Compositional Deep Argument Flattening

MARTIN ELSMAN, University of Copenhagen, Denmark

We present a mechanism for unboxing function arguments in a way that allows nested objects and curried arguments to be passed to functions flattened and unboxed in registers. The mechanism supports that the transformation is performed in multiple passes, perhaps interleaved by other optimisation passes that promote new opportunities for argument unboxing, flattening, and uncurrying. Moreover, the technique fits well within a framework for incremental recompilation where transformations may be applied to functions across compilation unit boundaries.

We report on key properties of the technique, including a type preservation property and compositional properties. We also report on an implementation and on the performance benefits of the approach.

1 INTRODUCTION

Unboxing of function arguments is paramount for achieving high performance, in particular in the context of compilers that implement support for polymorphic functions using a uniform representation of values. Various techniques may be used for unboxing and uncurrying and it is important that functions may be called using efficient calling conventions also when the calls pass compilation unit boundaries. At the same time, it must hold that a transformed program does not perform worse than the non-transformed program, in particular with respect to memory usage. Such a guarantee can be provided if it is ensured that transformed call sites will not lead to additional allocations that were not previously performed by the non-transformed function.

In this paper, we present a mechanism for unboxing function arguments in a way that allows nested objects and curried arguments to be passed to functions flattened and unboxed in registers. The mechanism supports that the transformation is performed in multiple passes, perhaps interleaved by other optimisation passes that promote new opportunities for argument unboxing, flattening, and uncurrying. More importantly, the technique is designed so that unboxing decisions for a function are made based only on analysing the function definition (i.e., the function body) and not its non-recursive uses. Such a design turns out to fit well within a framework for incremental recompilation where transformations may be applied to functions across compilation unit boundaries [7], which is in contrast to how similar optimisations are made in compilers that assume whole-program compilation [3, 20, 21]. The technique is based on the concept of so-called compositional argument transformers, which serve to adjust function types and call sites and which are compositions of atomic transformations that each eliminates an argument, adds a new unboxed argument, or uncurries the function.

The mechanism is concerned only with the optimisation of calling conventions for so-called known functions, that is, functions that are defined using letrec-definitions and which (by convention) are always fully applied. Unknown-functions use a uniform representation of arguments and an optimiser will do its best for eliminating unknown functions at compile time, using standard optimisation techniques such as inlining (compile-time beta-reductions) and function specialisation [11, 17]. This paper is concerned with optimisations that cannot be performed with these standard optimisation techniques, including argument flattening and uncurrying of functions, which also directly eliminates a certain class of unknown functions.

The contributions of this work are as follows:

- We present a mechanism for unboxing function arguments in a way that allows nested objects and curried arguments to be passed to functions flattened and unboxed in registers.

ML Family Workshop 2025. October 16, 2025. Singapore.
Author's address: Martin Elsman, Department of Computer Science, University of Copenhagen, Universitetsparken 5, Copenhagen, Denmark, DK-2100, mael@di.ku.dk.

- We demonstrate key properties of the technique, including a type preservation property and compositional properties.
- We have implemented the technique in the MLKit, a Standard ML compiler, which uses a smart-recompilation management system for achieving fast rebuilds upon changes of source code. We describe the implementation and outline how the technique is integrated with other optimisations.
- We give an overview of the performance benefits of the technique based on a series of benchmarks and discuss how the technique can be extended to allow for more unboxing.

The extended abstract is structured as follows. In Section 2, we present examples that benefit from compositional deep argument flattening. In Section 3, we present a minimal polymorphic typed functional language for which we provide a type system. In Section 4, we present the argument flattening technique and demonstrate the key properties of the technique.

In Section 7, we present a series of benchmarks and demonstrate the effectiveness of the technique in terms of performance improvements observed when enabling the various features of the technique. In Section 8, we present related work. In Section 9, we conclude and discuss future work.

2 EXAMPLES OF DEEP ARGUMENT FLATTENING

Good examples of functions that benefit from deep argument flattening are functions that operate on 2-dimensional real-vectors, including an `add` function on such vectors and the dot-product function `dotp` over such vectors. In Standard ML, here are some definitions:

```
type v2 = real * real
val zero : v2 = (0.0, 0.0)
fun add (x1,y1) (x2,y2) : v2 = (x1+x2,y1+y2)
fun dotp (x1,y1) (x2,y2) : real = x1*x2+y1*y2
```

Conventionally, calling conventions for the functions `add` and `dotp` can be obtained from the types of the functions:

```
val add  : v2 → v2 → v2
val dotp : v2 → v2 → real
```

Without knowing more about the functions, their types suggest that, given `v2`-values as arguments, each of the functions should return functions that again will accept `v2`-values as arguments. Due to the possibility of writing polymorphic functions that will extract elements from arbitrary triples, it is standard to represent values of type `v2` as boxed tuples containing references to boxed floating-point values. That is, without knowing more about a function than its type, due to polymorphism, all values are required to be represented using one word (in memory or in a register). Letting `add` and `dotp` return closures that are immediately called with their second argument is expensive and most ML implementations therefore make use of inlining strategies or whole-program compilation techniques to mitigate the problematic overhead. An alternative strategy would be to recognise that some functions may be represented uncurried. For instance, the `add` function could be implemented with the type `v2*v2→v2`. The reason this representation is possible is that applying `add` to one argument has no effect other than returning a closure. Moreover, because the `add` function is not using the `v2` values for anything else than using their components (e.g., none of the `v2` values are stored in data structures), it would be safe to pass the elements of the `v2` values unboxed, which leads to `add` having the internal type `⟨real,real,real,real⟩→v2`, where the function now takes multiple parameters. Finally, because each of the passed boxed real values are only destructed by the `add` function, we can pass the arguments unboxed in floating-point registers (or on the stack, depending on the calling convention for the specific architecture), as reflected in `add` having the

internal representation type ⟨f64,f64,f64,f64⟩→v2, where f64 is the type of unboxed floating point values, which are not allowed to be instantiated for type variables in calls to polymorphic functions.

Notice that partial applications of add and dotp may be safely translated into explicit closures. Thus map (add zero) [zero,zero] will be compiled into the intermediate-level code map (**fn** x ⇒ add ⟨0.0,0.0,f64(#1 x),f64(#2 x)⟩) [zero,zero], while guaranteeing that code execution is not duplicated at runtime. The deep argument flattening technique that we present assumes that decisions on how a function is represented (its calling convention) is purely a property of the function itself and not of its call sites. We further assume that all call sites may be updated using the notion of compositional argument transformers. Here is a compositional argument transformer that turns out to be a valid compositional argument transformer for both the add and dotp functions and which describes how call-sites to these functions should be updated to safely represent calls to the functions:

$$\begin{aligned}\mu \quad = \quad & \texttt{drop 1} \circ \texttt{drop 1} \circ \texttt{drop 1} \circ \texttt{drop 1} \circ \texttt{f64 4} \circ \texttt{f64 3} \circ \texttt{f64 2} \circ \texttt{f64 1} \\ & \circ \texttt{drop 1} \circ \texttt{drop 1} \\ & \circ \texttt{addprj}(2,2) \circ \texttt{addprj}(2,1) \circ \texttt{addprj}(1,2) \circ \texttt{addprj}(1,1) \\ & \circ \texttt{uncurry}\end{aligned}$$

The argument transformer μ is composed of a series of atomic transformers (to be read from the right), which can be used to transform both types and call sites. For transforming the type v2→v2→real of dotp, first it is uncurried, yielding the type ⟨v2,v2⟩→real. Then four unboxing transformations happen, followed by two drop transformations, which yields the type ⟨real,real,real,real⟩→real. Finally, four float-unboxing transformations add unboxed float arguments corresponding to the boxed real arguments, followed by four drop transformations, which yield the final type ⟨f64,f64,f64,f64⟩→real.

Notice that the present work are concerned only with arguments and not with return values. We envision that the transformation mechanism that we present here can be used also to return values unboxed in registers and on the stack.

3 A MINIMAL LANGUAGE

We present a simple polymorphically-typed functional language. We give a type system for the system compatible with a standard small-step contextual semantics (which we do not present here). The language features recursive functions and tuples. The language does not feature sum types, for which unboxing can be considered as an orthogonal problem [9], and for simplicity, we leave out the formalisation of unboxing floating-point arguments, which can be modeled as unboxing singleton tuples.

We assume a denumerable infinite set of *type variables*, ranged over by α. Whenever $o_1, \cdots, o_n$ is some sequence of objects, we use the notation $\vec{o}^{(n)}$ to denote this sequence and we shall sometimes just write $\vec{o}$ if the length of the sequence is either non-restricted or is implicitly determined from the context. We define *types* (τ) and *type schemes* (σ) according to the following grammar:

$$\begin{aligned}\tau \quad &::= \quad \alpha \mid \texttt{int} \mid \langle \tau, \cdots, \tau \rangle \to \tau \mid \tau \to \tau \mid \tau \times \cdots \times \tau \\ \sigma \quad &::= \quad \forall \vec{\alpha}.\tau\end{aligned}$$

Notice that we distinguish between function types with multiple arguments and function types with a single argument. For type schemes $\forall\vec{\alpha}.\tau$, we consider $\vec{\alpha}$ bound in τ and we consider type schemes identical up to renaming of bound variables. We use S to range over *type substitutions*, which map type variables to types. When A is some object, we write $S(A)$ to denote the object A

with each free occurrences of a type variables $\alpha \in \mathrm{Dom}\ A$ replaced with $S(\alpha)$. When $\sigma = \forall\vec{\alpha}.\tau'$ is some type scheme and τ is some type, we write $\sigma \geq \tau$ if there exists a substitution S such that $\mathrm{Dom}\ S = \{\vec{\alpha}\}$ and $S(\tau') = \tau$.

For defining the grammar of values and expressions, we assume a denumerable infinite set of *program variables*, ranged over by x and f, and we use d to ranger over integer constants. We define *values* (v), *access expressions* (a), *patterns* (p and q), and *expressions* (e) as follows:

$$\begin{array}{lcl} v & ::= & d \mid \lambda x:\tau.e \mid (v,\cdots,v) \mid \mathtt{fix}\ f:\sigma\ \langle\vec{p}\rangle = e \\ a & ::= & x \mid \#i\ a \mid v \\ p & ::= & x:\tau \\ e & ::= & a \mid e\ e \mid f\ \langle a,\cdots,a\rangle \mid (e,\cdots,e) \mid \mathtt{fun}\ f:\sigma\ \langle\vec{p}\rangle = e\ \mathtt{in}\ e \end{array}$$

Notice that ordinary lambda abstractions are distinguished from recursive function definitions, both for values and expressions. For reasons that will become clear later, we separate access expressions from ordinary expressions. When a is some access expression denoting a tuple, the access expression $\#i\ a$, where i is some positive integer, projects the $i'th$ element of the tuple (assuming the number of elements in the tuple is larger than i).

For values of the form $\lambda x : \tau.e$, we consider x bound in e. Further, for values of the form $\mathtt{fix}\ f : \forall\vec{\alpha}.\tau\ \langle\vec{p}\rangle = e$ and for expressions of the form $\mathtt{fun}\ f : \forall\vec{\alpha}.\tau\ \langle\vec{p}\rangle = e\ \mathtt{in}\ e'$, we consider f bound in e and in e' and we consider program variables in $\vec{p}$ bound in e. Moreover, we consider $\vec{\alpha}$ bound in τ, $\vec{p}$, and e. Values and expressions are considered identical up to renaming of bound type variables and bound program variables. As is standard, we write $e[v/x]$ for the substitution of the value v for the variable x in the expression e. When $\vec{p} = x_1 : \tau_1, \cdots, x_n : \tau_n$ is some pattern sequence, we write $\mathrm{types}(\vec{p})$ to denote the type sequence $\tau_1, \cdots, \tau_n$.

We sometimes write $\mathtt{let}\ x : \tau = e_1\ \mathtt{in}\ e_2$ as an abbreviation for $(\lambda x : \tau.e_2)\ e_1$. We also sometimes write $f\ \langle e_1, \cdots, e_n\rangle$ as an abbreviation for $\mathtt{let}\ x_1 : \tau_1 = e_1\ \mathtt{in}\ \cdots \mathtt{let}\ x_n : \tau_n = e_n\ \mathtt{in}\ f\ \langle x_1, \cdots, x_n\rangle$ and we write $\#i\ e$ as an abbreviation for $\mathtt{let}\ x : \tau = e\ \mathtt{in}\ \#i\ x$, with suitable fresh variables $x, x_1, \cdots, x_n$.

A *type environment* (Γ) maps program variables to type schemes and when Γ and Γ' are type environments with disjoint domains, we write Γ, Γ' to denote map composition. Moreover, we write $x : \sigma$ to denote a singleton type environment mapping x to the type scheme σ and we write $\cdot$ to denote the empty type environment.

3.1 Type System

We define a type system for the language, which is based on a standard Hindley-Milner type system with polymorphic types. The type system, which is presented in Figure 1, is defined in terms of a set of inference rules that allow inferences of sentences of the form $\vdash v : \sigma$, which are read, the value v has type scheme σ, and of the form $\Gamma \vdash e : \sigma$, which are read, the expression e has type scheme σ in the type environment Γ.

The following two propositions hold, which are easily shown by induction on the respective typing derivations:

Proposition 3.1 (Typing Closed Under Value Substitution). *If* $\Gamma, x : \sigma' \vdash e : \sigma$ *and* $\vdash v : \sigma'$ *then* $\Gamma \vdash e[v/x] : \sigma$.

Proposition 3.2 (Typing Closed Under Type Substitution). *If* $\Gamma \vdash e : \sigma$ *then* $S(\Gamma) \vdash S(e) : S(\sigma)$, *for any type substitution* S.

Moreover, we shall be using the following additional properties for the typing judgment:

Proposition 3.3 (Typing Depends Only on Free Variables). *If* $\Gamma, x : \sigma' \vdash e : \sigma$ *and* $x \notin \mathrm{fv}\ e$ *then* $\Gamma \vdash e : \sigma$.

Value Typing $\boxed{\vdash v : \sigma}$

$$\frac{}{\vdash d : \mathtt{int}}\ [\textsc{tv-int}] \qquad \frac{x : \tau_1 \vdash e : \tau_2}{\vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2}\ [\textsc{tv-lam}]$$

$$\frac{\vdash v_i : \tau_i, \quad i = 1, \cdots, n}{\vdash (v_1, \cdots, v_n) : \tau_1 \times \cdots \times \tau_n}\ [\textsc{tv-tup}] \qquad \frac{\sigma = \forall \vec{\alpha}.\langle \vec{\tau} \rangle \to \tau \qquad \text{types } p = \vec{\tau} \qquad f : \langle \vec{\tau} \rangle \to \tau, \vec{p} \vdash e : \tau}{\vdash \mathtt{fix}\ f : \sigma\ \langle \vec{p} \rangle = e : \sigma}\ [\textsc{tv-fix}]$$

Expression Typing $\boxed{\Gamma \vdash e : \sigma}$

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma}\ [\textsc{t-var}] \qquad \frac{i \in \{1, \cdots, n\} \qquad \Gamma \vdash a : \tau_1 \times \cdots \times \tau_n}{\Gamma \vdash \#i\ a : \tau_i}\ [\textsc{t-sel}] \qquad \frac{\vdash v : \sigma}{\Gamma \vdash v : \sigma}\ [\textsc{t-val}]$$

$$\frac{\Gamma \vdash e : \sigma \quad \sigma \geq \tau}{\Gamma \vdash e : \tau}\ [\textsc{t-sub}] \qquad \frac{\Gamma \vdash e_i : \tau_i, \quad i = 1, \cdots, n}{\Gamma \vdash (e_1, \cdots, e_n) : \tau_1 \times \cdots \times \tau_n}\ [\textsc{t-tup}]$$

$$\frac{\Gamma \vdash e : \langle \vec{\tau}^{(n)} \rangle \to \tau \quad \Gamma \vdash a_i : \tau_i, \quad i = 1, \cdots, n}{\Gamma \vdash e\ \langle a_1, \cdots, a_n \rangle : \tau}\ [\textsc{t-fapp}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ [\textsc{t-app}] \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2}\ [\textsc{t-lam}]$$

$$\frac{\sigma = \forall \vec{\alpha}.\langle \vec{\tau} \rangle \to \tau \quad \text{types } p = \vec{\tau} \quad \{\vec{\alpha}\} \cap \mathrm{ftv}(\Gamma, \sigma') = \emptyset \qquad \Gamma, f : \langle \vec{\tau} \rangle \to \tau, \vec{p} \vdash e_1 : \tau \quad \Gamma, f : \sigma \vdash e_2 : \sigma'}{\Gamma \vdash \mathtt{fun}\ f : \sigma\ \langle \vec{p} \rangle = e_1\ \mathtt{in}\ e_2 : \sigma'}\ [\textsc{t-fun}]$$

Fig. 1. Type system for the minimal language.

Proposition 3.4 (Typing Closed Under Access Expression Substitutions). *If* $\Gamma \vdash e : \sigma$ *and* $\Gamma \vdash a : \tau$ *and* $\Gamma(x) = \tau$ *then* $\Gamma \vdash e[x/e] : \sigma$.

Proposition 3.5 (Typing Closed Under Extended Environments). *If* $\Gamma \vdash e : \sigma$ *and* $\mathrm{Dom}\ \Gamma \cap \mathrm{Dom}\ \Gamma_0 = \emptyset$ *then* $\Gamma, \Gamma_0 \vdash e : \sigma$.

4 ARGUMENT FLATTENING

We now formalise the notion of deep argument flattening, which, for simplicity, is defined only for expressions that are *fix-free*, that is, for expressions not containing `fix` constructs.

We first introduce the notion of *compositional argument transformers*, which are compositions of atomic transformers that aim at eliminating an argument, adding a new unboxed argument, or uncurrying a function. In the following, we write $\mathtt{prj}(i, \tau_1 \times \cdots \times \tau_n)$ to mean τ_i, provided $i \in \{1, \cdots, n\}$. The grammar for compositional argument transformers is as follows:

$$\mu \quad ::= \quad \mathtt{drop}(i) \mid \mathtt{addprj}(i, j) \mid \mathtt{uncurry} \mid \mu \circ \mu \mid \mathtt{id}$$

Call-site transformation $\boxed{\mu \vdash_{\text{call}} f\ \langle \vec{a} \rangle : \tau \Rightarrow f\ \langle \vec{a}' \rangle : \tau', \Gamma}$

$$\frac{\vec{a}' = a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n}{\texttt{drop}(i) \vdash_{\text{call}} f\ \langle \vec{a}^{(n)} \rangle : \tau \Rightarrow f\ \langle \vec{a}' \rangle : \tau, \cdot}\ [\textsc{c-drop}]$$

$$\frac{}{\texttt{addprj}(i,j) \vdash_{\text{call}} f\ \langle \vec{a}^{(n)} \rangle : \tau \Rightarrow f\ \langle \vec{a}, \#j\ a_i \rangle : \tau, \cdot}\ [\textsc{c-add}]$$

$$\frac{x \notin \text{fv}(f\ \langle \vec{a} \rangle)}{\texttt{uncurry} \vdash_{\text{call}} f\ \langle \vec{a} \rangle : \tau \to \tau' \Rightarrow f\ \langle \vec{a}, x \rangle : \tau', x : \tau}\ [\textsc{c-unc}]$$

$$\frac{\mu_1 \vdash_{\text{call}} f\ \langle \vec{a} \rangle : \tau \Rightarrow f\ \langle \vec{a}'' \rangle : \tau'', \Gamma_1 \quad \mu_2 \vdash_{\text{call}} f\ \langle \vec{a}'' \rangle : \tau'' \Rightarrow f\ \langle \vec{a}' \rangle : \tau', \Gamma_2}{\mu_2 \circ \mu_1 \vdash_{\text{call}} f\ \langle \vec{a} \rangle : \tau \Rightarrow f\ \langle \vec{a}' \rangle : \tau', \Gamma_1, \Gamma_2}\ [\textsc{c-comp}] \qquad \frac{}{\texttt{id} \vdash_{\text{call}} f\ \langle \vec{a} \rangle : \tau \Rightarrow f\ \langle \vec{a} \rangle : \tau, \cdot}\ [\textsc{c-id}]$$

Fig. 2. Transforming call sites. The rules specify how a call site is transformed given a compositional argument transformer μ.

Applying a compositional argument transformer μ to a type τ, written $\mu(\tau)$, yields a new type, provided the transformer matches the type:

$$\begin{array}{rcl} \texttt{drop}(i)(\langle \vec{\tau}^{(n)} \rangle \to \tau) & = & \langle \tau_1, \cdots, \tau_{i-1}, \tau_{i+1}, \cdots, \tau_n \rangle \to \tau \\ \texttt{addprj}(i,j)(\langle \vec{\tau}^{(n)} \rangle \to \tau) & = & \langle \vec{\tau}, \texttt{prj}(j, \tau_i) \rangle \to \tau \\ \texttt{uncurry}(\langle \vec{\tau} \rangle \to (\tau \to \tau')) & = & \langle \vec{\tau}, \tau \rangle \to \tau' \\ (\mu_2 \circ \mu_1)(\tau) & = & \mu_2(\mu_1(\tau)) \\ \texttt{id}(\tau) & = & \tau \end{array}$$

The result of applying a compositional argument transformer μ to a type scheme $\sigma = \forall \vec{\alpha}.\tau$, written $\mu(\sigma)$, is the type scheme $\forall \vec{\alpha}.\mu(\tau)$.

For specifying the effect of applying a compositional argument transformer to a call site $f\ \langle \vec{a} \rangle$, we define the relation $\mu \vdash_{\text{call}} f\ \langle \vec{a} \rangle : \tau \Rightarrow f\ \langle \vec{a}' \rangle : \tau'$, which states that applying the transformer μ to the call site $f\ \langle \vec{a} \rangle$ yields a new call site expression $f\ \langle \vec{a}' \rangle$ of type τ'. The relation is defined by the inference rules in Figure 2. Rule C-DROP expresses that argument i is dropped from the argument sequence. Rule D-ADD expresses that an a additional argument is added to the argument sequence. Rule C-UNC expresses uncurrying. Rules C-COMP and C-ID express how call sites are transformed by the composition of argument transformers and the identity argument transformer, respectively.

Whenever a type environment Γ binds variables only to monomorphic types, we use the notation $\Gamma \to \tau$ to denote the type of a function that takes arguments of types as determined by Γ and returns a value of type τ:

$$\begin{array}{rcl} (\cdot) \to \tau & = & \tau \\ (\Gamma, x : \tau') \to \tau & = & \Gamma \to (\tau' \to \tau) \end{array}$$

The following property holds:

Proposition 4.1 (Preservation of Call Site Types). *If* $\mu \vdash_{\text{call}} f\ \langle \vec{a} \rangle : \tau \Rightarrow e' : \tau', \Gamma$ *then* $\tau = \Gamma \to \tau'$.

Example 4.2. Consider a function `f` of type $\tau = \langle \texttt{int} \times \texttt{int} \rangle \to \texttt{int} \to \texttt{int}$, which takes a pair of integers as argument and returns a function that takes an integer as argument and returns an integer. Now, assume that, by analysing the body of `f`, it is recognised that its argument is deconstructed at all uses and that the function immediately returns a function of type $\texttt{int} \to \texttt{int}$. In that case, we can assume that the function can be transformed to take instead all three integers

Optimisations $\boxed{\mu \vdash_{\text{opt}} (\tau, \vec{p}, e) \Rightarrow (\tau', \vec{q}, e')}$

$$\frac{p_i = x_i : \tau_i \quad x_i \notin \text{fv}(e) \quad \text{types}(\vec{p}) = \vec{\tau} \quad \text{types}(\vec{q}) = \vec{\tau}' \quad \vec{p} = p_1, \cdots, p_n \quad \vec{q} = p_1, \cdots, p_{i-1}, p_{i+1}, \cdots, p_{n-1}}{\texttt{drop}(i) \vdash_{\text{opt}} (\langle \vec{\tau} \rangle \to \tau, \vec{p}, e) \Rightarrow (\langle \vec{\tau}' \rangle \to \tau, \vec{q}, e)} \text{ [O-DROP]}$$

$$\frac{p_i = x_i : \tau'_1 \times \cdots \times \tau'_m \quad 0 \leq j < m \quad \text{types}(\vec{p}) = \vec{\tau}}{\texttt{addprj}(i, j) \vdash_{\text{opt}} (\langle \vec{\tau} \rangle \to \tau, \vec{p}, e) \Rightarrow (\langle \vec{\tau}, \tau'_j \rangle \to \tau, (\vec{p}, x : \tau'_j), e[x/\#j\ x_i])} \text{ [O-ADD]}$$

$$\frac{\text{types}(\vec{p}) = \vec{\tau}}{\texttt{uncurry} \vdash_{\text{opt}} (\langle \vec{\tau} \rangle \to \tau \to \tau', \vec{p}, \lambda x : \tau.e) \Rightarrow (\langle \vec{\tau}, \tau \rangle \to \tau', (\vec{p}, x : \tau), e)} \text{ [O-UNC]}$$

$$\frac{\mu_1 \vdash_{\text{opt}} (\tau, \vec{p}, e) \Rightarrow (\tau'', \vec{r}, e'') \quad \mu_2 \vdash_{\text{opt}} (\tau'', \vec{r}, e'') \Rightarrow (\tau', \vec{q}, e')}{\mu_2 \circ \mu_1 \vdash_{\text{opt}} (\sigma, \vec{p}, e) \Rightarrow (\tau', \vec{q}, e')} \text{ [O-COMP]}$$

$$\frac{}{\texttt{id} \vdash_{\text{opt}} (\tau, \vec{p}, e) \Rightarrow (\tau, \vec{p}, e)} \text{ [O-ID]}$$

Fig. 3. The optimisation judgment. The rules specify the conditions under which an argument transformer μ is a valid transformer for transforming a function of type τ, with function parameters $\vec{p}$, and function body e into a refined type τ', refined function parameters $\vec{q}$, and a refined function body e'.

unboxed in registers, which is enforced by associating the function with the argument transformer $\mu = \texttt{uncurry} \circ \texttt{drop(1)} \circ \texttt{addprj(1, 2)} \circ \texttt{addprj(1, 1)}$. Applying the argument transformer to the function type τ yields $\mu(\tau) = \langle \texttt{int}, \texttt{int}, \texttt{int} \rangle \to \texttt{int}$. Moreover, based on the call site transformation $\mu \vdash f\ \langle y \rangle : \texttt{int} \to \texttt{int} \Rightarrow f\langle \#1\ y, \#2\ y, x \rangle, x : \texttt{int}$, we can envision that each call site $f\ \langle y \rangle$ may be transformed into the expression $\lambda x : \texttt{int}.f\langle \#1\ y, \#2\ y, x \rangle$, which may lead to compile-time β-reductions and static tuple projections depending on the call-site contexts.

Argument flattening makes use of a so-called optimisation judgment for identifying function transformations, transforming parameters, and defining a so-called access map. The optimisation judgment takes the form $\mu \vdash_{\text{opt}} (\tau, \vec{p}, e) \Rightarrow (\tau', \vec{q}, e')$, where μ is an argument transformer, the left-hand triple $(\tau, \vec{p}, e)$ specifies the type of the function, the parameter sequence, and the function body. The the right-hand triple $(\tau', \vec{q}, e')$ specifies the type of the function, its parameter sequence, and a parameter-transformed function body.

The rules defining the optimisation judgment are given in Figure 3. Rule O-DROP specifies that an argument may be dropped if the argument parameter is not used in the body of the function. Rule O-ADD suggests the addition of a new function parameter corresponding to a particular projection of an argument tuple. Rule O-UNCURRY specifies uncurrying of a function body that consists directly of an immediate lambda-abstraction. Rules O-ID and O-COMP specify the identity transformation and how transformations are composed, respectively. The rules specify, through the parameter sequences $\vec{p}$ and $\vec{q}$, how optimisations modify parameters.

In the following, we use the notation $\lambda\Gamma.e$ to denote a function that takes arguments according to Γ and returns the value resulting from evaluating e:

$$\begin{aligned} \lambda(\cdot).e &= e \\ \lambda(\Gamma, x : \tau').e &= \lambda\Gamma.\lambda x : \tau'.e \end{aligned}$$

The following property holds and is proven by induction over the derivation of the optimisation judgment.

Flattening $\boxed{\phi \vdash e \Rightarrow e' : \tau}$

$$\frac{\phi(x) = (\tau, id)}{\phi \vdash x \Rightarrow x : \tau}\ [\textsc{f-var}] \qquad \frac{}{\phi \vdash d \Rightarrow d : \mathtt{int}}\ [\textsc{f-int}]$$

$$\frac{i \in \{1, .., n\} \quad \phi \vdash a \Rightarrow a' : \tau_1 \times \cdots \times \tau_n}{\phi \vdash \#i\ a \Rightarrow \#i\ a' : \tau_i}\ [\textsc{f-sel}] \qquad \frac{\phi, x : (\tau, id) \vdash e \Rightarrow e' : \tau'}{\phi \vdash \lambda x : \tau.e \Rightarrow \lambda x : \tau.e' : \tau \to \tau'}\ [\textsc{f-lam}]$$

$$\frac{\phi(f) = (\sigma, \mu) \quad \sigma \geq \langle \vec{\tau} \rangle \to \tau \quad \mu \vdash_{\text{call}} f\ \langle \vec{a} \rangle : \tau \Rightarrow f\ \langle \vec{a}' \rangle : \tau', \Gamma}{\phi \vdash f\ \langle \vec{a} \rangle \Rightarrow \lambda\Gamma.f\ \langle \vec{a}' \rangle : \Gamma \to \tau'}\ [\textsc{f-fapp}] \qquad \frac{\phi \vdash e_1 \Rightarrow e_1' : \tau_1 \to \tau_2 \quad \phi \vdash e_2 \Rightarrow e_2' : \tau_1}{\phi \vdash e_1\ e_2 \Rightarrow e_1'\ e_2' : \tau_2}\ [\textsc{f-app}]$$

$$\frac{\phi \vdash e_i \Rightarrow e_i' : \tau_i, \quad i = 1, \cdots, n}{\phi \vdash (e_1, \cdots, e_n) \Rightarrow (e_1', \cdots, e_n') : \tau_1 \times \cdots \times \tau_n}\ [\textsc{f-tup}]$$

$$\frac{\tau = \langle \vec{\tau} \rangle \to \Gamma_0 \to \tau_0 \quad \mu \vdash_{\text{opt}} (\tau, \vec{p}, \lambda\Gamma_0.e_1) \Rightarrow (\tau', \vec{q}, e_1') \quad \phi, f : (\tau, \mu), \vec{q} : \mathtt{id} \vdash e_1' \Rightarrow e_1'' : \tau_0 \quad \phi, f : (\forall\vec{\alpha}.\tau, \mu) \vdash e_2 \Rightarrow e_2' : \tau''}{\phi \vdash \mathtt{fun}\ f : \forall\vec{\alpha}.\tau\ \langle \vec{p} \rangle = \lambda\Gamma_0.e_1\ \mathtt{in}\ e_2 \Rightarrow \mathtt{fun}\ f : \forall\vec{\alpha}.\tau'\ \langle \vec{q} \rangle = e_1''\ \mathtt{in}\ e_2' : \tau''}\ [\textsc{f-fun}]$$

Fig. 4. Deep argument flattening. The rules apply to fix-free expressions for which references to `fun`-bound functions appear only in direct application contexts of the form $f\ \langle \vec{a} \rangle$. The rules express how, given a flattening environment ϕ, an expression e is transformed into another expression e' of type τ.

Proposition 4.3 (Type Argument Transformation). *Assume* $\tau = \langle \vec{\tau} \rangle \to \Gamma_0 \to \tau_0$ *and* types $\vec{p} = \vec{\tau}$ *and* $\Gamma, \vec{p}, \Gamma_0 \vdash e : \tau_0$. *If* $\mu \vdash_{\text{opt}} (\tau, \vec{p}, \lambda\Gamma_0.e) \Rightarrow (\tau', \vec{q}, e')$ *then* $\mu(\tau) = \tau'$ *and* $\tau' = \langle \vec{\tau}' \rangle \to \tau_0$, *where* $\vec{\tau}' =$ types $\vec{q}$ *and* $\Gamma, \vec{q} \vdash e' : \tau_0$. *Moreover, if* e *is on the form* $\lambda\Gamma_1.e_1$ *for some* e_1, *then* e' *if of the form* $\lambda\Gamma_1.e_1'$, *for some* e_1'.

The last property is necessary for the proposition to be sufficiently strong that it can be shown by induction. Details are provided in Appendix A.

Argument flattening proper is formalised as a set of inference rules that allow inference of sentences of the form $\phi \vdash e \Rightarrow e' : \tau$, where e and e' are expressions, τ is a type, and ϕ is a *flattening environment* mapping program variables to pairs (σ, μ) of a type scheme and a compositional argument transformer. As an abbreviation, when Γ is some type environment $x_1 : \tau_1, \cdots, x_n : \tau_n$, we write $\Gamma : \mathtt{id}$ to denote the flattening environment $x_1 : (\sigma_1, \mathtt{id}), \cdots, x_n : (\sigma_n, \mathtt{id})$.

The inference rules for argument flattening are given in Figure 4. The rules apply to fix-free expressions for which references to `fun`-bound functions appear only in direct application contexts of the form $f\ \langle \vec{a} \rangle$, perhaps enforced by η-expansion. Most of the rules are straightforward inductive rules with the exception of rules F-ACC, F-FAPP and F-FUN. Rule F-ACC takes care of variables and of replacing tuple accesses, when possible. Rule F-FAPP specifies that a call to a known polymorphic function f may be transformed according to the argument transformer μ associated with the function f in the flattening environment ϕ. Rule F-FUN specifies how and under what conditions a polymorphic known (i.e., `fun`-bound) function f is transformed. Notice that the rule supports the transformation of both recursive and non-recursive call sites.

5 TYPE SOUNDNESS

We shall now set out to demonstrate that typing is closed under deep argument flattening. We first define a relation that relates variables in a source type environment Γ with variables in a target

Environment Modeling $\boxed{\phi \vdash \Gamma \sim \Gamma'}$

$$\frac{\phi \vdash \Gamma \sim \Gamma' \quad \mu(\sigma) = \sigma'}{\phi, x : (\sigma, \mu) \vdash \Gamma, x : \sigma \sim \Gamma', x : \sigma'}\ [\text{E-VAR}] \qquad \frac{}{\cdot \vdash \cdot \sim \cdot}\ [\text{E-EMP}]$$

Fig. 5. Relating environments.

environment Γ' through a flattening environment ϕ. The relation is written $\phi \vdash \Gamma \sim \Gamma'$ and is defined in Figure 5.

Before stating a property expressing that deep argument flattening is type-preserving, we first state a few auxiliary propositions. First, the following proposition states that the typing of access expressions under related environments is identical:

Proposition 5.1 (Relation of Access Expression Typings). *If* $\phi \vdash \Gamma \sim \Gamma'$ *and* $\Gamma \vdash a : \tau$ *then* $\Gamma' \vdash a : \tau$.

This property follows by the assumption that `fun`-bound variables are always fully applied and thus cannot appear in access expressions.

The following proposition relates the typing of a call expression with the typing of its translation:

Proposition 5.2 (Translation of Call Expressions Closed Under Typings). *Assume* $\sigma \geq \langle \vec{\tau} \rangle \to \tau$. *If* $\Gamma, f : \sigma \vdash f\ \langle \vec{a} \rangle : \tau$ *and* $\mu \vdash_{\text{call}} f\ \langle \vec{a} \rangle : \tau \Rightarrow f\ \langle \vec{a}' \rangle : \tau', \Gamma_0$ *and* $\sigma' = \mu(\sigma)$ *then* $\Gamma, f : \sigma', \Gamma_0 \vdash f\ \langle \vec{a}' \rangle : \tau'$ *and* $\tau = \Gamma_0 \to \tau'$.

The proposition is demonstrated by induction on the relation $\mu \vdash_{\text{call}} f\ \langle \vec{a} \rangle : \tau \Rightarrow f\ \langle \vec{a}' \rangle : \tau', \Gamma_0$. Details are given in Appendix A.

The following proposition states that deep argument flattening, as defined by the flattening relation, is type-preserving:

Proposition 5.3 (Typing Preserved Under Transformation). *Assume* $\phi \vdash \Gamma \sim \Gamma'$ *and* $\Gamma \vdash e : \tau$. *If* $\phi \vdash e \Rightarrow e' : \tau$ *then* $\Gamma' \vdash e' : \tau$.

The proposition is proven by straightforward induction and by use of Proposition 4.3 for the F-FUN case and Proposition 5.2 for the F-FAPP case.

Notice that Proposition 5.3 talks only about expressions e for which a transformation derivation exists, which assumes that e is fix-free and that all references to `fun`-bound functions are fully applied.

6 MULTIPLE OPTIMISATION PHASES

The process of deep argument flattening may benefit from traditional intermediate optimisation phases such as common sub-expression elimination, function specialisation, inlining, constant folding, dead-code elimination, and tuple-elimination. An implementation may interleave phases and compose argument transformers to create single transformers for functions.

Figure 6 lists a simplified version of the optimisation algorithm implemented in the MLKit. The main optimisation function takes as argument an *optimisation environment* (`E`), a flattening environment (`P`), and an expression (`e`), the compilation unit expression. The function returns a triple `(e',E',P')` of an (hopefully) optimised compilation unit expression, an optimisation environment, and a flattening environment. The optimisation environment `E'` and the flattening environment `P'` are used for compiling other compilation units that depends on the compiled compilation unit, for instance, to adjust call sites of exported functions using argument transformers. Each loop-function is iterated until there are no more code changes or until a maximum iteration limit

```
fun optimise (E:env) (P:phi) e =
  let fun loop1 N E e =
        let val () = reset_changes()
            val (e, E1) = contract E e
            val e = (cse o eliminate_explicit_records) e
        in if changes() andalso N > 0
           then loop1 (N-1) E e
           else (e, E1)
        end
      fun loop2 N E P e =
          let val () = reset_changes()
              val (e, P1) = flatten P e
              val (e, _) = contract E e
              val e = (cse o eliminate_explicit_records) e
          in if changes() andalso N > 0 then
               let val (e,P2) = loop2 (N-1) E
                                      (phi_nofun P) e
               in (e, phi_compose(P1,P2))
               end
             else (e, P1)
          end
      val (e, E') = loop1 MAX_ITER E e
      val (e, P') = loop2 MAX_ITER (unknown_env E) P e
  in (e,E',P')
  end
```

Fig. 6. A simplified version of the MLKit optimisation algorithm. Each expression rewrite forces a flag (a boolean reference) to be set to `true`. The flag may be dereferenced using the function `changes` and set to `false` using the function `reset_changes`. The binary infix-function `o` denotes function composition whereas `phi_compose` denotes composition of flattening environments.

(i.e., 20) is reached. The `loop1` function takes, besides from a counter `N` and an expression `e`, an optimiser environment `E` as argument, which allow for inlining, function specialisation, and constant folding to happen across compilation unit boundaries. The `contract` function implements efficient contract and reduce steps, which in a down-sweep and an up-sweep implements constant-folding, inlining, dead-code elimination, and function specialisation [1]. Common subexpression elimination (i.e., `cse`) and elimination of explicit tuples (i.e., `eliminate_explicit_records`) are intra-procedural and are interleaved with the other optimisations, which aim at introducing new optimisation opportunities. The `loop2` function starts by performing a flattening transformation using `P`, the environment mapping external function definitions to compositional argument transformers. Notice that `P` is used only for the first iteration of `loop2`. After the first iteration, all calls to externally defined functions have been modified according to the argument transformers in `P`. For consecutive passes, argument transformers in `P` are set to the identity transformer (`id`) using the function `phi_nofun`. Notice also that inlining of functions across compilation unit boundaries, which, as mentioned, is performed in the `loop1` phase, takes precedence over deep argument flattening across compilation unit boundaries, which is performed in the `loop2` phase; no inlining or constant propagation happen across compilation unit boundaries in the `loop2` phase, which is enforced

by the `unknown_env` function. Finally, notice that deep argument flattening is interleaved with other optimisations in the `loop2` phase, as other optimisations may trigger new opportunities for argument flattening, including argument uncurrying, argument unboxing, and argument dropping. The resulting flattening environments for consecutive passes are composed using the `phi_compose` function, which takes care of composing argument transformers for exported functions. In practice, optimisations stabilise after about five iterations.

Compared to the argument flattening presented in Section 4, the implementation of argument flattening in the MLKit is extended to support also the passing of arguments of type `real` (IEEE 64-bit floats) unboxed in floating-point registers, as demonstrated in Section 2.

6.1 Deep Argument Flattening and Incremental Recompilation

The deep argument flattening technique fits well within a framework for incremental recompilation [7], for which compilation units are compiled in separation but with the possibility that static information about functions is passed across compilation unit boundaries at compile time. A particular important aspect of the presented deep argument flattening technique is that it is compositional and that the composition of argument transformers expresses how each call site is transformed to match the refined function implementation. With respect to checking whether a compilation unit needs to be recompiled upon changes of source code, it suffices to check whether the compilation unit itself has changed or for each imported function whether any information about the function has changed, such as its type or its compositional argument transformer.

MLKit and ReML [10], which shares the source code with MLKit, uses region inference and region-based memory management to complement reference-tracing garbage collection [19]. Region inference is a type-based program analysis, which also requires refined type information to be passed across compilation unit boundaries at compile time. The deep argument flattening technique makes use of the same mechanism as region inference to detect, upon changes of source code, if assumptions about an imported function has changed.

7 BENCHMARKS

In this section, we evaluate the technique based on a series of benchmarks. All benchmarks are executed on a Thinkpad P14s Gen 4 equipped with a 13th Gen 16-core Intel Core i7-1360P processor and 32GiB RAM. The machine runs Ubuntu 25.04. We compare the performance of code generated by different configurations of MLKit 4.7.15, which implements the techniques presented in this paper, and MLton 20241230, a state-of-the-art Standard ML compiler, which uses a whole-program-compilation approach and which also implements deep flattening of arguments (and also deep-flattening of data structures). Reported numbers are average wall-clock-times of 20 runs, for which we have removed the five largest outliers.

The different configurations of MLKit that we compare are the following:

	Uncurrying	Tuple argument unboxing	Real argument unboxing
`MLKit`^{0}			
`MLKit`$^{\texttt{uc}}$	✓		
`MLKit`$^{\texttt{uc,tup}}$	✓	✓	
`MLKit`	✓	✓	✓

Here `MLKit`^{0} is a version of MLKit without any argument flattening and `MLKit`$^{\texttt{uc}}$ is a version of MLKit that implements uncurrying. The `MLKit`$^{\texttt{uc,tup}}$ version of MLKit implements uncurrying and tuple argument unboxing, whereas `MLKit` is the default version of MLKit, which performs uncurrying, tuple argument unboxing, and real argument unboxing. The MLKit configurations that we report on all use MLKit's native x86-64 backend. In addition, MLKit also features a JavaScript

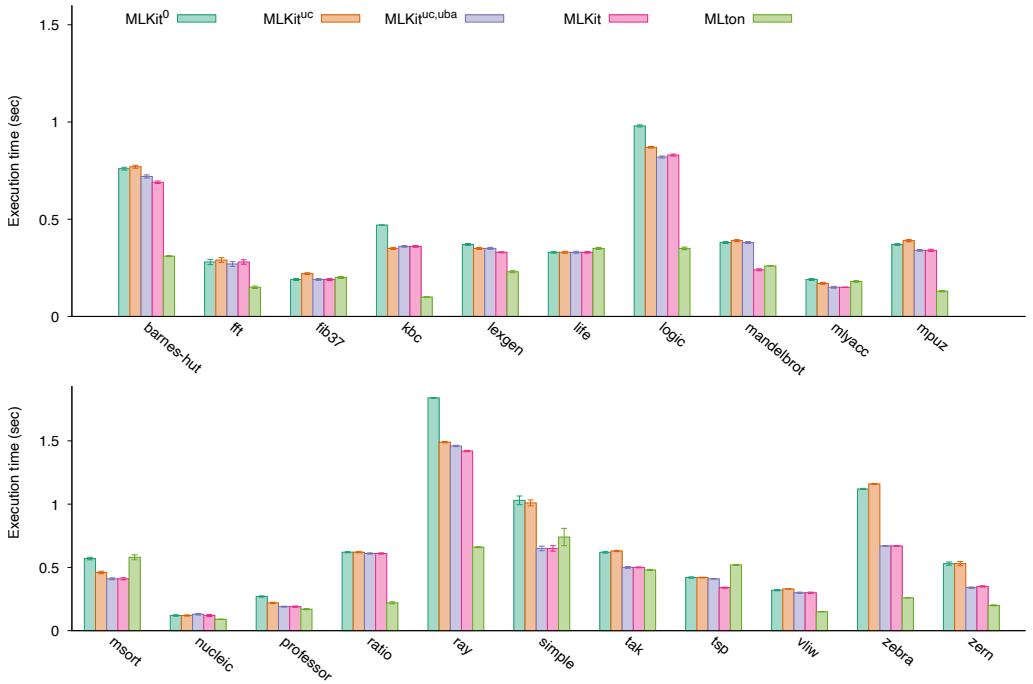

Fig. 7. Average wall-clock execution times for MLKit-generated executables compared to average wall-clock execution times for MLton generated executables. For each vertical bar, markers specify the absolute standard deviation for the corresponding 15 runs.

backend [8], which also benefits from the intermediate language optimisations that we report on here. We have not, however, evaluated directly the effect of the optimisations on JavaScript platforms.

The benchmark results are shown in Figure 7 in terms of user times. Wall-clock execution times are shown in Table 1. There are several observations to be made. First, throughout all benchmarks, `MLKit`^{0} does not perform significantly better than other MLKit configurations. Second, uncurrying provides improvements for `kbc` (Knuth-Bendix Completion), `logic` (unification-based deduction), `msort` (Merge-sort), and `ray` (a ray-tracing benchmark). Most significant is the effect of enabling unboxing of tuple arguments, which leads to significant improvements for `barnes-hut`, `mpuz`, `simple`, `tak`, `zebra`, and `zern`. Enabling also unboxing of real arguments leads to further improvements for `barnes-hut`, `mandelbrot`, `ray`, and `tsp`.

Comparing the performance results with those obtained with MLton, we observe that most of the benchmarks perform better with MLton than they do with any MLKit configuration. Exceptions are the benchmarks `fib37`, `life`, `mandelbrot`, `mlyacc`, `msort`, `simple`, and `tsp`, which run faster with the default MLKit configuration. Those benchmarks that performs particularly poorly with MLKit, compared to MLton, are `barnes-hut`, `fft`, `kbc`, `lexgen`, `logic`, `mpuz`, `ratio`, `ray`, `vliw`, `zebra`, and `zern`. We believe that the main reason for MLton's better performance is MLton's deep-flattening of data structures and function return values.

Table 1. Benchmark average wall-clock execution times (in seconds) for different MLKit configurations and MLton. Averages and their relative standard errors are computed, for each benchmark, based on 20 runs with the wall-clock execution time for the five worst runs removed.

Benchmark	*Lines*	MLKit^{0}	MLKit^{uc}	MLKit^{uc,tup}	MLKit	MLton
barnes-hut	1245	0.76 ± 1%	0.77 ± 1%	0.72 ± 1%	0.69 ± 1%	0.31 ± 1%
fft	71	0.28 ± 5%	0.29 ± 4%	0.27 ± 4%	0.28 ± 4%	0.15 ± 4%
fib37	7	0.19 ± 3%	0.22 ± 2%	0.19 ± 2%	0.19 ± 3%	0.20 ± 2%
kbc	682	0.47 ± 0%	0.35 ± 1%	0.36 ± 1%	0.36 ± 1%	0.10 ± 0%
lexgen	1322	0.37 ± 1%	0.35 ± 1%	0.35 ± 1%	0.33 ± 1%	0.23 ± 2%
life	202	0.33 ± 1%	0.33 ± 2%	0.33 ± 2%	0.33 ± 1%	0.35 ± 1%
logic	355	0.98 ± 1%	0.87 ± 0%	0.82 ± 1%	0.83 ± 1%	0.35 ± 2%
mandelbrot	62	0.38 ± 1%	0.39 ± 1%	0.38 ± 1%	0.24 ± 2%	0.26 ± 0%
mlyacc	7385	0.19 ± 3%	0.17 ± 3%	0.15 ± 3%	0.15 ± 0%	0.18 ± 2%
mpuz	124	0.37 ± 1%	0.39 ± 2%	0.34 ± 1%	0.34 ± 2%	0.13 ± 3%
msort	113	0.57 ± 2%	0.46 ± 2%	0.41 ± 2%	0.41 ± 2%	0.58 ± 3%
nucleic	3215	0.12 ± 5%	0.12 ± 4%	0.13 ± 5%	0.12 ± 7%	0.09 ± 0%
professor	282	0.27 ± 2%	0.22 ± 2%	0.19 ± 2%	0.19 ± 3%	0.17 ± 2%
ratio	620	0.62 ± 1%	0.62 ± 1%	0.61 ± 1%	0.61 ± 1%	0.22 ± 4%
ray	533	1.84 ± 0%	1.49 ± 0%	1.46 ± 0%	1.42 ± 0%	0.66 ± 1%
simple	1055	1.03 ± 3%	1.01 ± 2%	0.65 ± 3%	0.65 ± 3%	0.74 ± 9%
tak	12	0.62 ± 1%	0.63 ± 1%	0.50 ± 1%	0.50 ± 1%	0.48 ± 1%
tsp	494	0.42 ± 1%	0.42 ± 1%	0.41 ± 1%	0.34 ± 1%	0.52 ± 1%
vliw	3681	0.32 ± 2%	0.33 ± 0%	0.30 ± 2%	0.30 ± 2%	0.15 ± 0%
zebra	313	1.12 ± 0%	1.16 ± 0%	0.67 ± 0%	0.67 ± 0%	0.26 ± 0%
zern	605	0.53 ± 3%	0.53 ± 3%	0.34 ± 1%	0.35 ± 1%	0.20 ± 2%

8 RELATED WORK

Highly related to the current work is work on tuple flattening [21], arity raising [3], and unboxing of tuples [12–14, 18]. In this work we are, in particular, interested in supporting compiler implementations that do not assume compilation of whole programs. At the same time we are interested in supporting that some unboxing and flattening decisions can cross module boundaries by enriching the static information available at compile time.

Also related to this work is work on establishing and expressing dependencies between function implementations and function uses, in particular to ensure that a function's calling convention is satisfied by all possible calls to the function [16]. Compared to the higher-order setting supported in [16], our work does not support non-standard calling conventions for unknown functions (i.e., functions that are not `fun`-bound).

Another area of related work is work on type systems for memory layout [15], representation polymorphism [6], unboxed data types [4], and bit-stealing [2, 9], which together allows for more compact representations of data structures. Also related to our work is work on expressing boxing and unboxing operations in an intermediate language, while supporting polymorphism and higher-order functions [5]. Such mechanisms seam valuable for extending our work to also support unboxing and specialised calling conventions involving higher-order functions.

9 CONCLUSION AND FUTURE WORK

We have presented a technique for improving function argument passing that work well together with a framework that supports incremental compilation. There are many possibilities for future work. First, we have not here demonstrated a soundness property for the technique. We conjecture, however, that, using standard operational semantics techniques, it should be possible to demonstrate a soundness property, perhaps using a logical-relation argument. Another possibility for future work would be to support also flattened return values and to investigate the possibility for supporting flattening of higher-order functions based on properties of how passed functions are used within a known function.

Finally, providing support for flattening of abstract local data structures seems often paramount for obtaining code as efficient as that generated with MLton. We leave such an effort as future work.

REFERENCES

[1] Andrew W. Appel and Trevor Jim. 1997. Shrinking lambda expressions in linear time. *J. Funct. Program.* 7, 5 (sep 1997), 515–540. https://doi.org/10.1017/S0956796897002839

[2] Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. 2023. Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types. *Proc. ACM Program. Lang.* 7, ICFP, Article 216 (aug 2023), 34 pages. https://doi.org/10.1145/3607858

[3] Lars Bergstrom and John Reppy. 2009. Arity raising in Manticore. In *Proceedings of the 21st International Conference on Implementation and Application of Functional Languages* (South Orange, NJ, USA) *(IFL'09)*. Springer-Verlag, Berlin, Heidelberg, 90–106.

[4] Nicolas Chataing, Stephen Dolan, Gabriel Scherer, and Jeremy Yallop. 2024. Unboxed Data Constructors: Or, How cpp Decides a Halting Problem. *Proc. ACM Program. Lang.* 8, POPL, Article 51 (jan 2024), 31 pages. https://doi.org/10.1145/3632893

[5] Paul Downen. 2024. Call-by-Unboxed-Value. *Proc. ACM Program. Lang.* 8, ICFP, Article 265 (Aug. 2024), 35 pages. https://doi.org/10.1145/3674654

[6] Richard A. Eisenberg and Simon Peyton Jones. 2017. Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 525–539. https://doi.org/10.1145/3062341.3062357

[7] Martin Elsman. 2008. *A Framework for Cut-Off Incremental Recompilation and Inter-Module Optimization.* Technical Report. IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark.

[8] Martin Elsman. 2011. SMLtoJs: Hosting a Standard ML Compiler in a Web Browser. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients* (Portland, Oregon, USA) *(PLASTIC '11)*. Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/2093328.2093336

[9] Martin Elsman. 2024. Double-Ended Bit-Stealing for Algebraic Data Types. *Proc. ACM Program. Lang.* 8, ICFP, Article 239 (Aug. 2024), 33 pages. https://doi.org/10.1145/3674628

[10] Martin Elsman. 2024. Explicit Effects and Effect Constraints in ReML. *Proc. ACM Program. Lang.* 8, POPL, Article 79 (Jan. 2024), 25 pages. https://doi.org/10.1145/3632921

[11] Martin Elsman and Niels Hallenberg. 1995. An Optimizing Backend for the ML Kit Using a Stack of Regions. Student Project 95-7-8, University of Copenhagen (DIKU).

[12] Fritz Henglein and Jesper Jørgensen. 1994. Formally Optimal Boxing. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 213–226.

[13] Xavier Leroy. 1992. Unboxed Objects and Polymorphic Typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 177–188.

[14] Xavier Leroy. 1997. The effectiveness of type-based unboxing. In *Proceedings of the 1997 ACM Workshop on Types in Compilation* (Amsterdam). https://api.semanticscholar.org/CorpusID:2398412

[15] Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. 2003. A type theory for memory allocation and data layout. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 172–184. https://doi.org/10.1145/604131.604147

[16] Benjamin Quiring, David Van Horn, John Reppy, and Olin Shivers. 2025. Webs and Flow-Directed Well-Typedness Preserving Program Transformations. *Proc. ACM Program. Lang.* 9, PLDI, Article 177 (June 2025), 25 pages. https:

//doi.org/10.1145/3729280

[17] David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. 1996. TIL: A Type-Directed Optimizing Compiler for ML. In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*. 181–192. citeseer.nj.nec.com/tarditi95til.html

[18] Peter J. Thiemann. 1995. Unboxed values and polymorphic typing revisited. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/224164.224175

[19] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17, 3 (01 Sep 2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

[20] Stephen Weeks. 2006. Whole-Program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, USA) *(ML '06)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/1159876.1159877

[21] Lukasz Ziarek, Stephen Weeks, and Suresh Jagannathan. 2008. Flattening tuples in an SSA intermediate representation. *Higher Order Symbol. Comput.* 21, 3 (sep 2008), 333–358. https://doi.org/10.1007/s10990-008-9035-3

A APPENDIX: PROOFS

Proposition 4.3. (Type Argument Transformation). *Assume* $\tau = \langle\vec{\tau}\rangle \to \Gamma_0 \to \tau_0$ *and* types $\vec{p} = \vec{\tau}$ *and* $\Gamma, \vec{p}, \Gamma_0 \vdash e : \tau_0$. *If* $\mu \vdash_{\text{opt}} (\tau, \vec{p}, \lambda\Gamma_0.e) \Rightarrow (\tau', \vec{q}, e')$ *then* $\mu(\tau) = \tau'$ *and* $\tau' = \langle\vec{\tau}'\rangle \to \tau_0$, *where* $\vec{\tau}' =$ types $\vec{q}$ *and* $\Gamma, \vec{q} \vdash e' : \tau_0$. *Moreover, if* e *is on the form* $\lambda\Gamma_1.e_1$ *for some* e_1, *then* e' *if of the form* $\lambda\Gamma_1.e_1'$, *for some* e_1'.

Proof. By induction over the derivation of $\mu \vdash_{\text{opt}} (\tau, \vec{p}, e) \Rightarrow (\tau', \vec{q}, e')$.

Case o-drop We have ⌈1⌉ $\mu = \texttt{drop}(i)$ and ⌈2⌉ $\vec{p} = p_1, \cdots, p_n$ and ⌈3⌉ $\vec{q} = p_1, \cdots, p_{i-1}, p_{i+1}, \cdots p_n$ and ⌈4⌉ $\tau' = \langle\vec{\tau}'\rangle \to \tau_0$, where $\vec{\tau}' =$ types $\vec{q}$ and ⌈5⌉ $\Gamma_0 = \cdot$ and ⌈6⌉ $e' = e'$ and ⌈7⌉ $x_i \notin \text{fv}\ e$. It follows from definition of $\mu(\tau)$ that $\mu(\tau) = \tau'$ as required. Using ⌈2⌉, ⌈3⌉, ⌈6⌉, ⌈7⌉, and assumptions, we can apply Proposition 3.3 to get $\Gamma, \vec{q} \vdash e' : \tau_0$, as required. The last property follows immediately due to ⌈6⌉.

Case o-add We have ⌈1⌉ $\mu = \texttt{addprj}(i, j)$ ⌈2⌉ $\vec{p} = p_1, \cdots, p_n$ and $p_i = x_i : (\tau_1' \times \cdots \times \tau_m')$ and ⌈3⌉ $\vec{q} = \vec{p}, x : \tau_j'$ and ⌈4⌉ $\Gamma_0 = \cdot$ and ⌈5⌉ $e' = e[x/\#j\ x_i]$ and ⌈6⌉ $\vec{\tau}' = \vec{\tau}, \tau_j'$ and ⌈7⌉ $\tau' = \langle\vec{\tau}'\rangle \to \tau_0$, where $\vec{\tau}' =$ types $\vec{q}$. We have from the definition of $\mu(\tau)$ and from the definition of $\texttt{prj}(i, j)$ that ⌈8⌉ $\mu(\tau) = \tau'$ as required. We also have ⌈7⌉ as required. Let $a = \#j\ x_i$. From [t-sel], [t-var], ⌈2⌉, and ⌈3⌉, we have ⌈10⌉ $\Gamma, \vec{q} \vdash a : \tau_j'$. From Proposition 3.4 and ⌈10⌉, and because $(\Gamma, \vec{q})(x) = \tau_j'$, we have $\Gamma, \vec{q} \vdash e[x/a] : \tau_0$, as required. The last property follows immediately due to ⌈5⌉.

Case o-unc We have ⌈1⌉ $\tau = \langle\vec{\tau}\rangle \to \Gamma_0 \to \tau_0$ and ⌈2⌉ $\mu = \texttt{uncurry}$ and ⌈3⌉ $\Gamma_0 = x : \tau_1$ and ⌈4⌉ $\vec{\tau} =$ types $\vec{p}$ and ⌈5⌉ $\tau' = \langle\vec{\tau}, \tau_1\rangle \to \tau_0$ and ⌈6⌉ $\vec{q} = \vec{p}, x : \tau_1$ and ⌈7⌉ $\vec{\tau}' = \vec{\tau}, \tau_1$ and ⌈8⌉ $e' = e$. From the definition of $\mu(\tau)$, we have $\mu(\tau) = \tau'$ as required. We also have ⌈5⌉ and types $\vec{q} = \vec{\tau}'$, as required. From assumptions, we have ⌈9⌉ $\Gamma, \vec{p}, x : \tau_1 \vdash e : \tau_0$. Now, from ⌈6⌉, ⌈8⌉, and ⌈10⌉, we have $\Gamma, \vec{q} \vdash e' : \tau_0$, as required. The last property follows immediately due to ⌈8⌉.

Case o-comp We have ⌈1⌉ $\tau = \vec{\tau} \to (\Gamma_0 \to \tau_0)$ and ⌈2⌉ $\Gamma, \vec{p}, \Gamma_0 \vdash e : \tau_0$ and ⌈3⌉ $\mu_2 \circ \mu_1 \vdash_{\text{opt}} (\tau, \vec{p}, \lambda\Gamma_0.e) \Rightarrow (\tau', \vec{q}, e')$. From [o-comp] and ⌈3⌉, we have ⌈4⌉ $\mu_1 \vdash_{\text{opt}} (\tau, \vec{p}, \lambda\Gamma_0.e) \Rightarrow (\tau'', \vec{r}, e'')$ and ⌈5⌉ $\mu_2 \vdash_{\text{opt}} (\tau'', \vec{r}, e'') \Rightarrow (\tau', \vec{q}, e')$. We assume Γ_1 and Γ_2 such that ⌈6⌉ $\Gamma_0 = \Gamma_1, \Gamma_2$ and ⌈7⌉ $\lambda\Gamma_0.e = \lambda\Gamma_1.\lambda\Gamma_2.e$. From ⌈2⌉, ⌈6⌉, and ⌈7⌉, and by repeated use of [t-lam], we have ⌈8⌉ $\Gamma, \vec{p}, \Gamma_1 \vdash \lambda\Gamma_2.e : \Gamma_2 \to \tau_0$. From ⌈1⌉, we have ⌈9⌉ $\tau = \langle\vec{\tau}\rangle \to (\Gamma_1 \to (\Gamma_2 \to \tau_0))$. From ⌈4⌉ and ⌈7⌉, we have ⌈10⌉ $\mu_1 \vdash_{\text{opt}} (\tau, \vec{p}, \lambda\Gamma_1.\lambda\Gamma_2.e) \Rightarrow (\tau'', \vec{r}, e'')$.

By induction applied to ⌈8⌉, ⌈9⌉, and ⌈10⌉, we have ⌈11⌉ $\mu_1(\tau) = \tau''$ and ⌈12⌉ $\tau'' = \langle\vec{\tau}''\rangle \to (\Gamma_2 \to \tau_0)$, where $\vec{\tau}'' =$ types $\vec{r}$, and ⌈13⌉ $\Gamma, \vec{r} \vdash e'' : \Gamma_2 \to \tau_0$ and ⌈14⌉ if $\lambda\Gamma_2.e = \lambda\Gamma_2.\lambda\Gamma_3.e_3$, for some e_3, then ⌈15⌉ $e'' = \lambda\Gamma_2.\lambda\Gamma_3.e_3'$ for some e_3'. From ⌈13⌉ and ⌈15⌉, we have ⌈16⌉ $\Gamma, \vec{r} \vdash \lambda\Gamma_2.\lambda\Gamma_3.e_3' : \Gamma_2 \to \tau_0$. From ⌈5⌉ and ⌈15⌉, we have ⌈17⌉ $\mu_2 \vdash_{\text{opt}} (\tau'', \vec{r}, \lambda\Gamma_2.\lambda\Gamma_3.e_3') \Rightarrow (\tau', \vec{q}, e')$. From ⌈16⌉ and repeated use of [t-lam], we have ⌈18⌉ $\Gamma, \vec{r}, \Gamma_2 \vdash \lambda\Gamma_3.e_3' : \tau_0$.

By induction applied to ⌈12⌉, ⌈18⌉, and ⌈17⌉, we have ⌈19⌉ $\mu_2(\tau'') = \tau'$ and ⌈20⌉ $\tau' = \langle\vec{\tau}'\rangle \to \tau_0$, where $\vec{\tau}' =$ types $\vec{q}$, and ⌈21⌉ $\Gamma, \vec{q} \vdash e' : \tau_0$ and ⌈22⌉ $e' = \lambda\Gamma_3.e_3''$, for some e_3''. From ⌈19⌉ and ⌈11⌉, we have $\mu_2(\mu_1(\tau)) = \tau'$ and thus $(\mu_2 \circ \mu_1)(\tau) = \tau'$, as required. We also have ⌈20⌉ and ⌈21⌉ as required. Moreover, we have, as required, from ⌈14⌉, ⌈15⌉, and ⌈22⌉ that, if $e = \lambda\Gamma_3.e_3$, for some e_3, then $e' = \lambda\Gamma_3.e_3''$, for some e_3''.

Case o-id This case follows immediately from assumptions.

□

Proposition 5.2. (Translation of Call Expressions Closed Under Typings). *Assume* $\sigma \geq \langle\vec{\tau}\rangle \to \tau$. *If* $\Gamma, f : \sigma \vdash f\ \langle\vec{a}\rangle : \tau$ *and* $\mu \vdash_{\text{call}} f\ \langle\vec{a}\rangle : \tau \Rightarrow f\ \langle\vec{a}'\rangle : \tau', \Gamma_0$ *and* $\sigma' = \mu(\sigma)$ *then* $\Gamma, f : \sigma', \Gamma_0 \vdash f\ \langle\vec{a}'\rangle : \tau'$ *and* $\tau = \Gamma_0 \to \tau'$.

Proof. By induction on the relation $\mu \vdash_{\text{call}} f\ \langle\vec{a}\rangle : \tau \Rightarrow f\ \langle\vec{a}'\rangle : \tau', \Gamma_0$. We proceed by case analysis.

Case C-DROP We have ⌈1⌉ $\vec{a}' = a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n$ and ⌈2⌉ $\Gamma_0 = \cdot$ and ⌈3⌉ $\tau' = \tau$. From assumptions and [T-FAPP], we have ⌈4⌉ $\Gamma, f : \sigma \vdash a_i : \tau_i$, for all $i = 1..n$. Because access expressions do not contain `fun`-bound variables, we have ⌈5⌉ $\Gamma, f : \sigma' \vdash a_i : \tau_i$, for all $i = 1..n$. Because $\mu = \texttt{drop}(i)$ and from the definition of $\mu(\sigma)$ and $\mu(\sigma) = \sigma'$, we have ⌈6⌉ $\sigma' \geq \langle \vec{\tau}' \rangle \to \tau$, where $\vec{\tau}' = \tau_1, \cdots, \tau_{i-1}, \tau_{i+1}, \cdots, \tau_n$. Now, from [T-FAPP], [T-SUB], [T-VAR], ⌈1⌉, ⌈2⌉, ⌈3⌉, ⌈5⌉, and ⌈6⌉, we have $\Gamma, f : \sigma', \Gamma_0 \vdash f\ \langle \vec{a}' \rangle : \tau'$, as required. Moreover, from ⌈2⌉ and ⌈3⌉, we have $\tau = \Gamma_0 \to \tau'$, as required.

Case C-UNC We have ⌈1⌉ $\vec{a}' = \vec{a}, x$ and ⌈2⌉ $\Gamma_0 = x : \tau_0$ and ⌈3⌉ $\tau = \Gamma_0 \to \tau'$. From assumptions and [T-FAPP], we have ⌈4⌉ $\Gamma, f : \sigma \vdash a_i : \tau_i$ with $\vec{\tau} = \tau_1, \cdots, \tau_n$. Because access expressions do not contain `fun`-bound variables, we have ⌈5⌉ $\Gamma, f : \sigma' \vdash a_i : \tau_i$. Moreover, we can assume x is picked fresh such that $\mathrm{Dom}\ \Gamma_0 \cap \mathrm{Dom}\ \Gamma = \emptyset$. Thus, from Proposition 3.5, we have ⌈6⌉ $\Gamma, f : \sigma', \Gamma_0 \vdash a_i : \tau_i$. From assumptions and ⌈3⌉, we have ⌈7⌉ $\sigma \geq \langle \vec{\tau} \rangle \to \Gamma_0 \to \tau'$. From assumptions, we have ⌈8⌉ $\sigma' = \mu(\sigma)$. Because $\mu = \texttt{uncurry}$, from the definition of $\mu(\sigma)$, and from ⌈7⌉, we have ⌈9⌉ $\sigma' \geq \langle \vec{\tau}, \tau_0 \rangle \to \tau'$. It follows that we have ⌈11⌉ $\sigma' \geq \langle \vec{\tau}' \rangle \to \tau'$, where $\vec{\tau}' = \vec{\tau}, \tau_0$. From assumptions, we have $a'_i = a_i$ for $i = 1..n$ and $a'_{n+1} = x$, where $\vec{a}' = a'_1, \cdots, a'_{n+1}$. From ⌈2⌉, and [T-VAR], we have ⌈12⌉ $\Gamma, f : \sigma', \Gamma_0 \vdash x : \tau_0$. It follows from ⌈12⌉ and ⌈6⌉ that ⌈13⌉ $\Gamma, f : \sigma', \Gamma_0 \vdash a'_i : \tau'_i$ for $i = 1..n+1$. From [T-FAPP], [T-SUB], [T-VAR], ⌈11⌉, and ⌈13⌉, we have $\Gamma, f : \sigma', \Gamma_0 \vdash f\ \langle \vec{a}' \rangle : \tau'$ as required. We also have ⌈3⌉ as required.

Case C-COMP From [C-COMP], we have ⌈1⌉ $\mu_1 \vdash_{\text{call}} f\ \langle \vec{a} \rangle : \tau \Rightarrow f\ \langle \vec{a}'' \rangle : \tau'', \Gamma_1$ and ⌈2⌉ $\mu_2 \vdash_{\text{call}} f\ \langle \vec{a}'' \rangle : \tau'' \Rightarrow f\ \langle \vec{a}' \rangle : \tau', \Gamma_2$ and ⌈3⌉ $\Gamma_0 = \Gamma_1, \Gamma_2$. From assumptions, we have ⌈4⌉ $\sigma \geq \langle \vec{\tau} \rangle \to \tau$ and ⌈5⌉ $\Gamma, f : \sigma \vdash f\ \langle \vec{a} \rangle : \tau$.

By induction using ⌈1⌉, ⌈4⌉, ⌈5⌉, and by letting ⌈8⌉ $\sigma_1 = \mu_1(\sigma)$, we have ⌈6⌉ $\Gamma, \Gamma_1, f : \sigma_1 \vdash f\ \langle \vec{a}'' \rangle : \tau''$ and ⌈7⌉ $\tau = \Gamma_1 \to \tau''$. From [T-FAPP], [T-SUB], [T-VAR], and ⌈6⌉, we have there exists $\vec{\tau}''$ such that ⌈9⌉ $\sigma_1 \geq \langle \vec{\tau}'' \rangle \to \tau''$ and ⌈10⌉ $\Gamma, \Gamma_1 \vdash a''_i : \tau''_i$ for all a''_i in $\vec{a}''$, where $\vec{\tau}'' = \tau''_1, \cdots, \tau''_k$.

Now, by induction using ⌈2⌉, ⌈6⌉, ⌈9⌉, we have ⌈11⌉ $\Gamma, \Gamma_1, \Gamma_2, f : \mu_2(\sigma_1) \vdash f\ \langle \vec{a}' \rangle : \tau'$, where ⌈12⌉ $\tau'' = \Gamma_2 \to \tau'$. From ⌈3⌉, ⌈8⌉, and ⌈11⌉, we have $\Gamma, \Gamma_0, f : \sigma' \vdash f\ \langle \vec{a}' \rangle : \tau'$, as required. Moreover, from ⌈7⌉ and ⌈12⌉, we have ⌈13⌉ $\tau = \Gamma_1 \to \Gamma_2 \to \tau'$, thus, from ⌈3⌉ and ⌈13⌉, we also have $\tau = \Gamma_0 \to \tau'$, as required.

Case C-ID This case follows immediately from assumptions, [C-ID], and from $\Gamma_0 = \cdot$.

Case C-ADD This case follows similarly to the cases for C-UNC and C-DROP. □

Proposition 5.3. (Typing Preserved Under Transformation). *Assume* $\phi \vdash \Gamma \sim_\rho \Gamma'$ *and* $\Gamma \vdash e : \tau$. *If* $\phi \vdash e \Rightarrow e' : \tau$ *then* $\Gamma' \vdash e' : \tau$.

Proof. By induction over the derivation of $\phi \vdash e \Rightarrow e' : \tau$. Most of the cases are either trivial or follow directly by induction. We show the three interesting cases.

Case F-VAR From assumptions and [T-VAR], we have ⌈1⌉ $\Gamma(x) = \tau$ and from assumptions and [F-VAR], we have ⌈2⌉ $\phi(x) = (\tau, id)$ and ⌈3⌉ $e' = x$. From assumptions and [E-VAR], we have $\mu(\tau) = \tau'$, thus, because $\mu = id$ follows from ⌈2⌉, we have ⌈4⌉ $\tau' = \tau$. From [E-VAR], we have ⌈5⌉ $\Gamma'(x) = \tau$. It follows from [T-VAR], ⌈5⌉, and ⌈3⌉ that $\Gamma' \vdash e' : \tau$, as required.

Case F-FAPP From rule F-FAPP, we have ⌈1⌉ $\phi(f) = (\sigma, \mu)$ and ⌈2⌉ $\sigma \geq \langle \vec{\tau} \rangle \to \tau$ and ⌈3⌉ $\mu \vdash_{\text{call}} f\ \langle \vec{a} \rangle : \tau \Rightarrow f\ \langle \vec{a}' \rangle : \tau', \Gamma_0$ and ⌈4⌉ $\tau = \Gamma_0 \to \tau'$ and ⌈5⌉ $e' = \lambda \Gamma_0 . f\ \langle \vec{a}' \rangle$ and ⌈6⌉ $e = f\ \langle \vec{a} \rangle$. From [E-VAR], we have ⌈7⌉ $\Gamma(f) = \sigma$ and ⌈8⌉ $\Gamma'(f) = \sigma'$ and ⌈9⌉ $\sigma' = \mu(\sigma)$. From assumptions, we have ⌈10⌉ $\Gamma \vdash f\ \langle \vec{a} \rangle : \tau$ From ⌈7⌉, we have ⌈11⌉ $\Gamma = \Gamma_1, f : \sigma$, for some Γ_1. We can now apply Proposition 5.2 using ⌈2⌉, ⌈10⌉, ⌈11⌉, ⌈3⌉, and ⌈9⌉ to get ⌈12⌉ $\Gamma_1, f : \sigma', \Gamma_0 \vdash f\ \langle \vec{a}' \rangle : \tau'$ and ⌈13⌉ $\tau = \Gamma_0 \to \tau'$. From ⌈8⌉, we have ⌈14⌉ $\Gamma' = \Gamma'_1, f : \sigma'$, for some Γ'_1. From ⌈12⌉, [T-FAPP], [T-SUB], [T-VAR], we have ⌈15⌉ $\Gamma_1, f : \sigma', \Gamma_0 \vdash a'_i : \tau'_i$, for all $a'_i \in \vec{a}'$, ⌈16⌉ $\vec{\tau}' = \tau'_1, \cdots, \tau'_k$ and ⌈17⌉ $\sigma' = \langle \vec{\tau}' \rangle \to \tau'$. Because $f \notin \mathrm{fv}(\vec{a}')$, we can apply Proposition 3.3 to ⌈15⌉ to get ⌈18⌉ $\Gamma_1, \Gamma_0 \vdash a'_i : \tau'_i$, for all $a'_i \in \vec{a}'$.

From ⌈1⌉, we have ⌈19⌉ $\phi = \phi_1, f : (\sigma, \mu)$, for some ϕ_1. From [E-VAR], assumptions, ⌈11⌉, ⌈14⌉, ⌈19⌉, we have ⌈20⌉ $\phi_1 \vdash \Gamma_1 \sim \Gamma_1'$. From [E-VAR] and ⌈20⌉, we have ⌈21⌉ $\phi_1, \Gamma_0 : \mathtt{id} \vdash \Gamma_1, \Gamma_0 \sim \Gamma_1', \Gamma_0$. From Proposition 5.1 using ⌈21⌉ and ⌈18⌉, we have ⌈22⌉ $\Gamma_1', \Gamma_0 \vdash a_i' : \tau_i'$, for all $a_i' \in \vec{a}'$. From Proposition 3.5 using ⌈22⌉, we have ⌈23⌉ $\Gamma_1', f : \sigma', \Gamma_0 \vdash a_i' : \tau_i'$, for all $a_i' \in \vec{a}'$. Now, using [T-FAPP], [T-SUB], [T-VAR], ⌈14⌉, ⌈16⌉, ⌈17⌉, and ⌈23⌉, we have ⌈24⌉ $\Gamma', \Gamma_0 \vdash f\ \langle \vec{a}' \rangle : \tau'$. From ⌈24⌉ and by repeated use of [T-LAM], we have ⌈25⌉ $\Gamma' \vdash \lambda\Gamma_0.f\ \langle \vec{a}' \rangle : \Gamma_0 \to \tau'$. From ⌈5⌉, ⌈13⌉, and ⌈25⌉, we have $\Gamma' \vdash e' : \tau$, as required.

Case F-FUN From [F-FUN], we have ⌈1⌉ $\tau_1 = \langle \vec{\tau} \rangle \to \Gamma_0 \to \tau_0$ and ⌈2⌉ $e = \mathtt{fun}\, f : \forall\vec{\alpha}.\tau_1\ \langle \vec{p} \rangle = \lambda\Gamma_0.e_1\ \mathtt{in}\ e_2$ and ⌈3⌉ $e' = \mathtt{fun}\, f : \forall\vec{\alpha}.\tau_1'\ \langle \vec{q} \rangle = e_1''\ \mathtt{in}\ e_2'$ and ⌈4⌉ $\mu \vdash_{\mathrm{opt}} (\tau_1, \vec{p}, \lambda\Gamma_0.e_1) \Rightarrow (\tau_1', \vec{q}, e_1')$ and ⌈5⌉ $\phi, f : (\tau_1, \mu), \vec{q} : \mathtt{id} \vdash e_1' \Rightarrow e_1'' : \tau_0$ and ⌈6⌉ $\phi, f : (\forall\vec{\alpha}.\tau_1, \mu) \vdash e_2 \Rightarrow e_2' : \tau$. From assumptions and [T-FUN], we have ⌈7⌉ $\Gamma, f : \tau_1, \vec{p} \vdash \lambda\Gamma_0.e_1 : \tau_0$ and ⌈8⌉ types $\vec{p} = \vec{\tau}$ and ⌈9⌉ $\Gamma, f : \forall\vec{\alpha}.\tau_1 \vdash e_2 : \tau$ and ⌈10⌉ $\{\vec{\alpha}\} \cap \mathrm{ftv}(\Gamma, \tau) = \emptyset$. From ⌈7⌉ and by repeated application of [T-LAM], we have ⌈11⌉ $\Gamma, f : \tau_1, \vec{p}, \Gamma_0 \vdash e_1 : \tau_0$. We can now apply Proposition 4.3 using ⌈1⌉, ⌈8⌉, ⌈11⌉, and ⌈4⌉ to get ⌈12⌉ $\mu(\tau_1) = \tau_1'$ and ⌈13⌉ $\tau_1' = \langle \vec{\tau}' \rangle \to \tau_0$, where ⌈14⌉ $\vec{\tau}' = \mathrm{types}\ \vec{q}$, and ⌈15⌉ $\Gamma, f : \tau_1, \vec{q} \vdash e_1' : \tau_0$. Now, let ⌈16⌉ $\phi' = \phi, f : (\tau_1, \mu), \vec{q} : \mathtt{id}$. Further, let ⌈17⌉ $\Gamma_1 = \Gamma, f : \tau_1, \vec{q}$ and ⌈18⌉ $\Gamma_1' = \Gamma', f : \tau_1', \vec{q}$. From assumptions we have ⌈19⌉ $\phi \vdash_\rho \Gamma \sim \Gamma'$. Now, from the definitions of Γ_1, Γ_1', ϕ', from [E-VAR] and ⌈12⌉, we have ⌈20⌉ $\phi' \vdash \Gamma_1 \sim \Gamma_1'$. From ⌈16⌉ and ⌈5⌉, we have ⌈21⌉ $\phi' \vdash e_1' \Rightarrow e_1'' : \tau_0$. From ⌈17⌉ and ⌈15⌉, we have ⌈22⌉ $\Gamma_1 \vdash e_1' : \tau_0$. We can now apply induction using ⌈20⌉, ⌈22⌉, and ⌈21⌉ to get ⌈23⌉ $\Gamma_1' \vdash e_1'' : \tau_0$.

Now, let ⌈24⌉ $\phi'' = \phi, f : (\forall\vec{\alpha}.\tau_1, \mu)$, let ⌈25⌉ $\Gamma_2 = \Gamma, f : \forall\vec{\alpha}.\tau_1$, and let ⌈26⌉ $\Gamma_2' = \Gamma', f : \forall\vec{\alpha}.\tau_1'$. From ⌈12⌉, we have ⌈27⌉ $\mu(\forall\vec{\alpha}.\tau_1) = \forall\vec{\alpha}.\tau_1'$. From the definitions of ϕ'', Γ_2, and Γ_2', and from [E-VAR] and ⌈27⌉, we have ⌈28⌉ $\phi'' \vdash \Gamma_2 \sim \Gamma_2'$. From ⌈6⌉ and ⌈24⌉, we have ⌈29⌉ $\phi'' \vdash e_2 \Rightarrow e_2' : \tau$. From ⌈9⌉ and ⌈25⌉, we have ⌈30⌉ $\Gamma_2 \vdash e_2 : \tau$. We can now apply induction using ⌈28⌉, ⌈30⌉, and ⌈29⌉ to get ⌈31⌉ $\Gamma_2' \vdash e_2' : \tau$.

Now, from [T-FUN], ⌈3⌉, ⌈10⌉, ⌈13⌉, ⌈18⌉, ⌈23⌉, ⌈26⌉, and ⌈31⌉, we have $\Gamma' \vdash e' : \tau$, as required.

□