Type-Preserving Flat Closure Optimization

ADAM T. GELLER, University of British Columbia, Canada
SEAN BOCIRNEA, University of British Columbia, Canada
CHESTER J. F. GOULD, University of British Columbia, Canada
PAULETTE KORONKEVICH, University of British Columbia, Canada
WILLIAM J. BOWMAN, University of British Columbia, Canada

Type-preserving compilation seeks to make *intent* as much as a part of compilation as *computation*. Specifications of intent in the form of types are preserved and exploited during compilation and linking, alongside the mere computation of a program. This provides lightweight guarantees for compilation, optimization, and linking. Unfortunately, type-preserving compilation typically interferes with important optimizations. In this paper, we study typed closure representation and optimization. We analyze limitations in prior typed closure conversion representations, and the requirements of many important closure optimizations. We design a new typed closure representation in our Flat-Closure Calculus (FCC) that admits all these optimizations, prove type safety and subject reduction of FCC, prove type preservation from an existing closure converted IR to FCC, and implement common closure optimizations for FCC.

CCS Concepts: • **Theory of computation → Type structures**; • **Software and its engineering → Formal software verification**; *Software performance*.

Additional Key Words and Phrases: Types, Compilers, Optimization, Type-preserving Compilation, Closure Conversion

ACM Reference Format:
Adam T. Geller, Sean Bocirnea, Chester J. F. Gould, Paulette Koronkevich, and William J. Bowman. 2025. Type-Preserving Flat Closure Optimization. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 103 (April 2025), 27 pages. https://doi.org/10.1145/3720437

1 Introduction

Compiler correctness is critically important, as a single compiler bug can affect millions of developers, and go on to affect hundreds of millions of end users. Unfortunately, formally specifying and verifying a compiler is incredibly hard. For context, the CompCert C compiler has a specification and verification overhead of approximately 6x—the amount of code written to formally specify and verify the compiler is 6x the size of the compiler itself [Leroy 2009]. The situation can get worse if you want guarantees for separate compilation and linking with external or handwritten target code—Compositional CompCert [Stewart et al. 2015] doubled the specification and proof overhead of CompCert.

Authors' Contact Information: Adam T. Geller, University of British Columbia, Vancouver, Canada, atgeller@cs.ubc.ca; Sean Bocirnea, University of British Columbia, Vancouver, Canada; Chester J. F. Gould, University of British Columbia, Vancouver, Canada; Paulette Koronkevich, University of British Columbia, Vancouver, Canada, pletrec@cs.ubc.ca; William J. Bowman, University of British Columbia, Vancouver, Canada, wjb@williamjbowman.com.

ACM 2475-1421/2025/4-ART103
https://doi.org/10.1145/3720437

Type-preserving compilation presents a lightweight alternative to compiler verification that can substantially reduce compiler bugs in practice. A type-preserving compiler uses typed intermediate languages (ILs) as the target of compiler passes and optimizations, and preserves the type annotations from source to target [Morrisett et al. 1999; Tarditi et al. 1996].

Type checking in a typed IL is significantly easier than writing a compiler correctness proof, but provides many guarantees. Common typed ILs guarantee type-and-memory safety of the output of any pass or optimization, so long as the typed IL is safe. Wasm [Haas et al. 2017], a typed IL, guarantees type and memory safety from any source language, although in general that means a program that may have "worked" (with silent vulnerabilities) may raise an error in Wasm. Other popular examples of type-preserving compilers include Java (to typed JVM bytecode), .NET languages (to the typed .NET Common Intermediate Language), Haskell (to the strongly typed Core). Even without a compiler, these typed ILs are useful to ensuring safe linking and execution without trusting the original source or compiler. Handwritten components can have annotations attached, and checked, or compilers can emit wrappers that type check or raise an explicit error.

While a type-preserving compiler provides fewer guarantees than a fully verified compiler in theory, in practice they can be very useful at preventing compilation bugs. From the GHC trenches, Peyton Jones [1996] reports "Maintaining types is a big win. ...Perhaps the largest single benefit came from an unexpected quarter: it is very easy to check a Core program for type correctness. Now most bogus transformations are identified much earlier, and much more precisely."

A key problem in type-preserving compilation is designing a type system for a low-level IL that is (1) safe and (2) expressive enough to enable sophisticated optimizations.

Unfortunately, most type-preserving compilers and typed ILs restrict many optimizations. The example we are concerned with is closure conversion. Common type-preserving closure conversion translations guarantee correctness and even security, but heavily restrict low-level optimizations on which practical compilers for functional languages rely [Ahmed and Blume 2008; Bowman and Ahmed 2018; Kovács 2018; Minamide et al. 1996a,b; Morrisett et al. 1999; New et al. 2016]. This is a fundamental limitation to practical implementation of type-preserving compilation for functional languages.

In this paper, we present a novel typed IL, Flat-Closure Calculus (FCC), for expressing closure conversion and closure optimization. FCC is an important and novel advance on the state-of-the-art by providing a model for efficient typed closure conversion and optimization. We demonstrate how various optimizations found in practice work in FCC. We also discuss the requirements necessary to enable these optimizations, and how FCC implements them.

In short our contributions are:

(1) An analysis of the elements required to express efficient closure representation and optimization (Section 2). This is important for understanding the requirements of a typed IL for closure conversion, and clarifying our design decisions.
(2) The novel IL FCC realizing those requirements, enabling efficient type safe closure conversion and optimization (Section 3). This provides evidence that we've identified correct design elements, and, importantly, that they can be realized.
(3) A proof of type safety and subject reduction for FCC (Subsection 3.1), which provides evidence that FCC does guarantee correctness properties of closure conversion and closure optimizations. Type safety is important for ensuring lightweight correctness of compiled output through type checking. Subject reduction is important for ensuring correctness of various optimizations that can be modelled by reduction, such as inlining.

(4) A translation from abstract closures to FCC (Section 4). This provides evidence that FCC can be targeted by existing type-preserving compilers, and is important for integrating FCC into past work based on abstract closure conversion.
(5) An implementation of standard optimizations for FCC (Section 5), in particular, the (1) (mutually) recursive closure optimization [Keep et al. 2012; Morrisett and Harper 1998] and (2) known and well known procedure optimizations [Steele Jr. 1978]. This provides evidence that FCC is expressive enough to implement and optimize flat closures, and validate the correctness of optimization.

We also provide a Redex [Felleisen et al. 2009; Klein et al. 2012] model of FCC, and implementation of the optimizations in Section 5 in the anonymous supplementary materials.

2 Main Ideas

We distinguish two separate notions of a closure necessary for typed ILs: a *representation* and an *abstraction*. The representation is the underlying data structure. A simple representation could be a pair of the code pointer and a list of the values of free variables as the environment. A closure abstraction hides the elements and size of the environment, ensuring functions of the same type (but different environments) remain the same type after closure conversion. For example, consider a higher-order function $f : (Nat \to Nat) \to B$, and two functions $g = \lambda x.x$ and $h = \lambda x.x + y$ of type $Nat \to Nat$. Both $f\,g$ and $f\,h$ are well typed. However, if we expose the type of the environment, as in the representation of a closure, then what should the type of f be? We must be able to hide the environment to ensure both $f\,g$ and $f\,h$ remain well typed. An efficient, typed closure abstraction of the flat closure representation is one of the main goals of our IL design.

Requirement 1: A typed closure abstraction must hide the type of the environment.

In typical type-preserving compilers, the closure representation is a pair of a code pointer and an environment. The standard abstraction uses an existential type, assigning a function of type $A \to B$ the closure type $\exists\alpha.\langle A \to \alpha \to B, \alpha\rangle$. The environment of type α is paired with the code pointer of type $A \to \alpha \to B$, which now expects the original parameter and the environment. The existential type keeps the type of the environment abstract, and does not expose the elements or size of the environment.

A closure abstraction must also enforce a *calling convention* that ensures a code pointer is called with the expected environment. The expected environment is the one packaged with the code pointer in a closure representation. Calling a code pointer with the incorrect environment gives different behavior than what is expected from the source function, where the correct environment is enforced and implicit by scope.

Requirement 2: A typed closure abstraction must enforce the calling convention: the code pointer is called with its packaged environment.

The standard abstraction, using an existential type $\exists\alpha.\langle A \to \alpha \to B, \alpha\rangle$, follows this requirement. The code pointer expects the abstract type α as the environment, and the only expression of type α is the environment packaged in the pair with the code pointer.

Practical compilers typically use a flat, also called display, closure [Appel 2006, Chapter 10], discovered by Cardelli [1984]. A flat closure is a record[1], a contiguous in-memory data structure with constant access time to each field, containing the code pointer and the values of all the free variables of the original function, *i.e.,* the environment of the closure. This representation is safe-for-space [Shao and Appel 1994], each free variable is accessible with a single memory indirect, and the cost of allocation is proportional to the number of free variables.

[1]Not a record with unordered labels, but an *n-tuple*. We use the terminology *record* for consistency with past work.

As an example, we might represent a flat closure of the function $\lambda x.x + y$ as $c = \{f, y\}$, where f is the code pointer, and y is the free variable included in the environment. To call a flat closure on argument e_1, we project out the code pointer and reuse the closure itself as the environment, as in $c[0]\ c\ e_1$. In the body of the now-closed procedure f we project out the free variable from the closure parameter: $f = \lambda c\ x.x + c[1]$.

> *Requirement 3:* A typed closure abstraction should support the efficient flat closure representation and introduce no overhead.

The existential type abstraction introduces no overhead for the pair representation. To call a closure c (represented as a pair) on an expression e_1, we project both the code pointer and environment, and then apply the code pointer to the environment: $\mathbf{unpack}\langle \alpha, p\rangle = c\ \mathbf{in}\ ((\mathbf{fst}\ p)\ (\mathbf{snd}\ p)\ e_1)$. The **unpack** form is computationally irrelevant, so the calling convention for the pair representation has no overhead caused by the type abstraction.

However, the existential type abstraction is not suitable for a flat closure representation. The type variable abstracts over an entire object, the environment, and cannot support abstracting over *part* of an object, *e.g.,* the part of the record that is the environment. Additionally, the type of the code pointer does not support passing the closure itself as the environment.

Keep et al. [2012] provides several sets of closure optimizations over flat closures implemented in the high-performance Chez Scheme compiler. Supporting these sets of optimizations with our typed representation and abstraction is one of our goals.

The first set of optimizations are *known procedure optimizations*[2]. These optimizations require the ability to call a procedure, *i.e.,* the code pointer of a closure, directly, rather than only through a closure. This saves a memory indirect when the code pointer of a closure is known at the call site. For recursive closures, this enables recursion *directly* through code pointers, in addition to *indirectly* through the closure.

> *Requirement 4:* Procedures (including recursive) are directly callable through their code pointer.

The existential type closure abstraction supports this optimization without any changes. However, abstract closure conversion, as presented by Minamide et al. [1996a] and Bowman and Ahmed [2018], does not support calling a procedure directly through its code pointer.

The second set of optimizations are *well known procedure optimizations*. When a procedure is known at all of its possible call sites (*i.e.,* well known), we can break the closure abstraction, and optimize the underlying representation. A well known procedure can always be called directly, so we can remove the code pointer from its closure. We can change the representation of the closure; for example, when there are no free variables, we can eliminate the closure and environment entirely. These optimizations require that procedure signatures be flexible.

> *Requirement 5:* Procedures must have unrestricted signatures.

Past work like the existential type abstraction of closures, *mostly* supports these optimizations. However, since the existential type quantifies over an entire object, and entire parameter, it's difficult to entirely eliminate the environment parameter. Since abstract closure conversion does not support calling a procedure directly through its code pointer, these optimizations are not immediately possible in current presentations of abstract closure conversion.

The third set of optimizations eliminate unnecessary free variables. These optimizations must be supported in the IL, as optimizing closures may cause variables to become unnecessary. Removing

[2]Keep et al. [2012] don't consider this separate from well known procedure optimization, but we do as each optimization imposes different requirements on the IL.

the free variables from a closure changes its representation, but not its abstraction. Thus, these optimizations rely (in part) on Requirement 1 and Requirement 5. Most optimizations in this set identify unnecessary free variables through constant propagation and aliasing. We model these as partial evaluation in our IL, which means our IL must satisfy subject reduction: a term of type A must still have type A after reduction.

Requirement 6: The IL must satisfy subject reduction.

All prior work on type-preserving closure conversion supports this requirement.

The fourth and last set of optimizations involve optimizations for sets of mutually recursive closures. Many of these introduce no new requirements: we need the ability to change the signature of well-known procedures (per Requirement 5), and to directly call known procedures (per Requirement 4). However, there is one subtlety in the typed representation. Any escaping recursive closure, such as a reference to a closure in a mutually recursive set or a self reference passed to a higher-order function, must not reallocate the closure. Instead, the reference to the recursive closure in scope should be used—either the closure argument to the procedure, or the closure bound in a mutually recursive set. This is a special case of Requirement 3, but we distinguish it in a new requirement as it has been studied separately in past work [Morrisett and Harper 1998].

Requirement 7: A typed closure representation must enable (mutually) recursive calls without reallocating the closure.

Morrisett and Harper [1998] analyze three alternatives to representing typed recursive closures, but each violates one of our requirements. The first alternative uses a recursive procedure definition, allowing a procedure to recur directly through its own label. This satisfies Requirement 4, but violates Requirement 7. Because the procedure takes its *environment*, rather than its *closure*, the closure must be reconstructed if it escapes or is used in a mutually recursive call. The second alternative uses recursive closures, so a closure is part of its own environment. However, this violates Requirement 4, since all recursion must go through the environment. This also changes the closure abstraction to require accessing the self reference indirectly through the environment, introducing an extra memory indirect, violating Requirement 3. Keep et al. [2012] also note that this self reference is unnecessary as a free variable, so this adds additional allocation overhead (violating Requirement 3, again). The third alternative uses a recursive type, so a closure's type is a recursive existential type, allowing the closure itself to be passed as an argument, instead of the environment. This is close to the flat closure representation and satisfies Requirement 7. However, it still requires all recursion to go indirectly through the closure, violating Requirement 4.

The state of the art presents us with two choices: an efficient representation with an unenforcable abstraction, or an enforcable abstraction with an inefficient representation.

2.1 A Novel IL

The existential type closure abstraction does not meet our requirements. In our IL, FCC, we develop an abstract closure abstraction and typed representation that meets all the requirements.

The representation of a flat closure in FCC has type $\mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \overrightarrow{A}\rightarrow B), \overrightarrow{C})$. The first field of the record is the code pointer to the closure's procedure. The rest of the record contains environment values of types $\overrightarrow{C}$. The code pointer accepts the closure itself, so we use a recursive type to include a self reference α in the code pointer's signature.

The closure abstraction in FCC has the type $\mathsf{Clos}(\overrightarrow{A}\rightarrow B)$, which only exposes the parameter and output type of the code pointer, but hides the type of the environment satisfying Requirement 1. We create an abstract closure from a closure representation using the clos type cast with the following typing rule.

$$\frac{\Gamma \vdash e : \mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \vec{A} \to B), \vec{C})}{\Gamma \vdash \mathsf{clos}(e) : \mathsf{Clos}(\vec{A} {\to} B)}$$

Unlike past work, we want this abstract closure to support projecting out the code pointer, rather than only application to the remaining arguments. This is related to Requirement 3 and Requirement 4; we want clos to be erasable, and the projection from it should be replaceable by the code pointer. This requires an operation on $cl : \mathsf{Clos}(\vec{A}{\to}B)$ like $\pi_c\ cl : \mathsf{Code}(?, \vec{A}{\to}B)$, but what should ? be?

We use a *singleton type* to enforce the correct calling convention, per Requirement 2. There is always exactly *one* environment that is safe to pass to the code pointer: the closure itself. This matches the in-practice use of flat closures, and the singleton type lets us express this pattern. A singleton type is indexed by a value, which is the only value inhabiting the type. The simplified typing rule below demonstrates how singleton types are introduced.

$$\frac{\Gamma \vdash v : A}{\Gamma \vdash \mathsf{is}(v) : \mathsf{The}(v)}$$

Now, when we project the code pointer from a closure $\mathsf{Clos}(\vec{A}{\to}B)$, we change the signature of the code pointer to only accept the closure as its environment argument. We show a simplified typing rule below.

$$\frac{\Gamma \vdash cl : \mathsf{Clos}(\vec{A}{\to}B)}{\Gamma \vdash \pi_c\ cl : \mathsf{Code}(\mathsf{The}(cl), \vec{A}{\to}B)}$$

For example, a closure cl is applied to an argument v as $\mathsf{apply}\ (\pi_c\ cl)\ (\mathsf{is}(cl))\ v$.

Unfortunately, there's a problem: the projection $\pi_c\ cl$ has a different type than the underlying code pointer. This violates requirements Requirement 6 and Requirement 4; reduction or known procedure optimizations will result in terms of different types. The concrete code pointer type expects a *record*, the representation of a closure, not an abstract *closure* nor a singleton of a closure. Consider a closure $cl = \mathsf{clos}(\mathsf{rec}(f, v_1, v_2))$. If we project out the code pointer $g = \pi_c\ cl$ then g expects $\mathsf{is}(cl)$ as the environment. However, g is reduced to some procedure f that expects the record $\mathsf{rec}(f, v_1, v_2)$ and not the singleton $\mathsf{is}(cl)$.

We add the type cast accept-clos to FCC to resolve this discrepancy. Applied to a code pointer f, accept-clos modifies the type signature to expect a singleton of a closure instead of the underlying record. The type cast accept-clos checks against any closure cl with a valid representation type.

$$\frac{\Gamma \vdash f : \mathsf{Code}(\mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \vec{A}{\to}B), \vec{C}), \vec{A}{\to}B) \qquad \Gamma \vdash cl : \mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \vec{A}{\to}B), \vec{C})}{\Gamma \vdash \mathsf{accept\text{-}clos}(f) : \mathsf{Code}(\mathsf{The}(cl), \vec{A}{\to}B)}$$

The expectation is that accept-clos is only introduced during certain intra-language optimizations and evaluation. accept-clos enables the code pointer to have the same type as projecting the code pointer from a closure, supporting local type-preserving optimizations. The reduction rule for projecting from a closure demonstrates how the cast is introduced during evaluation.

$$\pi_c\ \mathsf{clos}(\mathsf{rec}(f, \vec{v})) \hookrightarrow \mathsf{accept\text{-}clos}(f)$$

The accept-clos cast is computationally irrelevant, though the cast must be eliminated simultaneously with other type casts in the IL. The reduction rule for an indirect procedures call through a closure with accept-clos demonstrates this:

$$\mathsf{apply}\ \mathsf{accept\text{-}clos}(f)\ \mathsf{is}(\mathsf{clos}(cl))\ \vec{v} \hookrightarrow e[x_0 \mapsto cl][\vec{x} \mapsto \vec{v}]$$

$$n \in \mathbb{N} \quad x, l \in \mathit{Variable} \quad \alpha \in \mathit{TypeVariable}$$

$$\begin{array}{rcl}
A, B, C & ::= & \alpha \mid \mathsf{Nat} \mid \mathsf{Clos}(\vec{A} \to B) \mid \mathsf{Code}(\vec{A} \to B) \mid \mathsf{Rec}(\vec{A}) \mid \mathsf{The}(v : A) \mid \mu\alpha.A \\
v & ::= & x \mid l \mid n \mid \mathsf{rec}(\vec{v}) \mid \mathsf{clos}(v) \mid \mathsf{accept\text{-}clos}(v) \mid \mathsf{is}(v) \mid \mathsf{fold}(v) \\
e & ::= & v \mid \mathsf{apply}\ e\ \vec{e} \mid \mathsf{rec}(\vec{e}) \mid \pi_n\ e \mid \pi_c\ v \mid \mathsf{clos}(e) \mid \mathsf{fold}(e) \mid \mathsf{unfold}(e) \\
 & & \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{cletrec}\ \overrightarrow{x = \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\vec{v})))}\ \mathsf{in}\ e
\end{array}$$

$$\begin{array}{rcl}
f & ::= & \lambda \overrightarrow{x : A}.e \\
p & ::= & \mathsf{letrec}\ \overrightarrow{x = f}\ \mathsf{in}\ e
\end{array}
\qquad
\begin{array}{rcl}
\Gamma & ::= & \emptyset \mid \Gamma, x : A \\
\Sigma & ::= & \emptyset \mid \Sigma, \alpha \\
\Delta & ::= & \emptyset \mid \Delta, x : \mathsf{Code}(\vec{A} \to B)
\end{array}$$

Fig. 1. FCC Syntax

By contrast, FCC also includes the following reduction rule for a direct procedure call.

$$\mathsf{apply}\ f\ \vec{v} \hookrightarrow e[\vec{x} \mapsto \vec{v}]$$

In both, the body e of the procedure f is called with the parameters substituted by arguments $\vec{v}$. However, the reduction rule for applying a procedure with accept-clos removes the is constructor and clos type cast on the underlying representation of the closure cl.

After erasing the type casts, the reduction rules are equal. This satisfies Requirement 3, that the abstraction introduces no overhead, and supports applying procedures directly with no requirements on the procedure signature, satisfying Requirement 5. We use these type casts to keep the abstraction boundary of closures, while still using the efficient representation effectively.

FCC includes two forms of recursive function: indirectly, through closures via recursive types, and directly, through letrec. The former is used to satisfy Requirement 7. A recursive closure call would look something like: $f = \lambda env.\lambda arg.(\ldots\ \mathsf{apply}\ (\pi_c\ env)\ env \ldots)$, where the recursive code pointer is projected directly from the environment parameter. Without the recursive type, env could not be passed directly, and would have to be reconstructed (reallocated) from f and the rest of the environment. Similar situations arise if the recursive reference escapes, or in a mutually recursive reference. letrec is used to satisfy Requirement 4. To optimize the recursive call to a direct call, such as in $f = \lambda env.\lambda arg.(\ldots f\ env \ldots)$, we also require direct (mutual) recursion through the code pointer f. This requires f to be bound in a (mutually) recursive block via letrec.

3 The FCC Intermediate Language

The syntax of FCC is shown in Figure 1. A program p is a set of mutually recursive top-level procedure definitions, followed by a final expression e. Values v are either variables, run-time values, or type casts on values. Expressions e include standard introduction and elimination forms for isorecursive types (fold and unfold), records (rec and $\pi_n\ e$), and a let binding form. Closures are introduced using clos, or cletrec for mutually defined closures, and code pointers are projected from closures with the $\pi_c\ v$ form. A code pointer is applied to a sequence of arguments using apply.

Most types are standard, such as a type Nat for natural numbers, Rec for record types, and $\mu\alpha.A$ for recursive types. The type $\mathsf{Code}(\vec{A} \to B)$ represents the type of closed procedures, where $\vec{A}$ represents the sequence of input types and B the output type. The closure abstraction is given the type $\mathsf{Clos}(\vec{A} \to B)$. Finally, the type $\mathsf{The}(v : A)$ is a singleton of the value v of type A. We syntactically restrict singletons to values to avoid deciding equality between expressions while type checking.

The lexical typing environment Γ maps variables to types. The Σ is the set of free type variables. The global procedure environment Δ maps variables to procedure types.

Program Typing. The judgement $\vdash p : A$ defined in Figure 2 checks the type of a top-level program. A program is well typed if all procedure definitions are well typed under the procedure

$$\boxed{\vdash p : A}$$

$$\textsc{Program}\ \frac{\emptyset;\Delta \vdash e : C \qquad \Delta = \overrightarrow{x : \mathsf{Code}(\vec{A}\to B)} \qquad \overrightarrow{\Delta \vdash f : \mathsf{Code}(\vec{A}\to B)}}{\vdash \mathsf{letrec}\ \overrightarrow{x = f}\ \mathsf{in}\ e : C}$$

$$\boxed{\Delta \vdash f : A}$$

$$\textsc{Code}\ \frac{\overrightarrow{x : A};\Delta \vdash e : B \qquad \emptyset;\emptyset;\emptyset \vdash \mathsf{Code}(\vec{A}\to B)}{\Delta \vdash \lambda\overrightarrow{x : A}.e : \mathsf{Code}(\vec{A}\to B)}$$

Fig. 2. FCC Program Typing

$$\boxed{\Gamma;\Delta \vdash e : A}$$

$$\textsc{Nat}\ \frac{\emptyset;\Delta \vdash \Gamma}{\Gamma;\Delta \vdash n : \mathsf{Nat}} \qquad \textsc{Var}\ \frac{\emptyset;\Delta \vdash \Gamma \qquad x : A \in \Gamma}{\Gamma;\Delta \vdash x : A} \qquad \textsc{Proc}\ \frac{\emptyset;\Delta \vdash \Gamma \qquad x : \mathsf{Code}(\vec{A}\to B) \in \Delta}{\Gamma;\Delta \vdash x : \mathsf{Code}(\vec{A}\to B)}$$

$$\textsc{Rec}\ \frac{\overrightarrow{\Gamma;\Delta \vdash e : A}}{\Gamma;\Delta \vdash \mathsf{rec}(\vec{e}) : \mathsf{Rec}(\vec{A})} \qquad \textsc{Proj}\ \frac{\Gamma;\Delta \vdash e : \mathsf{Rec}(\vec{A})}{\Gamma;\Delta \vdash \pi_i\ e : A_i} \qquad \textsc{The}\ \frac{\Gamma;\Delta \vdash v : A}{\Gamma;\Delta \vdash \mathsf{is}(v) : \mathsf{The}(v : A)}$$

$$\textsc{CLetrec}\ \frac{\overrightarrow{\Gamma,\overrightarrow{x : A};\Delta \vdash \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\vec{v}))) : A} \qquad \Gamma,\overrightarrow{x : A};\Delta \vdash e : B \qquad \overrightarrow{\Gamma;\emptyset;\Delta \vdash A} \qquad \Gamma;\emptyset;\Delta \vdash B}{\Gamma;\Delta \vdash \mathsf{cletrec}\ \overrightarrow{x = \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\vec{v})))}\ \mathsf{in}\ e : B}$$

$$\textsc{Let}\ \frac{\Gamma;\Delta \vdash e_1 : A \qquad \Gamma,\ x : A;\Delta \vdash e_2 : B \qquad \Gamma;\emptyset;\Delta \vdash B}{\Gamma;\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : B} \qquad \textsc{Fold}\ \frac{\Gamma;\Delta \vdash e : A[\alpha \mapsto \mu\alpha.A]}{\Gamma;\Delta \vdash \mathsf{fold}(e) : \mu\alpha.A}$$

$$\textsc{Unfold}\ \frac{\Gamma;\Delta \vdash e : \mu\alpha.A}{\Gamma;\Delta \vdash \mathsf{unfold}(e) : A[\alpha \mapsto \mu\alpha.A]} \qquad \begin{array}{l}\textsc{Apply}\\ \dfrac{\Gamma;\Delta \vdash f : \mathsf{Code}(\vec{A}\to B) \qquad \overrightarrow{\Gamma;\Delta \vdash e : A}}{\Gamma;\Delta \vdash \mathsf{apply}\ f\ \vec{e} : B}\end{array}$$

$$\textsc{Decl-Clos}\ \frac{\Gamma;\Delta \vdash e : \mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \vec{A}\to B), \vec{C}) \qquad \Gamma;\emptyset;\Delta \vdash \mathsf{Clos}(\vec{A}\to B)}{\Gamma;\Delta \vdash \mathsf{clos}(e) : \mathsf{Clos}(\vec{A}\to B)}$$

$$\textsc{Clos-Proj}\ \frac{\Gamma;\Delta \vdash v : \mathsf{Clos}(\vec{A}\to B)}{\Gamma;\Delta \vdash \pi_c\ v : \mathsf{Code}(\mathsf{The}(v : \mathsf{Clos}(\vec{A}\to B)), \vec{A}\to B)}$$

$$\textsc{Accept-Clos}\ \frac{\begin{array}{c}\Gamma;\Delta \vdash v_1 : \mathsf{Code}(\mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \vec{A}\to B), \vec{C}), \vec{A}\to B)\\ \Gamma;\Delta \vdash v_2 : \mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \vec{A}\to B), \vec{C})\end{array}}{\Gamma;\Delta \vdash \mathsf{accept\text{-}clos}(v_1) : \mathsf{Code}(\mathsf{The}(\mathsf{clos}(v_2) : \mathsf{Clos}(\vec{A}\to B)), \vec{A}\to B)}$$

Fig. 3. FCC Expression Typing

environment Δ. Procedure types must be well formed under empty environments, which restricts the use of a dependent type at a top level. The body e is checked under the same procedure environment Δ and the empty lexical type environment $\emptyset$.

$\boxed{\Gamma;\Sigma;\Delta \vdash A}$

$$\textsc{WFNat}\ \frac{}{\Gamma;\Sigma;\Delta \vdash \mathsf{Nat}} \qquad \textsc{WFClos}\ \frac{\overrightarrow{\Gamma;\Sigma;\Delta \vdash A} \qquad \Gamma;\Sigma;\Delta \vdash B}{\Gamma;\Sigma;\Delta \vdash \mathsf{Clos}(\vec{A}\to B)} \qquad \textsc{WFCode}\ \frac{\overrightarrow{\Gamma;\Sigma;\Delta \vdash A} \qquad \Gamma;\Sigma;\Delta \vdash B}{\Gamma;\Sigma;\Delta \vdash \mathsf{Code}(\vec{A}\to B)}$$

$$\overset{\textsc{WFRec}}{\frac{\overrightarrow{\Gamma;\Sigma;\Delta \vdash A}}{\Gamma;\Sigma;\Delta \vdash \mathsf{Rec}(\vec{A})}} \qquad \overset{\textsc{WFThe}}{\frac{\Gamma;\Delta \vdash v : A}{\Gamma;\Sigma;\Delta \vdash \mathsf{The}(v : A)}} \qquad \overset{\textsc{WFMu}}{\frac{\Gamma;\alpha,\Sigma;\Delta \vdash A}{\Gamma;\Sigma;\Delta \vdash \mu\alpha.A}} \qquad \overset{\textsc{WFAlpha}}{\frac{\alpha \in \Sigma}{\Gamma;\Sigma;\Delta \vdash \alpha}}$$

Fig. 4. FCC Well-Formed Types

$\boxed{\Sigma;\Delta \vdash \Gamma}\ \boxed{\vdash \Delta}$

$$\overset{\textsc{WFEmptyG}}{\frac{\vdash \Delta}{\Sigma;\Delta \vdash \emptyset}} \qquad \overset{\textsc{WFGamma}}{\frac{\Sigma;\Delta \vdash \Gamma \qquad \Gamma;\Sigma;\Delta \vdash A \qquad x \notin \Gamma}{\Sigma;\Delta \vdash \Gamma, x : A}}$$

$$\overset{\textsc{WFDelta}}{\frac{\Delta = \overrightarrow{x : \mathsf{Code}(\vec{A_i}\to B_i)} \qquad \overrightarrow{\emptyset;\emptyset;\emptyset \vdash \mathsf{Code}(\vec{A}\to B)}}{\vdash \Delta}}$$

Fig. 5. FCC Well-Formed Type and Procedure Environments

The judgement $\Delta \vdash f : A$, also in Figure 2, checks top-level procedures. The procedure body e is checked with only its parameters parameters $\vec{x}$ of types $\vec{A}$ in the lexical environment, ensuring the procedure is closed. Because singleton types may refer to variables, we require the procedure type $\mathsf{Code}(\vec{A}\to B)$ to be well formed to prevent the types $\vec{A}$ and output type B referring to variables out of scope. At the top-level, procedure types must be well formed in an empty context to avoid some cyclic well formedness checking, since the singleton type (a dependent type) causes well formedness and well typedness to be mutually defined.

Expression Typing. The typing judgement for expressions $\Gamma;\Delta \vdash e : A$ is defined in Figure 3 and states that e has type A under the type environment Γ and procedure environment Δ. Many of the typing rules are essentially standard. Rule Apply is standard for a closure-converted language, and is essentially similar to typing standard function application. Rule Fold and Rule Unfold are the standard rules for isorecursive types [Pierce 2002], where the type variable α is substituted with the recursive type. Rule Proc is the equivalent of Rule Var for procedure variables, whose types are in the procedure environment Δ. Rule Nat, Rule Var and Rule Proc each require the type environment Γ to be well formed. Mutually defined closures are typed using Rule CLetrec, which checks both the body e and the binding expressions $\overrightarrow{x = \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\vec{v})))}$ with types $\vec{A}$ under a type environment binding all of $\overrightarrow{x : A}$. A singleton type $\mathsf{The}(v : A)$ is introduced using the is constructor in Rule The, requiring the value v to be well typed.

Rule Decl-Clos ensures abstract closures $\mathsf{Clos}(\vec{A}\to B)$ are created from the correct recursive closure representation type. Rule Clos-Proj casts the procedure projected from a closure v to expect a singleton of the same closure $\mathsf{The}(v : \mathsf{Clos}(\vec{A}\to B))$. Finally, in Rule Accept-Clos, accept-clos modifies a procedure v_1 that accepts the record representation of the closure to one that accepts a singleton of an abstract closure constructed from that representation v_2. The value v_2 is checked to ensure the correct closure representation type.

$$\boxed{\delta \vdash \langle \psi \mid e \rangle \hookrightarrow \langle \psi' \mid e' \rangle}$$

$$\begin{array}{rl}
\delta \vdash \langle \psi \mid \pi_i \ \mathsf{rec}(\vec{v}) \rangle \hookrightarrow_{\pi} & \langle \psi \mid v_i \rangle \\
\delta \vdash \langle \psi \mid \pi_c \ \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\vec{v}))) \rangle \hookrightarrow_{\pi-clos} & \langle \psi \mid \mathsf{accept\text{-}clos}(v_0) \rangle \\
\delta \vdash \langle \psi \mid \mathsf{apply}\ x_f \ \vec{v} \rangle \hookrightarrow_{\beta} & \langle \psi \mid \hat{e}[\overrightarrow{x \mapsto v}] \rangle \\
 & \text{where } x_f = \lambda \overrightarrow{x : A}.\hat{e} \in \delta \\
\delta \vdash \langle \psi \mid \mathsf{apply}\ \mathsf{accept\text{-}clos}(x_f)\ \mathsf{is}(\mathsf{clos}(v_0))\ \vec{v} \rangle \hookrightarrow_{\beta-clos} & \langle \psi \mid \hat{e}[x_0 \mapsto v_0][\overrightarrow{x \mapsto v}] \rangle \\
 & \text{where } x_f = \lambda x_0 : \_, \overrightarrow{x : A}.\hat{e} \in \delta \\
\delta \vdash \langle \psi \mid \mathsf{let}\ x = v \ \mathsf{in}\ e \rangle \hookrightarrow_{\zeta} & \langle \psi \mid e[x \mapsto v] \rangle \\
\delta \vdash \langle \psi \mid \mathsf{cletrec}\ \overrightarrow{x = cl}\ \mathsf{in}\ e \rangle \hookrightarrow_{\zeta-clos} & \langle \psi, \overrightarrow{l \mapsto cl[\overrightarrow{x \mapsto l}]} \mid e[\overrightarrow{x \mapsto l}] \rangle \\
 & \text{where } \overrightarrow{cl = \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\vec{v})))} \\
\delta \vdash \langle \psi, l \mapsto cl \mid l \rangle \hookrightarrow_{\alpha} & \langle \psi, l \mapsto cl \mid cl \rangle \\
 & \text{where } cl = \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\vec{v}))) \\
\delta \vdash \langle \psi \mid \mathsf{unfold}(\mathsf{fold}(v)) \rangle \hookrightarrow_{\gamma} & \langle \psi \mid v \rangle
\end{array}$$

$$\textsc{StepCtx}\ \frac{\delta \vdash \langle \varnothing \mid e \rangle \hookrightarrow \langle \psi \mid e' \rangle}{\delta \vdash \langle \psi \mid E[e] \rangle \hookrightarrow_{\kappa} \langle \psi' \mid E[e'] \rangle}$$

$$\begin{array}{rcl}
E & ::= & [\cdot] \mid \mathsf{apply}\ E\ \vec{e} \mid \mathsf{apply}\ v\ \vec{v}, E, \vec{e} \mid \mathsf{rec}(\vec{v}, E, \vec{e}) \mid \pi_n\ E \mid \pi_c\ E \\
 & & \mid \mathsf{fold}(E) \mid \mathsf{unfold}(E) \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \\
\delta & ::= & \varnothing \mid \delta, x = f \\
\psi & ::= & \varnothing \mid \psi, l \mapsto \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\vec{v})))
\end{array}$$

Fig. 6. FCC Small-Step Reduction

Well Formed Types and Environments. The judgement $\Gamma; \Sigma; \Delta \vdash A$ is defined in Figure 4 to check types are well formed. All the rules are essentially standard. Rule WFThe requires that the value argument v is well typed at the annotated type A, so well formed types are defined mutually with the typing of expressions. The type annotation in The syntactically prevents cyclic derivations.

Well formed types rely on well formed type environments Γ, defined by the judgement $\Sigma; \Delta \vdash \Gamma$ in Figure 5. Rule WFGamma ensures that each binding $x : A \in \Gamma$ has a well formed type A, given all preceding bindings in Γ. By Rule WFEmptyG, Γ is well formed only if the procedure environment is by $\vdash \Delta$. The judgement $\vdash \Delta$ ensures all procedures have well formed types under empty environments.

Reduction Rules. The reduction rules for FCC are given in Figure 6, presented as a small-step semantics under an evaluation context. The reduction rules $\delta \vdash \langle \psi \mid e \rangle \hookrightarrow \langle \psi' \mid e' \rangle$ use a static heap δ, which maps variables to procedures, and a dynamic heap ψ, which maps labels to closures. Under a static heap δ, the expression e with dynamic heap ψ reduces to an expression e' with a new heap ψ'.

Many of the reduction rules are essentially standard, so we focus on the reductions Rule $\hookrightarrow_{\pi-clos}$ and Rule $\hookrightarrow_{\beta-clos}$ unique to FCC. Projecting the code pointer from a closure wraps a type cast accept-clos around the concrete code pointer v_0. Recall from Subsection 2.1 that this type cast soothes the type discrepancy that arises from the abstract closure projection operation producing a code pointer with a concrete environment type. Applying a closure is similar to the standard procedure application rule Rule $\hookrightarrow_{\beta}$, except the type casts are erased ensuring the closure representation v_0 is substituted for the first parameter x_0. The type casts must be erased, as we note in the proof of Subject Reduction (Lemma 3.4), since the procedure body is typed expecting a record not an abstract closure.

$\boxed{\Delta \vdash \psi : \Gamma}$ $\boxed{\vdash \delta : \Delta}$

$$\textsc{Heap}\quad \frac{\psi = \overrightarrow{l \mapsto v} \qquad \Gamma = \overrightarrow{l : A} \qquad \overrightarrow{\Gamma; \Delta \vdash v : A}}{\Delta \vdash \psi : \Gamma}$$

$$\textsc{StaticCode}\quad \frac{\delta = \overrightarrow{x = f} \qquad \Delta = \overrightarrow{x : \mathsf{Code}(\vec{A} \to B)} \qquad \overrightarrow{\Delta \vdash f : \mathsf{Code}(\vec{A} \to B)}}{\vdash \delta : \Delta}$$

Fig. 7. FCC Run-Time Typing

3.1 Metatheory

We prove type safety of FCC, stated formally below, which states that all well typed programs in FCC either either step to a value or diverge, proving an absence of type, memory, and undefinedness errors. The proof follows the standard Progress (Lemma 3.2) and Subject Reduction (Lemma 3.4) lemmas [Harper 1996; Wright and Felleisen 1994].

Theorem 3.1 (Type Safety). *If* $\vdash p : C$, *then either* $\emptyset \vdash \langle \emptyset \mid p \rangle \hookrightarrow_k \langle \psi \mid v \rangle$ *or diverges.*

Progress and Subject Reduction require relating the static and dynamic heaps to the typing environments these heaps represent, formalized in Figure 7. The judgement $\vdash \delta : \Delta$ checks the static heap δ against the types in procedure environment Δ, and $\Delta \vdash \psi : \Gamma$ checks the dynamic heap ψ against the types in the typing environment Γ.

Progress guarantees that a well typed expression is already a value or steps to another expression under well typed heaps; note that it says nothing about whether the result is well typed.

Lemma 3.2 (Progress). *If* $\Gamma; \Delta \vdash e : C$, *then either* e *is a value or* $\delta \vdash \langle \psi \mid e \rangle \hookrightarrow \langle \psi' \mid e' \rangle$ *where* $\vdash \delta : \Delta$, *and* $\Delta \vdash \psi : \Gamma$.

Proof. The proof is straightforward by induction on the typing judgement $\Gamma; \Delta \vdash e : C$. We present the cases for applying and projecting from a closure, as these involve our new reduction rules for FCC. Both cases are expressions, so it suffices to show they always take a step.

Case Rule Apply By induction, either some subterm e_f or any of $\vec{e}$ takes a step, or all are values. If one takes a step, then $\delta \vdash \langle \psi \mid \mathsf{apply}\ e_f\ \vec{e} \rangle \hookrightarrow \langle \psi' \mid e' \rangle$ holds trivially by $\hookrightarrow_k$. Otherwise, since e_f is a value and of type $\mathsf{Code}(\vec{A} \to B)$, we have that e_f is either a variable or a $\mathsf{accept\text{-}clos}(x)$ with x a variable by Lemma 3.3. If e_f is a variable, $(e_f \mapsto f) \in \delta$ and stepping proceeds by $\hookrightarrow_\beta$. Otherwise, e_f is a $\mathsf{accept\text{-}clos}(x)$, and stepping proceeds by $\hookrightarrow_{\beta-clos}$.

Case Rule Accept-Clos By induction, either c steps to another expression c', or c is a value. If c steps, then $\delta \vdash \langle \psi \mid \pi_c\ c \rangle \hookrightarrow \langle \psi' \mid \pi_c\ c' \rangle$ holds trivially by $\hookrightarrow_k$. Otherwise, c is either a label or $\mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\vec{v})))$ for some $\vec{v}$ with $\vec{v}$ values by Lemma 3.3. If c is a label, $(c \mapsto v) \in \psi$ and stepping proceeds by $\hookrightarrow_\alpha$. Otherwise, c is $\mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\vec{v})))$, and stepping proceeds by $\hookrightarrow_{\pi-clos}$. □

Progress (Lemma 3.2) relies on the standard *Canonicity* lemma (Lemma 3.3) to reason about the form of a value from its type. The lemma is non-obvious since we have a dependent type, but it is straightforward since our type equality is based on syntactic equality, which is trivial. We give key cases of the statement here; a full version and its proof are provided in the anonymous supplementary materials.

Lemma 3.3 (Canonicity (excerpts)).

- *If* $\Gamma; \Delta \vdash e : \mathit{Code}(\overrightarrow{A} \rightarrow B)$*,* $e \in v$*, and* $\Delta \vdash \psi : \Gamma$ *then either* $e \in$ *Variable such that* $(e : \mathit{Code}(\overrightarrow{A} \rightarrow B)) \in \Delta$ *or* $e = \mathit{accept\text{-}clos}(x)$ *for some* $x \in$ *Variable.*
- *If* $\Gamma; \Delta \vdash e : \mathit{Clos}(\overrightarrow{A} \rightarrow B)$*,* $e \in v$*, and* $\Delta \vdash \psi : \Gamma$ *then either* $e \in$ *Variable such that* $(e \mapsto v_e) \in \psi$ *or* $e = \mathit{clos}(v_e)$ *for some* v_e *that is a value.*
- *If* $\Gamma; \Delta \vdash e : \mathit{The}(v_e : A)$*,* $e \in v$*, and* $\Delta \vdash \psi : \Gamma$ *then* $e = \mathit{is}(v_e)$ *for some* v_e *that is a value.*

Subject Reduction guarantees if a well typed expressions takes a step, the result is well typed.

Lemma 3.4 (Subject Reduction). *If* $\delta \vdash \langle \psi \mid e \rangle \hookrightarrow \langle \psi' \mid e' \rangle$ *and* $\Gamma; \Delta \vdash e : C$*, where* $\vdash \delta : \Delta$ *and* $\Delta \vdash \psi : \Gamma$*, then* $\Gamma'; \Delta \vdash e' : C$*, where* $\Delta \vdash \psi' : \Gamma'$

Proof. The proof proceeds by induction on the derivation $\delta \vdash \langle \psi \mid e \rangle \hookrightarrow \langle \psi' \mid e' \rangle$. We present the cases for novel reduction rules $\hookrightarrow_{\pi-clos}$ and $\hookrightarrow_{\beta-clos}$, and the standard reduction rule $\hookrightarrow_{\pi}$. Remaining cases are straightforward, and full details are included in the anonymous supplementary materials.

Case Rule $\hookrightarrow_{\pi}$: $\delta \vdash \langle \psi \mid \pi_i \ \mathsf{rec}(\overrightarrow{v}) \rangle \hookrightarrow_{\pi} \langle \psi \mid v_i \rangle$. We must show that v_i has the same type as $\pi_i \ \mathsf{rec}(\overrightarrow{v})$, which follows from inversion on the typing derivation for $\pi_i \ \mathsf{rec}(\overrightarrow{v})$ (Rule Proj).

Case Rule $\hookrightarrow_{\pi-clos}$: $\delta \vdash \langle \psi \mid \pi_c \ \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\overrightarrow{v}))) \rangle \hookrightarrow_{\pi-clos} \langle \psi \mid \mathsf{accept\text{-}clos}(v_0) \rangle$. We must show that the type cast $\mathsf{accept\text{-}clos}(v_0)$ has the same code pointer type as the closure projection, $\mathsf{Code}(\mathsf{The}(\mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\overrightarrow{v}))), \overrightarrow{A}) \rightarrow B)$.

By Rule Accept-Clos, it suffices to show $v_0 : \mathsf{Code}(\mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \overrightarrow{A} \rightarrow B), \overrightarrow{C}), \overrightarrow{A} \rightarrow B)$. This follows by inversion on the typing derivation for $\mathsf{clos}(\mathsf{fold}(\mathsf{rec}(\overrightarrow{v})))$ (Rule Decl-Clos, Rule Fold, Rule Rec), since the first component v_0 of $\overrightarrow{v}$ is a code pointer expecting the representation of a closure as a recursive record.

Case Rule $\hookrightarrow_{\beta-clos}$: $\delta \vdash \langle \psi \mid \mathsf{apply} \ \mathsf{accept\text{-}clos}(x_f) \ \mathsf{is}(\mathsf{clos}(v_0)) \ \overrightarrow{v} \rangle \hookrightarrow \langle \psi \mid \hat{e}[x_0 \mapsto v_0][\overrightarrow{x \mapsto v}] \rangle$. When applying a closure, the type casts around the code pointer and closure are erased. The type cast around the code pointer is no longer necessary when reduction moves to the procedure body. The procedure body $\hat{e}$ expects a record as the environment parameter, so erasing *both* the is and clos around v_0 is necessary. Then, given that the procedure body $\hat{e}$ and all the arguments are well typed (which they are by inversion on the derivations for the original expression and the well typed heaps), we conclude $\hat{e}$ is well typed with the parameters substituted with the arguments by the Substitution Lemma (Lemma 3.5). Note that well formedness of types guarantees that x_0 and $\overrightarrow{x}$ are not bound in the type of the application. □

Subject Reduction usually relies on a standard substitution lemma, as some steps of reduction (*e.g.*, function application) use substitution. Our version of the substitution lemma is stated below, and notice that substitution also occurs in the type T because of our use of the dependent equality type. The proof is straightforward by (mutual) induction on the typing and well formed judgments, and is provided in the anonymous supplementary materials. We also state and prove that the context $\Gamma, \Gamma'[x \mapsto b]$ and type $T[x \mapsto b]$ are well formed after substitution, but this is straightforward due to our well formed context and type judgements and restricted dependent type.

Lemma 3.5 (Substitution). *We have that*

(1) *If* $\Gamma, x : B, \Gamma'; \Delta \vdash t : T$*,* $\Gamma; \Delta \vdash b : B$*, and* $\vdash \Delta$*, then* $\Gamma, \Gamma'[x \mapsto b]; \Delta \vdash t[x \mapsto b] : T[x \mapsto b]$.
(2) *If* $\emptyset; \Delta \vdash \Gamma, x : B, \Gamma'$*,* $\Gamma; \Delta \vdash b : B$*, and* $\vdash \Delta$*, then* $\emptyset; \Delta \vdash \Gamma, \Gamma'[x \mapsto b]$.
(3) *If* $\Gamma, x : B, \Gamma'; \emptyset; \Delta \vdash T$*,* $\Gamma; \Delta \vdash b : B$*, and* $\vdash \Delta$*, then* $\Gamma, \Gamma'[x \mapsto b]; \emptyset; \Delta \vdash T[x \mapsto b]$.

The Substitution lemma relies on the types of well typed terms being well formed. The proof is straightforward since all typing derivations check for well formed environments, which must include well formed types.

$$
\begin{array}{rcl}
A, B, C & ::= & \mathbf{Nat} \mid \mathbf{Clos}(\overrightarrow{A} \to B) \mid \langle \overrightarrow{A} \rangle \mid \mathbf{Code}(\langle \overrightarrow{A} \rangle \mid B \to C) \\
e & ::= & \mathbb{N} \mid x \mid e_1\ \overrightarrow{e_2} \mid \langle \overrightarrow{e} \rangle \mid \pi_{\mathbb{N}}\ e \mid \langle\!\langle e_1, e_2 \rangle\!\rangle \mid \mathbf{cletrec}\ \overrightarrow{x = \langle\!\langle x_f, \langle \overrightarrow{x_v} \rangle \rangle\!\rangle}\ \mathbf{in}\ e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \\
f & ::= & \lambda x^{\mathsf{ve}} : A^{\mathsf{ve}}.\lambda \overrightarrow{x : B}.e \\
p & ::= & \mathbf{letrec}\ \overrightarrow{x = f}\ \mathbf{in}\ e
\end{array}
$$

Fig. 8. ACC Syntax

$$\boxed{\Delta \vdash f : \mathbf{Code}(\langle \overrightarrow{A} \rangle \mid \overrightarrow{B} \to C)}$$

$$
\textsc{a-Code}\ \frac{(\emptyset,\ x^{\mathsf{ve}} : \langle \overrightarrow{A} \rangle,\ \overrightarrow{x : B}); \Delta \vdash e : C}{\Delta \vdash \lambda x^{\mathsf{ve}} : \langle \overrightarrow{A} \rangle.\lambda \overrightarrow{x : B}.e : \mathbf{Code}(\langle \overrightarrow{A} \rangle \mid \overrightarrow{B} \to C)}
$$

Fig. 9. ACC Code Typing

Lemma 3.6. *If* $\Gamma; \Delta \vdash t : T$ *and* $\vdash \Delta$, *then* $\Gamma; \emptyset; \Delta \vdash T$.

4 Compiling to FCC

Minamide et al. [1996a] provide a type-preserving translation from a typed λ-caculus-like language into λ^{cl}, a typed IL with abstract closures, and a type-preserving translation from λ^{cl} into a typed IL where the closure abstraction is explicitly encoded using existential types.

In λ^{cl}, closures are created by packaging a code pointer with an environment. Closures only support one operation: being called as a package with arguments. This is the "abstract" part of "abstract closure conversion": the representation of closures is implicit, so we cannot rely on it for optimizations. The IL λ^{cl} does not support many of the optimizations we wish to support, such as the known procedure optimizations.

To show how FCC can be used as part of a type preserving compiler, we define a translation from a version of λ^{cl}, that we call the Abstract Closure Calculus (ACC), to FCC, and show that it preserves types. We choose to show the translation from ACC to FCC, as opposed to looking at a translation from a not-yet-closure-converted source language. This is because there is existing literature that targets ACC-style closures, and we wish to focus on the representation of closures for optimization in FCC, including the difference in representation between ACC and FCC.

ACC has the same closure abstraction as λ^{cl}, but assumes that procedures have already been hoisted to the top-level and that certain expressions are in a monadic form. Type-preserving hoisting and monadic form transformations are straightforward and have been implemented as part of, e.g., Morrisett et al. [1999]. In addition, ACC implements two forms of recursion. ACC has recursive procedure definitions via letrec, but also the recursive closure binding form, **cletrec**, which supports mutually recursive closures; this is necessary to generate mutually recursive closures from mutually recursive functions.

4.1 The ACC Language

As ACC is not novel, and many of the details are similar to FCC, we present a terse description of the language syntax and typing rules.

A program is a set of recursive procedure definitions $x = f$, followed by an expression e representing the body (see Figure 8). The typing rule for ACC programs is identical to the one for FCC programs and therefore omitted.

$$\boxed{\Gamma;\Delta \vdash e \,:\, A}$$

$$\frac{\Gamma;\Delta \vdash e_1 \,:\, \mathbf{Clos}(\vec{A} \to B) \qquad \overrightarrow{\Gamma;\Delta \vdash e_2 \,:\, A}}{\Gamma;\Delta \vdash e_1\ \vec{e_2} \,:\, B} \qquad \frac{\overrightarrow{\Gamma;\Delta \vdash e \,:\, A}}{\Gamma;\Delta \vdash \langle \vec{e} \rangle \,:\, \langle \vec{A} \rangle} \qquad \frac{\Gamma;\Delta \vdash e \,:\, \langle \vec{A} \rangle}{\Gamma;\Delta \vdash \pi_i\ e \,:\, A_i}$$

$$\frac{\overrightarrow{\Gamma, \overrightarrow{x \,:\, A};\Delta \vdash \langle\!\langle x_f, \langle \vec{x_v} \rangle \rangle\!\rangle \,:\, A} \qquad \Gamma, \overrightarrow{x \,:\, A};\Delta \vdash e \,:\, B}{\Gamma;\Delta \vdash \mathbf{cletrec}\ \overrightarrow{x = \langle\!\langle x_f, \langle \vec{x_v} \rangle \rangle\!\rangle}\ \mathbf{in}\ e \,:\, B} \qquad \frac{\begin{array}{c}\Gamma;\Delta \vdash e_1 \,:\, \mathbf{Code}(\langle \vec{A} \rangle \mid \vec{B} \to C) \\ \Gamma;\Delta \vdash e_2 \,:\, \langle \vec{A} \rangle\end{array}}{\Gamma;\Delta \vdash \langle\!\langle e_1, e_2 \rangle\!\rangle \,:\, \mathbf{Clos}(\vec{B} \to C)}$$

Fig. 10. ACC Expression Typing (Excerpts)

$$\boxed{\llbracket \Gamma;\Delta \vdash f \,:\, \mathbf{Code}(\langle \vec{A} \rangle \mid \vec{B} \to C) \rrbracket \triangleq f}$$

$$\begin{array}{ll} \llbracket \Gamma;\Delta \vdash \lambda x \,:\, \langle \vec{A} \rangle . \lambda \overrightarrow{y \,:\, B}.e & \triangleq \lambda \hat{x} \,:\, \mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \overrightarrow{\llbracket B \rrbracket} \to \llbracket C \rrbracket), \overrightarrow{\llbracket A \rrbracket}), \overrightarrow{y \,:\, \llbracket B \rrbracket}. \\ \quad :\, \mathbf{Code}(\langle \vec{A} \rangle \mid \vec{B} \to C) \rrbracket & \quad \mathsf{let}\ x = \mathsf{rec}(\pi_1\ \mathsf{unfold}(\hat{x}), ..., \pi_{n+1}\ \mathsf{unfold}(\hat{x})) \\ & \quad \mathsf{in}\ \llbracket \Gamma, x \,:\, \langle \vec{A} \rangle, \overrightarrow{y \,:\, B} \vdash e \,:\, C \rrbracket \end{array}$$

Fig. 11. Type-Directed Translation of ACC Procedures to FCC

An ACC procedure expects an environment argument x^{ve} with type $\langle \vec{A} \rangle$, and other arguments $\vec{x}$ with types $\vec{B}$, as shown in Figure 9. ACC environment arguments must be records, a property satisfied by the translation to ACC from Minamide et al. [1996a]. The type of procedures is represented by $\mathbf{Code}(\langle \vec{A} \rangle \mid \vec{B} \to C)$, where $\vec{B}$ are the input types, C is the output type, and $\langle \vec{A} \rangle$ is the environment argument type.

Environment arguments are represented by records $(\langle \vec{e} \rangle)$ and projecting from them $(\pi_i\ e)$. Records are given the type $\langle \vec{A} \rangle$, where A_i represents the type of the ith element of the record, as seen in Figure 10.

A closure, $\langle\!\langle e_1, e_2 \rangle\!\rangle$, in ACC is constructed by packaging a procedure label e_1 with an environment e_2, containing just the free variables of e_1. Closures have the closure type $\mathbf{Clos}(\vec{B} \to C)$, assuming the procedure has the type $\mathbf{Code}(\langle \vec{A} \rangle \mid \vec{B} \to C)$, and the environment then has the expected record type $\langle \vec{A} \rangle$, as seen in Figure 10. Closure application $e_1\ \vec{e_2}$ applies the closure e_1 as a package to the arguments $\vec{e_2}$

Like FCC, ACC has a recursive closure binding form, **cletrec**. However, whereas the FCC cletrec form requires a monadic form, only accepting closures in value form, the ACC **cletrec** has a stricter requirement where the closure must be explicitly constructed inline with only variable references. This restriction ensures that the translation produces valid FCC cletrec forms without additional monadic transformations. This is similar to restrictions on `letrec` in OCaml and standard ML[3]. In addition, the translation to ACC by Minamide et al. [1996a] already produces closures in this form.

4.2 From ACC to FCC

The translation from ACC to FCC fixes a representation for closures as flat closures. The main detail in the translation between ACC and FCC is the change in representation, which intuitively requires introducing some η-expansions to mediate between a flat closure record from a code pointer and an environment record. In practice, these η-expansions are easily optimized away or altogether avoided.

[3]The OCaml Reference: Chapter 12 https://v2.ocaml.org/manual/letrecvalues.html. [Accessed March 2023]

$$\boxed{[\![\Gamma;\Delta \vdash e : A]\!] \triangleq e}$$

$$\begin{aligned}
[\![\Gamma;\Delta \vdash e_1\ \overrightarrow{e_2} : B]\!] &\triangleq \mathsf{let}\ x = [\![\Gamma \vdash e_1 : \overrightarrow{A} \to B]\!] \\
&\quad\ \mathsf{in\ apply}\ (\pi_0\ x)\ \mathsf{is}(x)\ \overrightarrow{[\![\Gamma \vdash e_2 : A]\!]} \\
[\![\Gamma;\Delta \vdash \langle\!\langle e_1, e_2 \rangle\!\rangle : \mathbf{Clos}(\overrightarrow{B} \to C)]\!] &\triangleq \text{Given } \Gamma;\Delta \vdash e_2 : \langle \overrightarrow{A} \rangle \\
&\quad \mathsf{let}\ x = [\![\Gamma;\Delta \vdash e_2 : \langle \overrightarrow{A} \rangle]\!] \\
&\quad\ \mathsf{in\ let}\ x_0 = \pi_0\ x \\
&\quad\ \ \vdots \\
&\quad\ \mathsf{in\ let}\ x_n = \pi_n\ x \\
&\quad\quad \mathsf{in\ clos}(\mathsf{fold}(\mathsf{rec}([\![\Gamma;\Delta \vdash e_1 : \mathbf{Code}(\langle \overrightarrow{A} \rangle \mid \overrightarrow{B} \to C)]\!], \\
&\quad\quad\quad x_0, \dots, x_n))) \\
[\![\Gamma;\Delta \vdash \mathbf{cletrec}\ \overrightarrow{x = \langle\!\langle x_f, \langle \overrightarrow{x_v} \rangle \rangle\!\rangle}\ \mathbf{in}\ e : B]\!] &\triangleq \mathsf{cletrec}\ \overrightarrow{x = \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(x_f, \overrightarrow{x_v})))}\ \mathsf{in}\ [\![\Gamma, \overrightarrow{x : A};\Delta \vdash e : B]\!]
\end{aligned}$$

Fig. 12. Type-Directed Translation of ACC Expressions to FCC (Excerpts)

Type signatures are modified to the FCC closure representation type, which means that ACC environments must be reconstructed. The body of every ACC procedure uses an environment without the code pointer, which causes each projection out of the environment to be off-by-1 when translating the body from ACC to FCC. As we do not know everywhere the environment is accessed, the simplest way to reconstruct the environment is with the expected interface: a record only containing the values of free variables. This reconstruction, seen in Figure 11, is a type-directed translation so that the ACC environment can simply be reconstructed based on its type. We take the first to $n+1$ projections of the FCC environment $\hat{x}$ (a fresh parameter name) to reconstruct the environment representation as a record containing the values of free variables. We then let-bind the record to the previous environment parameter name x. Inside the let-binding, *i.e.*, the body of the procedure, x will have the expected representation for an ACC record.

Whereas in a procedure body we reconstruct the environment *without* the code pointer, at environment creation, we do the opposite: reconstructing the environment *with* the code pointer to fit the closure representation of FCC, as seen in Figure 12. Once the ACC environment e_2 is translated to an FCC record, we can create a closure in FCC. We construct the closure representation $\mathsf{rec}(x_{code}, x_0, \dots x_n)$ with the code pointer and values of free variables projected from the translation of the record e_2, and add the appropriate type casts fold and clos to produce an FCC closure abstraction.

Translating a closure application requires following the FCC closure calling convention. The closure is bound to a fresh variable x, to ensure monadic form, and then the code pointer is projected out to be applied, with the environment passed as the singleton of the closure represented by x. The rest of the arguments are translated and passed as usual.

While this translation appears to introduce substantial overhead, in practice, both of these steps that reconstruct the different representations of the environment are easy to eliminate by inlining. Closure conversion produces a procedure that uses the environment in a particular way: on entry, each free variable is immediately projected out of the environment and bound. Each procedure has essentially the following form $(env.\mathbf{let}\ x_0 = \pi_0\ env \dots x_n = \pi_n\ env\ \mathbf{in}\ \dots)$, where $x_0 \dots x_n$ are the free variables. Constant propagation and inlining then produces the correct projections of the new representation, where the indices are shifted by one, and produces a procedure of the form $(env.\mathbf{let}\ x_0 = \pi_1\ env \dots x_n = \pi_{n+1}\ env\ \mathbf{in}\ \dots)$. Closure conversion also produces environments that are a record of only references to free variables, for example $env = \langle x_0, \dots, x_n \rangle$. We could formalize this requirement in ACC and FCC, but it only adds unnecessary clutter to the definitions.

$$\begin{array}{rcl} A, B, C & ::= & \ldots \mid \mathsf{Code}^{wk}(\overrightarrow{A} \rightarrow B) \mid \mathsf{Code}(\overrightarrow{A} \rightarrow B) \mid \mathsf{Clos}^{\Omega}(\overrightarrow{A} \rightarrow B) \mid \mathsf{Dead} \\ v & ::= & \ldots \mid \mathsf{clos}^{\Omega}(v) \mid \mathsf{dead} \mid \mathsf{unknown}(v) \\ \Omega & ::= & \bot \mid \overrightarrow{l} \mid \top \end{array}$$

Fig. 13. FCC Extended Syntax for Optimizations

Alternatively, a direct translation from a source language to FCC would not have this mismatch, avoiding the issue altogether.

Theorem 4.1. *Type preservation*
If $\vdash$ ***letrec*** $\overrightarrow{x = f}$ ***in*** $e : B$, *then* $\vdash$ *letrec* $\overrightarrow{x = [\![\Delta \vdash f : A]\!]}$ *in* $[\![\emptyset; \Delta \vdash e : B]\!] : [\![B]\!]$

Proof. Follows by straightforward induction over the source typing derivation. □

5 Closure Optimizations

We implement the optimizations from Section 2 in FCC, demonstrating that FCC is expressive enough to implement and optimize flat closures, and validate the correctness of optimization. Type checking in FCC after optimization can be used to validate the optimizations preserve type and memory safety. We also detail several optimizations that FCC does not yet support. Following Keep et al. [2012], we implement the optimizations in an order that helps optimizations cascade. An evaluation of these optimizations within FCC is left as future work (see Section 6).

Many optimizations rely on removing indirection when a procedure is known to be called at the call sites of a closure. As is standard, we formalize several propositions about procedures and closures. A closure is *known* if we know the unique procedure reached by calling the closure, which allows us to eschew the closure abstraction and call the procedure directly. A procedure is *known at a call site* if that procedure must be the destination of the call. A procedure is *well known* if it is known at every possible call site; a well known procedure need never use the closure abstraction.

To implement these optimizations, we decorate FCC closure and code types with information representing *known* and *well known* information. This allows the optimizations to be defined as type-directed intra-language rewrites, although a compiler may choose to target FCC in any other way and still benefit from its type safety.

The decorated syntax for FCC is shown in Figure 13. The FCC closure type $\mathsf{Clos}(\overrightarrow{A} \rightarrow B)$ is annotated with a label set Ω, representing the set of possible procedure values for the closure. Ω may also be $\bot$, to represent a closure that will never be called, or $\top$ to represent any procedure of the right type. We also decorate the code type $\mathsf{Code}(\overrightarrow{A} \rightarrow B)$ with an optional flag *wk*, which indicates that the procedure is well known. We use the flag ? to indicate a procedure that may or may not be well known. We add a "dead" value and type, representing dead code introduced by optimization, to be cleaned by another pass. Finally, we add a type-cast unknown to use a known closure in a more general context. A known closure may become unknown when it escapes, such as when returned from a procedure. Decorated terms trivially erase to FCC, by removing the annotations and turning the dead value into an empty record. This decorated syntax, each of the translations, an erasure function, and several examples (including Figure 14) are all implemented in the Redex model of FCC in the anonymous supplementary materials.

This information can be generated by a standard analysis. For example, Serrano [1995] provide an approximation algorithm that decides whether functions are known and well known.

It does not matter whether the annotations are correct; we can rely on the FCC type checker to validate the output of the optimizations. Incorrect annotations will cause the optimization to produce ill-typed FCC programs, but this is easily detected by type checking after optimization. We argue that if the annotations are correct, then the implementations will always produce well typed FCC programs.

5.1 Known and Well Known Optimizations

Using the decorated terms, we formalize the known proposition as follows. A closure of type $\mathsf{Clos}^{l}(\vec{A}{\to}B)$, whose label set is a single element, is *known*. At any call site to a closure of such a type, the procedure l is known at that call site. For example, if the closure x has the type $\mathsf{Clos}^{l_0}(\mathsf{Nat}{\to}\mathsf{Nat})$ at a call site $\mathsf{apply}\ (\pi_c\ x)\ (\mathsf{is}(x))\ 0$, then l_0 is known at this call site.

The first optimization removes closure call indirection for procedures known at a call site, and eliminates closure allocation for well known procedures. This optimization is formalized as the metafunction $[\![\_]\!]^{k}$ over typing derivations, where the superscript k the (well) known procedure optimization. We walk through the interesting cases of the definitions; all other cases are recursive on the subderivations.

When a procedure is known at a call site, we call the procedure directly, eliminating an unnecessary memory access to project the code pointer out of the closure. This is defined in the below rule, where a call site $\mathsf{apply}\ (\pi_c\ v)\ (\mathsf{is}(v)), \vec{e}$ invoking a known closure $\Delta;\Gamma \vdash v : \mathsf{Clos}^{l}(\vec{A}{\to}B)$ is optimized to invoke the procedure directly through its code pointer l and passes the underlying closure representation $[\![\Delta;\Gamma \vdash v : \mathsf{Clos}^{l}(\vec{A}{\to}B)]\!]^{\mathrm{k}}$ instead of following the closure calling convention.

$$\begin{aligned}[\![\Delta;\Gamma \vdash \mathsf{apply}\ (\pi_c\ v)\ (\mathsf{is}(v)), \vec{e} : B]\!]^{\mathrm{k}} \quad &\triangleq \quad \mathsf{apply}\ l\ ([\![\Delta;\Gamma \vdash v : \mathsf{Clos}^{l}(\vec{A}{\to}B)]\!]^{\mathrm{k}}), \overrightarrow{[\![\Delta;\Gamma \vdash e : A]\!]^{\mathrm{k}}} \\ &\quad\quad \text{where } \Delta;\Gamma \vdash v : \mathsf{Clos}^{l}(\vec{A}{\to}B)\end{aligned}$$

Since the procedure expects the closure representation, rather than the closure abstraction, we must remove the closure tag from known closures. We translate a known closure type to its representation, a recursive record, as show in the below rules. The metafunction $[\![\_]\!]^{k}_{\_}$ defines the type translation for the optimization. The type of a known closure ($\mathsf{Clos}^{l}(\vec{A}{\to}B)$) is translated to the the type of the underlying closure representation ($\mu\alpha.\mathsf{Rec}(\mathsf{Code}^{?}(\alpha, \vec{A}{\to}B), \vec{C})$) gleaned from the type of the procedure represented by l. When constructing a known closure, we remove the closure tag.

$$\begin{aligned}[\![\mathsf{Clos}^{l}(\vec{A}{\to}B)]\!]^{\mathrm{k}}_{\Delta} &\triangleq [\![\mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \vec{A}{\to}B), \vec{C})]\!]^{\mathrm{k}}_{\Delta} \\ &\quad \text{where } l : \mathsf{Code}(\mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \vec{A}{\to}B), \vec{C}), \vec{A}{\to}B) \in \Delta \\ [\![\Delta;\Gamma \vdash \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(l, \vec{v}))) : \mathsf{Clos}^{l}(\vec{A}{\to}B)]\!]^{\mathrm{k}} &\triangleq [\![\Delta;\Gamma \vdash \mathsf{fold}(\mathsf{rec}(l, \vec{v})) : \mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \vec{A}{\to}B), \vec{C})]\!]^{\mathrm{k}} \\ &\quad \text{where } [\![\mathsf{Clos}^{l}(\vec{A}{\to}B)]\!]^{\mathrm{k}}_{\Delta} = \mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha, \vec{A}{\to}B), \vec{C})\end{aligned}$$

When a known closure escapes, we must reintroduce the closure tag, as seen in the below rule. This should never happen to a closure for a well known procedure.

$$\begin{aligned}[\![\Delta;\Gamma \vdash \mathsf{unknown}(v) : \mathsf{Clos}^{\Omega}(\vec{A}{\to}B)]\!]^{\mathrm{k}} \quad &\triangleq \quad \mathsf{clos}^{\Omega}(v) \\ &\quad\quad \text{where } \Delta;\Gamma \vdash v : \mathsf{Clos}^{l}(\vec{A}{\to}B) \text{ and } \Omega \supsetneq l\end{aligned}$$

Now, we formalize the well known closure proposition and define related optimizations. A well known procedure has type $\mathsf{Code}^{wk}(\vec{A}{\to}B)$, where the wk flag says that it can never be placed in an unknown closure. The code pointer of a closure for a well known procedure is unnecessary, since every call site will invoke it directly through its code pointer, so we remove it, leaving only the free variables for the procedure in the closure—the closure becomes only an environment. Then, we can optimize the representation based on the number of free variables. To optimize this representation, we change the type of the environment, the construction, and projections from it. We define three cases in our implenentation.

First, if a well known procedure has no free variables, there is no need to allocate or pass the environment, and it becomes dead code.

$$
\begin{array}{rcl}
[\![\mu\alpha.\mathsf{Rec}(\mathsf{Code}^{wk}(\alpha, \overrightarrow{A}{\rightarrow}B))]\!]^{\mathsf{k}}_{\Delta} & \triangleq & \mathsf{Dead} \\
[\![\Delta;\Gamma \vdash \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(l))) : \mathsf{Clos}^{l}(\overrightarrow{A}{\rightarrow}B)]\!]^{\mathsf{k}} & \triangleq & \mathsf{dead} \\
 & & \text{where } [\![\mathsf{Clos}^{l}(\overrightarrow{A}{\rightarrow}B)]\!]^{\mathsf{k}}_{\Delta} = \mathsf{Dead} \\
[\![\Delta;\Gamma \vdash \pi_n\ \mathsf{unfold}(e) : A]\!]^{\mathsf{k}} & \triangleq & \mathsf{dead} \\
 & & \text{where } \Delta;\Gamma \vdash e : \mu\alpha.\mathsf{Rec}(\mathsf{Code}^{wk}(\alpha, \overrightarrow{A}{\rightarrow}B), \overrightarrow{C}) \\
 & & \text{and } [\![\mu\alpha.\mathsf{Rec}(\mathsf{Code}^{wk}(\alpha, \overrightarrow{A}{\rightarrow}B))]\!]^{\mathsf{k}}_{\Delta} = \mathsf{Dead}
\end{array}
$$

Second, if a well known procedure has just one free variable (with type C), then that procedure is lambda lifted: the free variable is passed instead of the environment, and no allocation is necessary. We follow Keep et al. [2012] doing this only for the singleton environment, since lambda lifting larger environments can increase register pressure.

$$
\begin{array}{rcl}
[\![\mu\alpha.\mathsf{Rec}(\mathsf{Code}^{wk}(\alpha, \overrightarrow{A}{\rightarrow}B), C)]\!]^{\mathsf{k}}_{\Delta} & \triangleq & [\![C]\!]^{\mathsf{k}}_{\Delta} \\
[\![\Delta;\Gamma \vdash \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(l, v))) : \mathsf{Clos}^{l}(\overrightarrow{A}{\rightarrow}B)]\!]^{\mathsf{k}} & \triangleq & [\![\Delta;\Gamma \vdash v : C]\!]^{\mathsf{k}} \\
 & & \text{where } [\![\mathsf{Clos}^{l}(\overrightarrow{A}{\rightarrow}B)]\!]^{\mathsf{k}}_{\Delta} = C \\
[\![\Delta;\Gamma \vdash \pi_n\ \mathsf{unfold}(e) : A]\!]^{\mathsf{k}} & \triangleq & [\![\Delta;\Gamma \vdash e : \mu\alpha.\mathsf{Rec}(\mathsf{Code}^{wk}(\alpha, \overrightarrow{A}{\rightarrow}B), C)]\!]^{\mathsf{k}} \\
 & & \text{where } \Delta;\Gamma \vdash e : \mu\alpha.\mathsf{Rec}(\mathsf{Code}^{wk}(\alpha, \overrightarrow{A}{\rightarrow}B), C) \\
 & & \text{and } [\![\mu\alpha.\mathsf{Rec}(\mathsf{Code}^{wk}(\alpha, \overrightarrow{A}{\rightarrow}B), C)]\!]^{\mathsf{k}}_{\Delta} = C
\end{array}
$$

Finally, if a closure has two or more arguments (with types $\overrightarrow{C}$), we represent the closure with a record, but remove the code pointer. Note that the check $length(\overrightarrow{C}) > 1$ distinguishes the case where there is just one free variable that happens to have a record type from a closure represented with a record.

$$
\begin{array}{rcl}
[\![\mu\alpha.\mathsf{Rec}(\mathsf{Code}^{wk}(\alpha, \overrightarrow{A}{\rightarrow}B), \overrightarrow{C})]\!]^{\mathsf{k}}_{\Delta} & \triangleq & \mathsf{Rec}(\mathsf{Dead}, \overrightarrow{[\![C]\!]^{\mathsf{k}}_{\Delta}}) \\
[\![\Delta;\Gamma \vdash \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(l, \overrightarrow{v}))) : \mathsf{Clos}^{l}(\overrightarrow{A}{\rightarrow}B)]\!]^{\mathsf{k}} & \triangleq & \mathsf{rec}(\overrightarrow{[\![\Delta;\Gamma \vdash v : C]\!]^{\mathsf{k}}}) \\
 & & \text{where } [\![\mathsf{Clos}^{l}(\overrightarrow{A}{\rightarrow}B)]\!]^{\mathsf{k}}_{\Delta} = \mathsf{Rec}(\overrightarrow{C}) \\
[\![\Delta;\Gamma \vdash \pi_n\ \mathsf{unfold}(e) : A]\!]^{\mathsf{k}} & \triangleq & \pi_{n-1}\ [\![\Delta;\Gamma \vdash e : \mu\alpha.\mathsf{Rec}(\mathsf{Code}^{wk}(\alpha, \overrightarrow{A}{\rightarrow}B), \overrightarrow{C})]\!]^{\mathsf{k}} \\
 & & \text{where } \Delta;\Gamma \vdash e : \mu\alpha.\mathsf{Rec}(\mathsf{Code}^{wk}(\alpha, \overrightarrow{A}{\rightarrow}B), \overrightarrow{C}) \\
 & & \text{and } [\![\mu\alpha.\mathsf{Rec}(\mathsf{Code}^{wk}(\alpha, \overrightarrow{A}{\rightarrow}B), \overrightarrow{C})]\!]^{\mathsf{k}}_{\Delta} = \mathsf{Rec}(\overrightarrow{C}) \\
 & & \text{and } length(\overrightarrow{C}) > 1
\end{array}
$$

The translation presented here makes some assumptions about the syntax of how closure abstractions are constructed, and how closure environments are accessed, to simplify implementation. The translation to FCC presented in Section 4 meets both of these assumptions. First, closures abstractions are assumed to be constructed with the syntax ($\mathsf{clos}(\mathsf{fold}(\mathsf{rec}(l, \overrightarrow{v})))$), to simplify modifying the underlying representation for closures of well known procedures. Similarly, the translation assumes that closure environments are accessed with the syntax ($\pi_n\ \mathsf{unfold}(env)$) inside procedure bodies.

We conjecture that, if the annotations on closures are correct, then applying the known and well known procedure optimizations to a well typed FCC program should result in a well typed FCC program. Essentially, the known procedure optimization statically removes the clos, accept-clos, and is type casts, and statically evaluates the clos-proj along paths where the underlying closure representation is known. This statically transforms what would evaluate by $\beta{-}clos$ to something that evaluates by β. By subject reduction, we know the term at the call site after reduction is well typed, and since the procedure is known at the call site, it's easy to rebuild the typing derivation using Rule Apply without Rule Accept-Clos, Rule Clos-Proj, or Rule Decl-Clos. The operator is changed to the procedure directly, and the first argument is changed to be the concrete representation of the closure rather than a singleton closure. For a well known procedure, since we know all call sites for the closure, we rewrite the entire typing derivation as described above, resulting in the closure becoming dead code. We also rewrite the derivation at the procedure definition, changing the type of the environment as described in the implementation. Since we change both the type and the access to the environment, its easy to check the modified procedure definition is still well typed.

5.2 Eliminating Unnecessary Free Variables

Dead Code Elimination. Known procedure optimizations may remove the need for a closure, creating a dead free variable that can be removed from other closures. Dead code elimination is a standard optimization, so we do not discuss the implementation here.

Self Recursive Closures. A recursive procedure need not have its own closure as a free variable, as its closure is the environment argument. Instead of including a self reference in its environment, we can replace the self reference inside the procedure body with the environment. We can detect this case by looking for known closures of l, values with the type $\mathsf{Clos}^{l}(\vec{A}\to B)$, in the closure environment for the procedure l. Since these closures are not necessary, we mark them as dead code, as seen in the second rule below. Similarly, when constructing a closure, if we find that any of the environment values are known closures of the procedure we are constructing the closure for (*i.e.,* have the type $\mathsf{Clos}^{l}(\vec{A}\to B)$), then we mark it as dead.

The translation $[\![A]\!]^{\mathrm{sr}}(l)$ performs the self recursive closure elimination optimization over types. The argument l is the label of the procedure whose type is being translated. In the first translation rule, we inspect each parameter type of a procedure, and in the second rule, mark as dead any parameter whose type indicates that it represents a known closure of l.

$$\begin{aligned} [\![\mathsf{Code}^{?}(\mu\alpha.\mathsf{Rec}(\mathsf{Code}^{?}(\alpha,\vec{A}\to B),\vec{C}),\vec{A}\to B)]\!]^{\mathrm{sr}}(l) &\triangleq \mathsf{Code}^{?}(\mu\alpha.\mathsf{Rec}(\mathsf{Code}^{?}(\alpha,\vec{A}\to B),\overrightarrow{[\![C]\!]^{\mathrm{sr}}(l)}),\vec{A}\to B)\\ [\![\mathsf{Clos}^{l}(\vec{A}\to B)]\!]^{\mathrm{sr}}(l) &\triangleq \mathsf{Dead}\end{aligned}$$

The translation $[\![\Delta;\Gamma\vdash e\,:\,A]\!]^{\mathrm{sr}}(l_1,l_2,x)$ performs the self recursive closure elimination optimization over expression type derivations. The translation on expression type derivations takes 3 arguments: l_1, the known procedure of a closure; l_2, the current procedure; and x, the parameter name of the procedure's enviroment. In the first rule, the translation is applied recursively to each of the subterms of the closure, with l_1 being set to the known label of the closure being constructed. In the second rule, any value representing a known closure over l_1 represents a self recursive closure and gets marked as dead code.

$$\begin{aligned} [\![\Delta;\Gamma\vdash \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(l_1,\vec{v})))\,:\,\mathsf{Clos}^{l_1}(\vec{A}\to B)]\!]^{\mathrm{sr}}(\_,l_2,x) &\triangleq \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(l_1,\overrightarrow{[\![\Delta;\Gamma\vdash v\,:\,C]\!]^{\mathrm{sr}}(l_1,l_2,x)})))\\ [\![\Delta;\Gamma\vdash v\,:\,\mathsf{Clos}^{l_1}(\vec{A}\to B)]\!]^{\mathrm{sr}}(l_1,l_2,x) &\triangleq \mathsf{dead}\end{aligned}$$

In the body of a procedure l_2, if we ever project a known closure for l_2 (*i.e.,* a closure with the type $\mathsf{Clos}^{l_2}(\vec{A}\to B)$) out of the environment x, we can instead construct the same closure abstraction by closing over the environment x, eliminating a memory access, as shown in the below rule.

$$[\![\Delta;\Gamma\vdash \pi_n\ \mathsf{unfold}(x)\,:\,\mathsf{Clos}^{l_2}(\vec{A}\to B)]\!]^{\mathrm{sr}}(\_,l_2,x) \triangleq \mathsf{clos}(x)$$

As with the translation for the known and well known optimizations, all the remaining cases simply apply the translation recursively to the sub-derivations and rebuild the original term.

We conjecture this is type preserving as well. A self recursive procedure has its own closure in its closure environment. Since the known closure has the same type as itself as the environment parameter (up to folding the recursive type and the closure tag), we can substitute all projections of the closure from the environment with the environment parameter itself, appropriately tagged. Then, any field of the procedure's environment parameter whose type indicates that it is a known closure of the recursive procedure will never be projected out, so it becomes dead code.

Mutually Recursive Closures. Similar to the self recursive closure elimination optimization, if a set of mutually recursive procedures only have each others' closures as free variables, then the closures are not necessary. However, if even one of the procedures has another free variable that it does use, then all the closures are necessary: the first closure to contain the free variable, and the rest of the closures to contain the first closure with its useful information.

The translation $[\![\Delta;\Gamma \vdash e : A]\!]^{\mathrm{mr}}(\vec{l})$ performs the mutually recursive closure elimination optimization over expression type derivations. It expects a set of labels $\vec{l}$, instead of a single label, and any known closure for any of the procedures in the set of labels can be optimized to a closure over just the label (and dead code).

If these closures over just the labels could be allocated statically, then they would not add any allocation or record initialization overhead, while eliminating the reads on every recursive call. However, this is not currently expressible in FCC, as discussed further in Subsection 5.4.

$$\begin{aligned}
[\![\mathsf{Code}^?(\mu\alpha.\mathsf{Rec}(\mathsf{Code}^?(\alpha, \vec{A}\to B), \vec{C}), \vec{A}\to B)]\!]^{\mathrm{mr}}(\vec{l}) &\triangleq \mathsf{Code}^?(\mu\alpha.\mathsf{Rec}(\mathsf{Code}^?(\alpha, \vec{A}\to B), \overrightarrow{[\![C]\!]^{\mathrm{mr}}(\vec{l})}), \vec{A}\to B) \\
[\![\mathsf{Clos}^{l}(\vec{A}\to B)]\!]^{\mathrm{mr}}(\vec{l}) &\triangleq \mathsf{Dead} \qquad\qquad \text{where } l \in \vec{l} \\
[\![\Delta;\Gamma \vdash \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(l_2, \vec{v}))) : \mathsf{Clos}^{l_2}(\vec{A}\to B)]\!]^{\mathrm{mr}}(\vec{l}) &\triangleq \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(l_2, \overrightarrow{[\![\Delta;\Gamma \vdash v : C]\!]^{\mathrm{mr}}(\vec{l})}))) \\
&\ \text{where } l_2 \in \vec{l} \\
[\![\Delta;\Gamma \vdash v : \mathsf{Clos}^{l_2}(\vec{A}\to B)]\!]^{\mathrm{mr}}(\vec{l}) &\triangleq \mathsf{dead} \qquad\qquad \text{where } l_2 \in \vec{l} \\
[\![\Delta;\Gamma \vdash \pi_n\, e : \mathsf{Clos}^{l_2}(\vec{A}\to B)]\!]^{\mathrm{mr}}(\vec{l}) &\triangleq \mathsf{clos}(\mathsf{fold}(\mathsf{rec}(l_2, \overrightarrow{\mathsf{dead}}))) \\
&\ \text{where } l_2 \in \vec{l} \text{ and } \mathrm{length}(\overrightarrow{\mathsf{dead}}) = \mathrm{length}(\vec{A})
\end{aligned}$$

Like the other optimizations, we conjecture that this is type preserving. The argument is similar to that of self recursive closures: we can replace projecting a known closure from the environment with a freshly constructed one of the same type. Then, each of the recursive closures in the environment is unused, so we mark them as dead code. Since we only change the types of unused values, the transformation is type preserving.

5.3 Sharing Closures

In some cases, a group of closures can be optimized to a single closure to reduce allocations, further discussed by Keep et al. [2012]. The group of closures must share the same free variable values and require at most one code pointer. A group of closures requiring at most one code pointer can be found *after* the well known and known optimizations are applied, *e.g.,* at most one of the closures is for a procedure that is not well known. We do not formalize this optimization in FCC as it requires complex graph-based reasoning outside the scope of FCC; however, we show FCC's closure abstraction and representation can type check such an optimization through a concrete example.

We show an example program after the well known and known optimizations have been applied on the left side of Figure 14. The program has three procedures p_1, p_2, and p_3, and three closures f_1, f_2, f_3. Suppose that the procedures p_1 and p_2 are well known, but p_3 is not (p_3 is in fact well known in this example, but it is simplified for clarity). The program cannot call the procedure p_3 directly and must go through the closure f_3. The known closures f_1 and f_2, and f_3 all have the same enviroment. We can optimize this to instead allocate a single closure *temp*, shown on the right side of Figure 14. Because the group of closures only require one code pointer for the procedure p_3, the closure can be shared for calling procedures p_1 and p_2. Since p_1 and p_2 are well known, we can modify their signatures to accept the same closure representation as the procedure p_3.

Before

```
letrec
$p_1$ : $\mathsf{Code}^{wk}(\mathsf{Rec}(\mathsf{Nat}, \mathsf{Nat}) \to \mathsf{Nat})$
    = $\lambda env.\ \pi_0\ env + \pi_1\ env$
$p_2$ : $\mathsf{Code}^{wk}(\mathsf{Rec}(\mathsf{Nat}, \mathsf{Nat}) \to \mathsf{Nat})$
    = $\lambda env.\ \pi_0\ env * \pi_1\ env$
$p_3$ : $\mathsf{Code}(\mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha \to \mathsf{Nat}), \mathsf{Nat}, \mathsf{Nat}) \to \mathsf{Nat})$
    = $\lambda env.\ (\pi_1\ \mathsf{unfold}(env))\ /\ (\pi_2\ \mathsf{unfold}(env))$
in

let $f_1$ = $\mathsf{rec}(v_1, v_2)$ in
let $f_2$ = $\mathsf{rec}(v_1, v_2)$ in
let $f_3$ = $\mathsf{clos}^{\top}(\mathsf{fold}(\mathsf{rec}(p_3, v_1, v_2)))$ in
  $(\mathsf{apply}\ p_1\ f_1) + (\mathsf{apply}\ p_2\ f_2)$
   $+ (\mathsf{apply}\ (\pi_c\ f_3)\ \mathsf{is}(f_3))$
```

After

```
letrec
$p_1$ : $\mathsf{Code}^{wk}(\mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha \to \mathsf{Nat}), \mathsf{Nat}, \mathsf{Nat}) \to \mathsf{Nat})$
    = $\lambda env.\ (\pi_1\ \mathsf{unfold}(env)) + (\pi_2\ \mathsf{unfold}(env))$
$p_2$ : $\mathsf{Code}^{wk}(\mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha \to \mathsf{Nat}), \mathsf{Nat}, \mathsf{Nat}) \to \mathsf{Nat})$
    = $\lambda env.\ (\pi_1\ \mathsf{unfold}(env)) * (\pi_2\ \mathsf{unfold}(env))$
$p_3$ : $\mathsf{Code}(\mu\alpha.\mathsf{Rec}(\mathsf{Code}(\alpha \to \mathsf{Nat}), \mathsf{Nat}, \mathsf{Nat}) \to \mathsf{Nat})$
    = $\lambda env.\ (\pi_1\ \mathsf{unfold}(env))\ /\ (\pi_2\ \mathsf{unfold}(env))$
in
+ let $temp$ = $\mathsf{rec}(p_3, v_1, v_2)$ in
let $f_1$ = $\mathsf{fold}(temp)$ in
let $f_2$ = $\mathsf{fold}(temp)$ in
let $f_3$ = $\mathsf{clos}^{\top}(\mathsf{fold}(temp))$ in
  $(\mathsf{apply}\ p_1\ f_1) + (\mathsf{apply}\ p_2\ f_2)$
   $+ (\mathsf{apply}\ (\pi_c\ f_3)\ \mathsf{is}(f_3))$
```

Fig. 14. An example of sharing closures through aliasing

5.4 Optimizations Not Yet Supported in FCC

There are some optimizations from Keep et al. [2012] that FCC cannot yet express, but we conjecture that all of the optimizations are possible with an extension allowing the (mutual) global static allocation of data along with procedures. Currently, FCC only allows closed procedures in the global scope to simplify the mutual induction between well typed expressions, well formed types, and well formed environments. Extending FCC's global scope to include other values requires proving mutual induction between expressions and types is still well defined, which is not immediately clear because of the dependent type used for our closure abstraction. Mutually inductive dependent structures, such as mutually inductive indexed type families, are well understood and may provide a starting point for resolving this detail.

With such an extension, dynamic allocation overhead of closures can be reduced when the environment only refers to global scope values. Global values do not need to be included in closures because procedures can refer to these values directly. A closure c can be promoted to a global scope when it only refers to other globally scoped values. Once a closure c is a global value, any references to c in other closures are no longer necessary and can be removed, causing cascading optimizations that possibly promote more closures.

In Section 5.2, we saw that a group of mutually recursive closures with no free variables other than the other closures in the group can be optimized into closures containing just the label of the target procedure. We could further optimize this by statically allocating each of the closures, instead of re-allocating a closure for each mutually recursive call. However, static allocation in this form is not currently supported by FCC.

Figure 15 shows an example from Keep et al. [2012], with the left side showing two mutually recursive procedures *evenp* and *oddp*, with mutually recursive closures *evenc* and *oddc* defined using cletrec (note that the known procedure optimization is not performed in this example for clarity). We can optimize the mutually recursive procedures and closures together to a version of FCC with static global allocation of closures mutually defined with procedures. The result of this potential optimization is shown on the right of Figure 15. The closures *evenc* and *oddc* may still be necessary, but are now statically allocated ahead of time. Since the closures *evenc* and *oddc* do not need to hold references to each other after static allocation, we can optimize the allocation of the closures to only include each respective code pointer. The procedures *evenp* and *oddp* can then be

Before

```
letrec
  evenp, oddp : Code((μα.Rec(Code(α, Nat→Nat),
                              Clos(Nat→Nat)),
                     Nat)→Nat)
  evenp = λenv, x.
–     let g = π₁ env in
      (x = 0) or (apply π_c g is(g) (– x 1))
  oddp = λenv, x.
–     let g = π₁ env in
      (not(apply π_c g is(g), x))
in
–   cletrec
    evenc : Clos^evenp(Nat→Nat)
–         = clos^evenp(fold(rec(evenp, oddc)))
    oddc : Clos^oddp(Nat→Nat)
–         = clos^oddp(fold(rec(oddp, evenc)))
    in apply π_c evenc (is(evenc), 10)
```

After (not currently expressible in FCC)

```
letrec
  evenp, oddp : Code((μα.Rec(Code(α, Nat→Nat),
                              Dead),
                     Nat)→Nat)
  evenp = λenv, x.

      (x = 0) or (apply π_c oddc is(oddc) (– x 1))
  oddp = λenv, x.

      (not(apply π_c evenc (is(evenc), x)))


    evenc : Clos^evenp(Nat→Nat)
+         = clos^evenp(fold(rec(evenp)))
    oddc : Clos^oddp(Nat→Nat)
+         = clos^oddp(fold(rec(oddp)))
    in apply π_c evenc (is(evenc), 10)
```

Fig. 15. (Conjecture) Mutually Recursive Procedures After a Static Allocation Optimization

optimized to call each other through the statically allocated closures, rather than from the dynamic closure environment argument.

6 Related and Future Work

Abstract Closure Optimizations in TIL. The TIL project used a typed intermediate language to compile ML, including abstract closure conversion in their typed closure IL [Morrisett 1995; Tarditi 1996; Tarditi et al. 1996]. While TIL performed well against the SML/NJ compiler at the time, it lacks some of the optimizations we discuss. TIL lambda-lifts well known functions, only closure converting escaping functions. It uses a pair representation of a closure, not the flat closure representation of FCC. We infer that TIL does not support the known function optimizations we discuss, and requires reallocating escaping recursive closures. TIL is only described at a high level, but is based on the formalism of Minamide et al. [1996a], which does not support these optimizations. TIL does model global variables, and avoids including them in closures.

Polymorphism and Closure Conversion. In the typed closure conversion work of Minamide et al. [1996a] and Morrisett et al. [1999], the goal was to support closure conversion from System F, which requires supporting polymorphism in the source language. Both projects use the existential type abstraction and representation described in Section 2, but they differ in their interpretation of polymorphism and how type variables interact with closures.

Minamide et al. [1996a] use a type passing interpretation of polymorphism, which requires a complex typing of closures to account for capturing free type variables, which also appear in the type of the procedure. Minamide et al. [1996a] use separate type and value environments as part of closures. In this approach, types may need to be projected out of the type environment, but the type environment is hidden behind the existential type, which means it cannot be projected from. Their solution is to use a translucent function type, a kind of singleton type restricted to parameter position on functions, allowing values from the type environment to be used in the type. As their approach uses a singleton type and FCC has a singleton type, we conjecture that FCC can be extended to handle type-passing polymorphism and this style of closure conversion.

By contrast, Morrisett et al. [1999] use a type erasure strategy, which simplifies the typing of procedures and closures, but may restrict the way type variables are used at run time. Typically, type erasure prevents an intensional interpretation of polymorphism, which precludes a number of optimizations, such as those discussed by Leroy [1992]. On the other hand, Crary et al. [2002] present a framework supporting intensional polymorphism with type erasure. A typed erasure interpretation does not affect closure conversion at all, so FCC could be extended with standard polymorphic type constructs for this style of closure conversion, and would not require additional complexity for encoding the type environment.

Explicit Allocation. The next pass in a type-preserving compiler is allocation, following the System F to TAL compiler [Morrisett et al. 1999]. In this compiler pass, records are explicitly allocated in memory and then initialized, as follows.

$$\llbracket \langle e_1, e_2 \ldots \rangle : \mathrm{Rec}(A_1, A_2, \ldots) \rrbracket \overset{\mathrm{def}}{=} \begin{array}{l} \textbf{let } \mathbf{y} = \textbf{malloc } \llbracket A_1 \rrbracket, \llbracket A_2 \rrbracket, \ldots \textbf{ in} \\ \textbf{let } \mathbf{y_1} = \mathbf{y}[\mathbf{1}] \leftarrow \llbracket e_1 \rrbracket \textbf{ in} \\ \ldots \\ \textbf{let } \mathbf{y_n} = \mathbf{y_{n-1}}[\mathbf{n}] \leftarrow \llbracket e_n \rrbracket \textbf{ in} \\ \mathbf{y_n} \end{array}$$

The allocation pass changes the representation of closures in FCC to a location in memory, where the first cell contains the code pointer, and subsequent cells are the values for the environment. All other syntactic constructs, including our closure abstraction, would be translated to the same syntactic construct in the target language. The target language must have the same abstractions for closures as FCC to preserve the abstractions (and thus types).

The main challenge in extending type preservation from FCC to explicit allocation is our use of the singleton type, a dependent type. Explicit allocation necessarily introduces effects, which can be difficult to combine with dependent types.

Effects. Effects can introduce two problems for FCC.

First, effects such as allocation and mutability cause inconsistencies when mixed with dependent types; this is well known, and described well by Pédrot and Tabareau [2020]. In particular, effects can disrupt subject reduction, since a type may depend on a value, which could change due to effects. However, we conjecture FCC can be extended with these effects and maintain subject reduction, because of the value restrictions we introduce on the singleton type. While the underlying values in the environment of a closure may change, we will only depend on the immutable label identifying that closure. We will need to ensure that changes to the underlying data do not affect type safety, but this should be type safe if we do not have strong updates, and if we do not have any additional computations in our dependent types (such as the ability to dereference a label in the type system).

Second, to closure convert a source language with mutable variables, FCC must be extended with support for mutability. Dybvig [1987] builds on the work of Cardelli [1984] by supporting mutable variables in flat closures. Unlike in ML, where the type system ensures that mutable variables are explicitly boxed by the programmer, Scheme variables may or may not be mutable. Dybvig [1987] solves this by allocating boxes for mutable free variables in closures, to avoid the substantial cost of hunting down where the variable originated through the call chain. Box allocation is handled by callees, as callers do not know which free variables may be mutated. Each procedure constructs a box for each variable mutated in its body before executing the body.

Dependent Types. We conjecture that our typed closure representation can be modified to work with for a dependently typed language IL. Bowman and Ahmed [2018] observe that the existential type for closures does not work to compile a dependently typed source language because it relies on impredicativity, which conflicts with large elimination and computation relevance (and we want closures to be computationally relevant). We do not rely on existential types, avoiding a key problem with dependent types.

One other problem adapting to dependent types is our use of the recursive type, which could introduce non-termination and inconsistency into a dependent type theory. However, we conjecture that the recursive type could be replaced by a $\top$ type and an additional cast operation defined below. The recursive type enables optimizations of recursive closures, but these optimizations are not relevant in the dependently typed case due to restrictions on recursion in dependently-typed languages.

Recall from Section 2 that the recursively closure type is used to enable optimization for recursive escaping closures. However, escaping recursive references are typically impossible in dependently-typed languages. Using the standard syntactic guard condition, recursive functions must appear in operator position and called on a structurally smaller argument, so all recursive closures are known at their call sites. This means we never have an escaping recursive closure, and never have the additional allocation overhead.

We conjecture we could cast the code pointer to $\top$, a super type of all types with no elimination form, when passing a closure as environment. The ensures that no recursion is possible through the code pointer field of the closure; instead, all recursion must occur directly through the procedure label. A closure's representation would have type $\mathsf{Rec}(\mathsf{Code}(\mathsf{Rec}(\top, \vec{C}), A) \to B)$, while an environment would have type $\mathsf{Rec}(\top, \vec{C})$, where the code pointer's type has been forgotten. The closure would be cast to an environment with env, defined below, before being passed as the environment, forgetting the code pointer and breaking the cycle. Other closure typing rules would need to be slightly modified, such as the accept-clos form.

$$\frac{\Gamma; \Delta \vdash e : \mathsf{Rec}(A, \vec{B})}{\Gamma; \Delta \vdash \mathsf{env}(e) : \mathsf{Rec}(\top, \vec{B})} \qquad \frac{\Gamma; \Delta \vdash e : \mathsf{Code}(\mathsf{Rec}(\top, \vec{C}), \vec{A} \to B) \quad \Gamma; \Delta \vdash v : \mathsf{Rec}(\mathsf{Code}(\mathsf{Rec}(\top, \vec{C}), \vec{A} \to B), \vec{C})}{\Gamma; \Delta \vdash \mathsf{accept\text{-}clos}(e) : \mathsf{Code}(\mathsf{The}(\mathsf{clos}(v)), \vec{A} \to B)}$$

Eliminating Abstract Closures. Our representation of closures is still a new custom primitive, but we would like high-level concepts like a closure to be expressible as a design pattern over general-purpose type formers. Minamide et al. [1996a] separate closure conversion into two passes: abstract closure conversion, in which closures are made explicit (but abstract), and closure representation, in which both closure representation and abstraction are completely encoded with low-level general-purpose features, particularly pairs, procedure pointers, and the existential type. Our closure abstraction still relies on a new special purpose primitive type Clos.

We might be able to eliminate the abstract closure type *and* express all our optimizations with a different general-purpose type former: row-polymorphic records. Using existential row variables $\exists\rho$ to quantify over parts of a record [Harper and Pierce 1991; Morris and McKinna 2019; Wand 1991a,b], we could define our closure type as $\mathsf{Clos}(\vec{A}) \to B \overset{\text{def}}{=} \exists\rho.\mu\alpha.\mathsf{Rec}((\mathsf{Code}(\mathsf{Rec}(\alpha|\rho), \vec{A}) \to B)|\rho)$. In this syntax, a record $\mathsf{Rec}(\vec{A}|\rho)$ contains at least a $\mathit{length}(\vec{A})$ entries of types $\vec{A}$, but may contain some unknown other entries ρ. Expressing the closure type requires existential quantification over row variables. In this view, by hiding the environment row, there should be only two operations possible: project out the code pointer, and call the code pointer on the closure and its arguments.

However, this reintroduces existential quantification over types, and therefore possibly impredicativity. If we want to extend this translation to dependent types, we'd like the interpretation of this to be a second class existential quantification. We must ensure that the existential row variable does not add impredicativity to the type system, and that it is still computationally irrelevant. Row polymorphism has been investigated in the context of a restricted class of dependent types lacking large elimination [Chlipala 2010], but not existential row quantification, and leaves unstudied the interaction with impredicativity. Given the unknowns, while abstract closures are known to work in dependent type theory [Bowman and Ahmed 2018], we leave investigating expressing closure abstraction as existential row polymorphism for future work.

Performance Evaluation. A limitation in this, and most type-preserving compilation work, is the lack of performance evaluation. In our work, as is often the case for type preservation, we consider performance evaluation out of scope for this paper. Before we can implement a type preserving compiler, we need to know how to design it, after all. Even then, there are many variables that interfere with a direct comparison between the performance of a compiler designed to use FCC compared to, *e.g.,* Chez Scheme (in which Keep et al. [2012] implement their optimizations).

The biggest difference is the use of typed ILs. The typed ILs are very likely to decrease compile-time performance, since they increase the size of program representation during compilation, and require generating additional annotations to guarantee decidable checking. The typed IL may also decrease run-time performance for reasons unrelated to the closure conversion itself; even if we achieve every optimization that Keep et al. [2012] do, types may interfere with other optimizations. In fact, that's likely: this work is addressing just such a limitation in past work on type-preserving compilation! It seems likely there exist similar limitations in passes other than closure conversion. Given FCC's correspondence to past work on untyped efficient closures, we think that the main overhead would be the cost of typechecking during compilation, but it is possible that FCC's types inhibit some optimizations that we did not study in this paper.

The most direct measurement of FCC itself would be to design and implement two type-preserving compiler differing only in the use of FCC or ACC for closure conversion. However, this requires an existing theory for FCC, and substantial additional work besides, such as how to preserve types from FCC to a typed assembly language. Such additional work is clearly necessary for a thorough investigation of the performance of a type-preserving compiler, but out of scope for the time being.

7 Conclusion

Type preservation offers lightweight correctness guarantees for compilers, but it should not sacrifice performance. In FCC, we strive to ensure we can express many of the closure optimizations and the efficient representation used in practice in a typed intermediate language, ensuring safety and performance. FCC takes lessons from much of the past work on typed closure conversion, combining ideas that work well to solve individual problems into a single language that solves many problems. Particularly, FCC combines recursive closure types, recursive code, singleton types (translucent types), and abstract closure types, paying particular attention to how these are likely to work with extensions to the language found in the literature. We're able to validate many optimizations found in practice as type safe, although there remains some important future work related to global variables and constants.

Acknowledgments

We gratefully acknowledge the feedback of the many reviewers of this work. We also thank Andy Keep for feedback on early iterations of this work.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-2019-04207. Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), numéro de référence RGPIN-2019-04207.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. NN66001-22-C-4027. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or NIWC Pacific.

Data Availability Statement

The software artifact with a mechanized Redex model of the FCC type system has been made publically available at Geller et al. [2025].

References

Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *International Conference on Functional Programming (ICFP).* https://doi.org/10.1145/1411204.1411227

Andrew W. Appel. 2006. *Compiling with Continuations* (second ed.). Cambridge University Press.

William J. Bowman and Amal Ahmed. 2018. Typed Closure Conversion for the Calculus of Constructions. In *International Conference on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3192366.3192372

Luca Cardelli. 1984. Compiling a functional language. In *LISP and Functional Programming (LFP).* https://doi.org/10.1145/800055.802037

Adam Chlipala. 2010. Ur: statically-typed metaprogramming with type-level record computation. In *International Conference on Programming Language Design and Implementation (PLDI)*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 122–133. https://doi.org/10.1145/1806596.1806612

Karl Crary, Stephanie Weirich, and Greg Morrisett. 2002. Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming (JFP)* (2002). https://doi.org/10.1017/S0956796801004282

R. Kent Dybvig. 1987. *Three Implementation Models for Scheme.* Ph. D. Dissertation. Chapel Hill. https://www.cs.unm.edu/~williams/cs491/three-imp.pdf

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex.* MIT Press. https://mitpress.mit.edu/9780262062756/

Adam T. Geller, Sean Bocirnea, Chester J. F. Gould, Paulette Koronkevich, and William J. Bowman. 2025. *Type-Preserving Flat Closure Optimization Artifact.* https://doi.org/10.5281/zenodo.14941604

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *International Conference on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3062341.3062363

Robert Harper. 1996. A Note on "A Simplified Account of Polymorphic References". *Inf. Process. Lett.* 57, 1 (1996), 15–16. https://doi.org/10.1016/0020-0190(95)00178-6

Robert Harper and Benjamin C. Pierce. 1991. A Record Calculus Based on Symmetric Concatenation. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, David S. Wise (Ed.). ACM Press, 131–142. https://doi.org/10.1145/99583.99603

Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. 2012. Optimizing closures in O(0) time. In *Scheme and Functional Programming Workshop (Scheme)*, Olivier Danvy (Ed.). ACM, 30–35. https://doi.org/10.1145/2661103.2661106

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run your research: on the effectiveness of lightweight mechanization. In *Symposium on Principles of Programming Languages (POPL).* ACM. https://doi.org/10.1145/2103656.2103691

András Kovács. 2018. Closure Conversion for Dependent Type Theory with Type-Passing Polymorphism. In *International Workshop on Types for Proofs and Programs (TYPES).* https://github.com/AndrasKovacs/misc-stuff/blob/master/MemControl/types2018/abstract-types-2018-cconv.pdf

Xavier Leroy. 1992. Unboxed objects and polymorphic typing. In *Symposium on Principles of Programming Languages (POPL).* https://doi.org/10.1145/143165.143205

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (Nov. 2009). https://doi.org/10.1007/s10817-009-9155-4

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996a. Typed Closure Conversion. In *Symposium on Principles of Programming Languages (POPL).* https://doi.org/10.1145/237721.237791

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996b. *Typed Closure Conversion.* techreport. https://www.cs.cmu.edu/~rwh/papers/closures/tr.pdf

J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.* 3, POPL (2019), 12:1–12:28. https://doi.org/10.1145/3290325

Greg Morrisett. 1995. *Compiling with Types.* Ph. D. Dissertation. Carnegie Mellon University. https://www.cs.cmu.edu/~rwh/theses/morrisett.pdf

Greg Morrisett and Robert Harper. 1998. Typed Closure Conversion for Recursively-Defined Functions. *Electronic Notes in Theoretical Computer Science* 10 (June 1998). https://doi.org/10.1016/s1571-0661(05)80702-9

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (May 1999). https://doi.org/10.1145/319301.319345

Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation via Universal Embedding. In *International Conference on Functional Programming (ICFP).* https://doi.org/10.1145/2951913.2951941

Pierre-Marie Pédrot and Nicolas Tabareau. 2020. The fire triangle: how to mix substitution, dependent elimination, and effects. In *Symposium on Principles of Programming Languages (POPL).* https://doi.org/10.1145/3371126

Simon L. Peyton Jones. 1996. Compiling Haskell by Program Transformation: A Report from the Trenches. In *European Symposium on Programming (ESOP).* https://doi.org/10.1007/3-540-61055-3_27

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press Ltd.

Manuel Serrano. 1995. Control flow analysis: a functional languages compilation paradigm. In *Symposium on Applied Computing (SAC).* https://doi.org/10.1145/315891.315934

Zhong Shao and Andrew W. Appel. 1994. Space-Efficient Closure Representations. In *Conference on LISP and Functional Programming (LFP).* https://doi.org/10.1145/182409.156783

Guy Lewis Steele Jr. 1978. *RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization).* Technical Report 474. MIT Artificial Intelligence Lab. https://dspace.mit.edu/bitstream/handle/1721.1/6913/AITR-474.pdf

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Symposium on Principles of Programming Languages (POPL).* https://doi.org/10.1145/2676726.2676985

David Tarditi. 1996. *Design and implementation of code optimizations for a type-directed compiler for Standard ML.* Ph. D. Dissertation. Carnegie Mellon University. https://csd.cmu.edu/sites/default/files/phd-thesis/CMU-CS-97-108.pdf

David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. 1996. TIL: A Type-Directed Optimizing Compiler for ML. In *International Conference on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/231379

Mitchell Wand. 1991a. Correctness of procedure representations in higher-order assembly language. In *International Conference on Mathematical Foundations of Programming Semantics.* Springer, 294–311. https://doi.org/10.1007/3-540-55511-0_15

Mitchell Wand. 1991b. Type Inference for Record Concatenation and Multiple Inheritance. *Inf. Comput.* 93, 1 (1991), 1–15. https://doi.org/10.1016/0890-5401(91)90050-C

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

Received 2024-10-16; accepted 2025-02-18