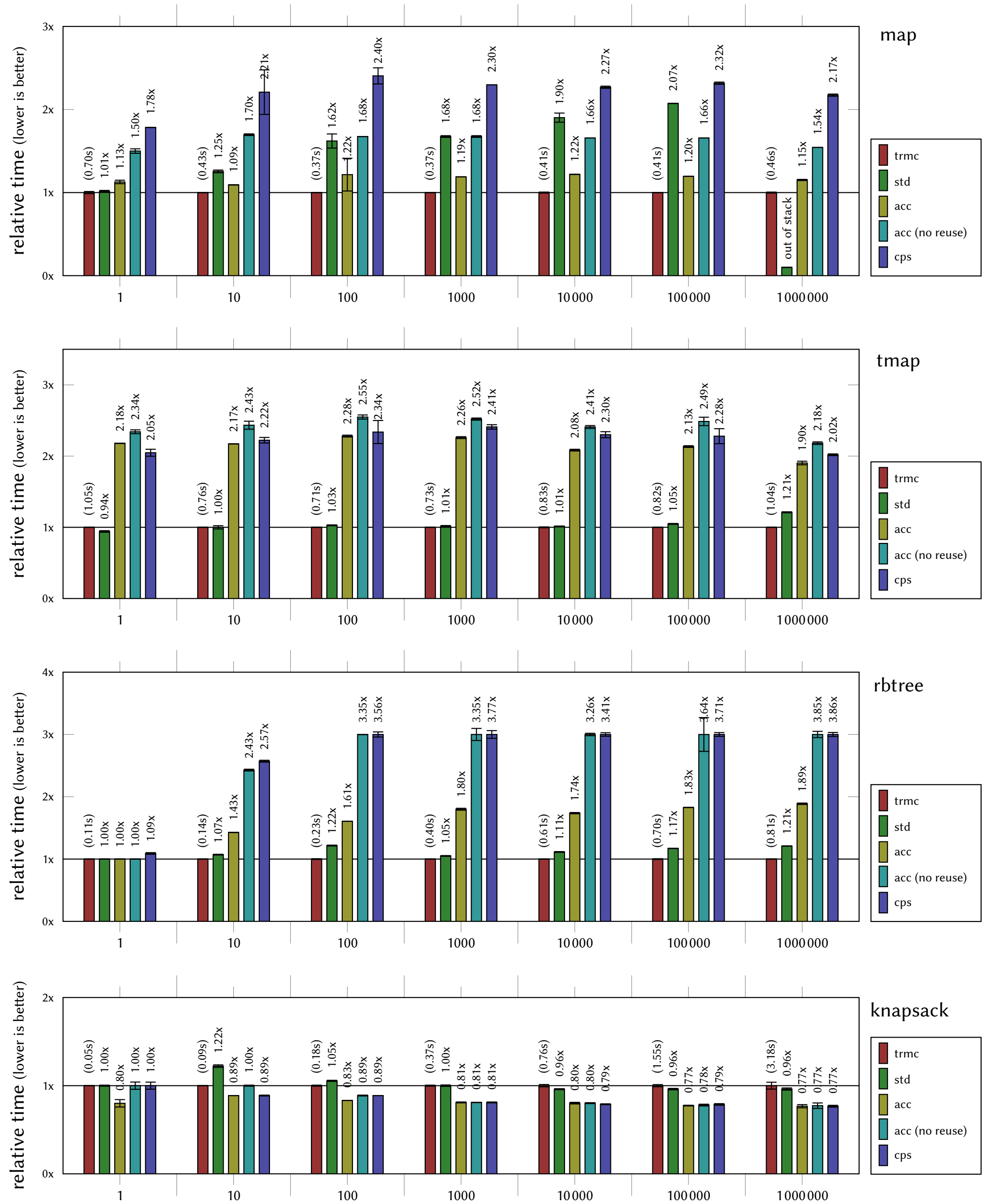


Fig. 4. Benchmarks on Ubuntu 20.04 (AMD 5950x), Koka v2.4.1-dev. The benchmarks are map over a list (*map*), map over a tree (*tmap*), balanced red-black tree insertion (*rbtree*), and the *knapsack* program that use non-linear control flow. Each workload is scaled to process the same number of total 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*).

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.

10 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 4.4. A more precise description of the TRMc transformation was given by Friedman and Wise [1975].

More recently, Bour et al. [2021] describe 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:

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

Here a single rule does various transformations that we treat as orthogonal, such as folding, extraction, instantiation of composition, and the actual TRMc transformation.

In logic languages, difference lists [Clark and Tärnlund 1977] can be used to encode a form of TRMc: difference lists are usually presented as a pair (L, X) where X is a logic variable which is the last element of the list L. With in-place update of the unification variable X, one can thus append to L in constant time – quite similar to our constructor contexts. Engels [2022] describes an implemention of TRMC for the Elm language, that can also tail-optimize calls to the right of a list append by keeping the last cell of the right-appended list as a context. 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. Lazyness works similar to TRMc for the functions we consider: recursive calls guarded by a constructor are thunked and incremental forcing can happen without using the stack. The listless machine [Wadler 1984] is an elegant model for this behaviour.

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 as a composition of partially applied append functions (which are sometimes also called difference lists). His function `rep(xs)` (defined as `fn(ys) xs ++ ys`) creates such abstract list, and is equal to our ctx when instantiated to append functions and list contexts (Section 4.1). Similarly, his `abs(f)` function (defined as `f []`) corresponds to our app k [] in that case, and finally, the correctness condition would correspond to 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 [Danvy and Nielsen 2001; Reynolds 1972] has often been used to eliminate all higher-order calls and obtain a first-order version of a program. Wand and Friedman [1978] describes a defunctionalization algorithm in the context of 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 usage. 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 4.3. However, they do not give an algorithm for this and the technique has so far mainly been used manually [Danvy and Goldberg 2002; Gibbons 2022].

An early implementation of TRMc in a typed language was in the OPAL compiler [Didrich et al. 1994]. Similar to Bour et al. [2021] they also used destination passing style compilation with an extra destination argument where the final result is written to. Like Koka and Lean, OPAL also managed memory using reference counting and could reuse matched constructors [Schulte and Grieskamp 1992]. Reuse combines well with TRMc and in recent work Lorenzen and Leijen [2022] show how this can be used to speed up balanced insertion into red-black trees using the *functional but in-place* (FBIP) technique. 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, reuse based on dynamic reference counts can improve upon this by for example also reusing the original data for the accumulator (and generalize to non-linear control).

We are using the linearity of the Perceus heap semantics [Lorenzen and Leijen 2022; Reinking, Xie et al. 2021] to reason about linear chains and the essence of in-place updates. In our case, these linear chains are used to reason about the shape of a separate part of the heap. This suggest that separation logic [Reynolds 2002] could also be used effectively for such proofs. For example, Moine et al. [2023] use separation logic to reason about space usage under garbage collection.

11 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.

Conflicts of Interest. None

References

Andrew W. Appel. 1991. *Compiling with Continuations*. Cambridge University Press. doi:10.1017/CBO9780511609619.

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

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

Joshua Bloch. 2008. *Effective Java (2nd Edition) (The Java Series)*. 2nd edition. Prentice Hall PTR, USA. doi:10.5555/1377533.

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

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

Olivier Danvy, and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 151–160. LFP '90. Nice, France. doi:10.1145/91556.91622.

Olivier Danvy, and Mayer Goldberg. 2002. There and Back Again. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, 230–234. ICFP '02. Pittsburgh, PA, USA. doi:10.1145/581478.581500.

Olivier Danvy, and Lasse R. Nielsen. 2001. Defunctionalization at Work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 162–174. PPDP '01. Florence, Italy. doi:10.1145/773184.773202.

Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. 1994. OPAL: Design and Implementation of an Algebraic Programming Language. In *Programming Languages and System Architectures*, 228–244. Springer. doi:10.1007/3-540-57840-4_34.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. Sep. 2015. Effective Concurrency through Algebraic Effects. In *OCaml Workshop*.

Damien Doligez, and Xavier Leroy. 1993. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 113–123. POPL '93. Association for

Computing Machinery, New York, NY, USA. doi:10.1145/158511.158611.

Zdeněk Dvořák. 2004. Declarative World Inspiration. In *GCC Developers' Summit*, 25. See also https://github.com/gcc-mirror/gcc/blob/master/gcc/tree-tailcall.cc.

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

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 237–247. PLDI '93. Albuquerque, New Mexico, USA. doi:10.1145/155090.155113.

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

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

Robert Harper. 2012. 15-150 Equational Reasoning Guide. 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. Sep. 1986. *Introduction to Combinators and Lambda-Calculus. Lambda-Calculus and Combinators, an Introduction*. Cambridge University Press. doi:10.1017/CBO9780511809835.

Gérard P. Huet. 1997. The Zipper. *Journal of Functional Programming* 7 (5): 549–554. doi:10.1017/S0956796897002864.

R John Muir Hughes. 1986. A Novel Representation of Lists and Its Application to the Function "reverse." *Information Processing Letters* 22 (3). Elsevier: 141–144. doi:10.1016/0020-0190(86)90059-1.

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

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

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

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

Daan Leijen, and Anton Lorenzen. Jul. 2022. *Tail Recursion Modulo Context – An Equational Approach*. MSR-TR-2022-18. Microsoft Research.

Daan Leijen, and Anton Lorenzen. 2023. Tail Recursion Modulo Context: An Equational Approach. *Proceedings of the ACM on Programming Languages* 7 (POPL). ACM New York, NY, USA: 1152–1181. doi:10.1145/3571233.

Anton Lorenzen, and Daan Leijen. Aug. 2022. Reference Counting with Frame Limited Reuse. In *Proc. ACM Program. Lang.*, volume 6. ICFP. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3547634.

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proceedings of the ACM on Programming Languages* 7 (ICFP). ACM New York, NY, USA: 275–304. doi:10.1145/3607840.

Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. 2024. The Functional Essence of Imperative Binary Search Trees. *Proceedings of the ACM on Programming Languages*, number PLDI. ACM New York, NY, USA.

Luke Maurer, Paul Downen, Zena M Ariola, and Simon Peyton Jones. 2017. Compiling without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 482–494.

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

Yasuhiko Minamide. 1998. 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. doi:10.1145/268946.268953.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 271–283. POPL '96. Association for Computing Machinery, New York, NY, USA. doi:10.1145/237721.237791.

Alexandre Moine, Arthur Charguéraud, and François Pottier. Jan. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection (Extended Version). In *Proceedings of the 50th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1–27. POPL '23.

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

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

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

François Pottier, and Nadji Gauthier. 2004. Polymorphic Typed Defunctionalization, POPL '04, . Association for Computing Machinery, New York, NY, USA, 89–98. doi:10.1145/964001.964009.

François Pottier, and Jonathan Protzenko. 2013. 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. doi:10.1145/2500365.2500598.

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

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

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, 55–74. LICS '02. IEEE Computer Society, USA.

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

Herbert Schorr, and William M Waite. 1967. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. *Communications of the ACM* 10 (8). ACM New York, NY, USA: 501–506.

Wolfram Schulte, and Wolfgang Grieskamp. 1992. Generating Efficient Portable Code for a Strict Applicative Language. In *Declarative Programming, Sasbachwalden 1991*, 239–252. Springer. doi:10.1007/978-1-4471-3794-8_16.

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

Dorai Sitaram, and Matthias Felleisen. 1990. Control Delimiters and Their Hierarchies. *LISP and Symbolic Computation* 3 (1): 67–99. doi:10.1007/BF01806126.

Jonathan Sobel, and Daniel P. Friedman. 1998. Recycling Continuations. In *Proc. of the Third ACM SIGPLAN Int. Conf. on Functional Programming*, 251–260. ICFP '98. Baltimore, Maryland, USA. doi:10.1145/289423.289452.

Andrew Tolmach, and Dino P Oliva. 1998. From ML to Ada: Strongly-Typed Language Interoperability via Source Translation. *Journal of Functional Programming* 8 (4). Cambridge University Press: 367–412. doi:10.1017/S0956796898003086.

Sebastian Ullrich, and Leonardo de Moura. Sep. 2019. Counting Immutable Beans – Reference Counting Optimized for Purely Functional Programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL'19)*. Singapore.

Philip Wadler. 1984. 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. LFP '84. Association for Computing Machinery, New York, NY, USA. doi:10.1145/800055.802020.

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

Mitchell Wand, and Daniel P. Friedman. Jan. 1978. Compiling Lambda-Expressions Using Continuations and Factorizations. *Comput. Lang.* 3 (4). Pergamon Press, Inc., USA: 241–263. doi:10.1016/0096-0551(78)90042-5.

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

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

Ningning Xie, and Daan Leijen. 2020. Effect Handlers in Haskell, Evidently. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, 95–108. Haskell 2020. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3406088.3409022.

Ningning Xie, and Daan Leijen. Aug. 2021. Generalized Evidence Passing for Effect Handlers: Efficient Compilation of Effect Handlers to C. In *Proc. ACM Program. Lang.*, volume 5. ICFP. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3473576.

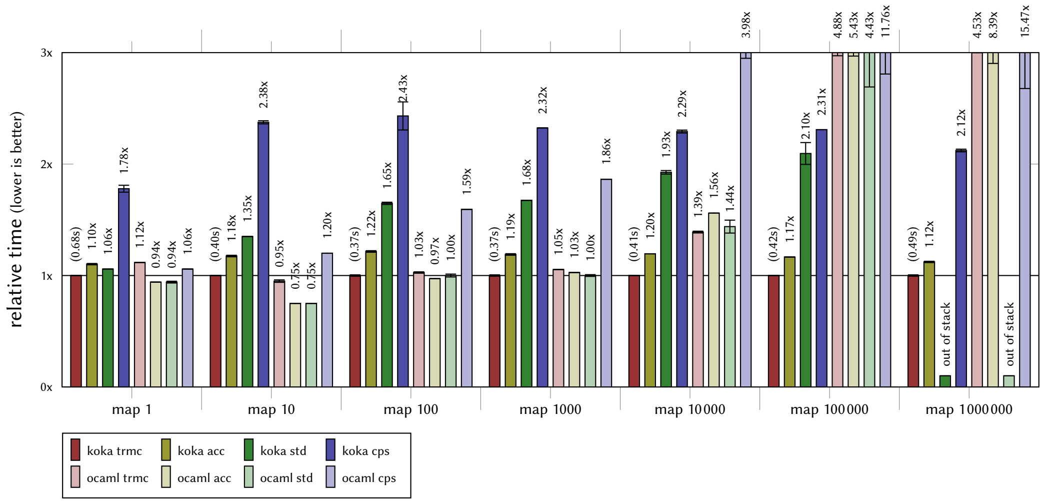


Fig. 5. Benchmarks on Ubuntu 20.04 (AMD 5950x), Koka v2.4.1-dev, OCaml 4.14.0. The benchmark repeatedly maps the increment function over a list of a given size and sums the result list. Each workload is scaled to process the same number of total elements (100 000 000). The tested variants of `map` are TRMC (trmc), accumulator style (acc), the standard non tail recursive style (std), and finally CPS style (cps).

1 Further Benchmarks

Figure 5 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.

2 Proofs

2.1 Context Laws for Defunctionalized Contexts

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

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

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

For application we have:

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

□

2.2 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:

$$
\begin{array}{lll}
 & \mathsf{app}\;(\mathsf{ctx}\,\mathsf{A})\;e & \\
= & \mathsf{app}\;\llparenthesis \mathsf{A} \rrparenthesis\;e & \{\;(\mathit{rctx})\;\} \\
= & e \odot \llparenthesis \mathsf{A} \rrparenthesis & \{\;(\mathit{rapp})\;\}
\end{array}
$$

We proceed by induction over A.

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

$$
\begin{array}{lll}
= & e \odot \llparenthesis \square \rrparenthesis & \\
= & e \odot \mathit{unit} & \{\;\mathit{fold}\;\} \\
= & e & \{\;\mathit{unit}\;\} \\
= & \square[e] & \{\;\square\;\}
\end{array}
$$

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

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

2.3 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

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

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

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

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

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

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

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

□

2.4 Context Laws for Exponent Contexts

We prove the composition law by induction on k_2:

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

and

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

Appliction can be derived as:

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

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

$$\begin{array}{lll}
= & \mathsf{app}\,(\!|\square|\!)\,e & \\
= & \mathsf{app}\,0\,e & \{\ \mathit{fold}\ \} \\
= & e & \{\ (\mathit{xapp})\ \} \\
= & \square[e] & \{\ \square\ \}
\end{array}$$

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

$$\begin{array}{lll}
= & \mathsf{app}\,(\!|g\,\mathsf{A}'|\!)\,e & \\
= & \mathsf{app}\,((\!|\mathsf{A}'|\!) + 1)\,e & \{\ \mathit{fold}\ \} \\
= & \mathsf{app}\,(\!|\mathsf{A}'|\!)\,(g\,e) & \{\ (\mathit{xapp})\ \} \\
= & \mathsf{A}'[g\,e] & \{\ \mathit{induction\,hyp.}\ \} \\
= & \mathsf{A}[e] & \{\ \mathsf{A}\,\mathit{context}\ \}
\end{array}$$

□

2.5 Constructor Contexts

Composition:

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

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

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

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

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

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

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

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

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

Fig. 6. Minamide's type system adapted to our language

and application:

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

□

2.6 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 3. (*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 $\Gamma|_{\mathsf{N}}, f\,;\, \varnothing \vdash_{\mathsf{M}} \mathsf{fun} f' \,=\, \lambda xs.\, \lambda k.\, [\![e]\!]_{f,k} \,:\, \tau_1 \to \ldots \to \tau_n \to ((\tau,\ \tau)\ \mathit{hfun}) \to \tau.$

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

Lemma 4. (*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 $\Box$.

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

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

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

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 3*) By the FUNDECL and ABS rules we obtain:

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

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

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

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

Case (*base*).

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

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

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

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

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

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

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

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

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

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

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

□

2.7 Contexts Form Linear Chains

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

Case $C \ldots \Box_i \ldots$:

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

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

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

$\Box$

2.8 Deriving Constructor Context Fold

Given the specification:

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

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

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

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

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

2.9 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[\square]\;\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 \square_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 \square_i \ldots]^1_{x_1},\ [H_2, y_2 \mapsto^1 C_2 \ldots \square_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{lemm} \\
= & H,\ [H_1, y_1 \mapsto^1 C_1 \ldots \square_i \ldots]^1_{x_1},\ [H_2, y_2 \mapsto^1 C_2 \ldots \square_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 \square_i \ldots]^1_{x_1},\ [H_2, y_2 \mapsto^1 C_2 \ldots \square_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 \square_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 \square_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 \square_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 \square_i \ldots]^1_{x_1},\ [H_2, y_2 \mapsto^1 C_2 \ldots \square_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 \square_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\rangle\; e & \{\, (3) \,\} \\
\cong & H,\ [H_1, y_1 \mapsto^1 C_1 \ldots \square_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:

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

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 $k \bullet \mathsf{ctx}\,\Box$).

2.10 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 $[H', y \mapsto^m C \ldots \square_i \ldots]^n_x \mid \langle x, y@i \rangle$ if we have $H_0 \mid (\!|\mathsf{K}|\!) \cong H_0,\ [H', y \mapsto^m C \ldots \square_i \ldots]^1_x \mid \langle x, y@i \rangle$.

Application:

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

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

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

and

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