A Tale of Two Zippers

Philip Wadler
IOG and University of Edinburgh
Edinburgh, United Kingdom
philip.wadler@iohk.io

Ramsay Taylor
IOG
Sheffield, United Kingdom
ramsay.taylor@iohk.io

Jacco O.G. Krijnen
Utrecht University
Utrecht, Netherlands
j.o.g.krijnen@uu.nl

Abstract

We apply the zipper construct of Huet to prove correct an optimiser for a simply-typed lambda calculus with force and delay. The work here is used as the basis for a certifying optimising compiler for the Plutus smart contract language on the Cardano blockchain.

The paper is an executable literate Agda script, and its source may be found in the file

`Zippers.lagda.md`

available as an artifact associated with this paper.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → **Type theory**; **Program semantics**.

Keywords: Compilers, Optimisation, Formal Methods, Lambda Calculus, Agda

ACM Reference Format:
Philip Wadler, Ramsay Taylor, and Jacco O.G. Krijnen. 2025. A Tale of Two Zippers. In *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday (OLIVIERFEST '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3759427.3760383

Dedicated to Olivier Danvy on the occasion of his 64th birthday.

1 Introduction

It was the best of proofs, it was the worst of proofs.

Bad news for crypto investors is good news for formal methods researchers: smart contracts for cryptocurrencies regularly suffer exploits costing tens of millions of dollars, which makes a business case for applying formal methods.

This is the tale of one such application. The Cardano blockchain, created by IOG, uses a variant of System F as its on-chain smart contract language. System F was chosen in order to future-proof the system: if you want a language that will still be viable in fifty years, pick one that is fifty years old. The Cardano variant of System F is called TPLC (Typed PLutus Core). Formal methods are used throughout: TPLC is specified in Agda, including an operational semantics with a proof of progress and preservation (Chapman et al 2019). The constructive proof provides a way to execute TPLC in Agda, which is used to test the production interpreter against the formal specification.

To minimise the size of transactions (and hence reduce their cost) the blockchain uses an untyped variant of TPLC, referred to as UPLC (Untyped PLutus Core). Source programs are written in a call-by-value variant of Haskell, referred to as Plinth (formerly Plutus Tx). To further ensure reliability, formal methods are applied again. We are not developing a verified compiler, as that is still too costly, but we are developing a certifying compiler. We consider a portion of the compiler consisting of a sequence of passes, translating UPLC to optimised UPLC. Each optimisation pass generates a certificate, which is an instance of a relation between unoptimised source and optimised target code. Agda is used to prove, once and for all, that the certification relation preserves the semantics between source and target. The certificate makes it easy to check that the code produced by the compiler satisfies the certification relation, and hence is sound.

This paper is concerned with the certification of one pass of the compiler. When erasing TPLC to UPLC, type abstractions and applications are converted to `delay` and `force` operations, respectively. The compiler includes a pass that optimises in the obvious case where `force` is applied directly to `delay`.

```
force (delay M)  ⟶  M
```

The pass also optimises when instances of `force` and `delay` are separated by one or more applications.

```
force ((ƛ (delay N)) · M)  ⟶  (ƛ N) · M
```

Here `M` and `N` are terms of UPLC, `(ƛ N)` denotes lambda abstraction using de Bruijn indices, and `_·_` denotes application.

As we will see below, it is easy to specify equivalence between terms, which we write `M ~ N`. The obvious thing is to use `_~_` itself as the certification relation, but this is impractical. Given `M` and `N`, building a decision procedure to determine whether `M ~ N` is difficult. There is far too large a search space, mainly because equivalence is transitive so at any point the search to verify `M ~ N` may split into multiple searches, to find an `L` for which `M ~ L` and `L ~ N`.

The story begins when one of us (Philip) was presented by another of us (Ramsay) with the specification of the certification relation for this pass of the compiler. The relation

OLIVIERFEST '25, Singapore, Singapore

ACM ISBN 979-8-4007-2150-2/25/10
https://doi.org/10.1145/3759427.3760383

requires two counters, represented as natural numbers. One counts the number of `force` constructs encountered, while the other counts the number of applications stepped under while looking for the corresponding `delay`. It was confirmed by testing that the certification relation accurately captures the behaviour of this pass of the compiler. Further it is easy to write a decision procedure to check that the source and target satisfy the two-counter relation. However, it was not at all clear how to show that source and target terms satisfying the relation were necessarily equivalent.

The solution turned out to be to replace the two counters by a zipper. Zippers were introduced by Huet 1997 as a way to navigate through a data structure. In our case, the zipper represents an evaluation context threaded inside out. We need two zippers, one for the source and one for the target. Again, it is easy to write a decision procedure to check that source and target satisfy the two-zipper relation. It is also easy to design a second relation, an equivalence that specifies when two terms have identical behaviours. Further, it is now easy to specify how the two-zipper relation corresponds to the equivalence relation, and to show that the first implies the second, guaranteeing soundness.

The remainder of this paper develops our formulation as a literate Agda script. To focus on essentials, we use a simply-typed lambda calculus rather than TPLC or UPLC. Every line of Agda code is included in this paper, and the source is provided as an artifact (Wadler *et al* 2025). Code in colour has been type-checked by the Agda system, so you can be confident it is correct. A reviewer of the paper wondered why it was necessary to include *all* the code, even imports and trivial inductions? Too often we've read papers that we could decipher only by reading the accompanying code. By including all executable code in the paper, we guarantee nothing needed has been omitted. We encourage others to try this style—there is value in learning how to communicate clearly without elision.

The paper is intended to be accessible to anyone with a passing knowledge of proof assistants. Additional detail on how to formalise proofs in Agda can be found in the textbook *Programming Language Foundations in Agda* (henceforth PLFA), by Wadler, Kokke, and Siek (2018). Additional information on PLFA itself can be found in a paper by Kokke, Siek, and Wadler (2020). A previous paper by Wadler (2024) is also written as a literate Agda script, and also describes a simply-typed lambda calculus. In a few places we have borrowed phrasing from that paper or from PLFA.

Anyone familiar with Olivier Danvy's body of publications will see his inspiration here. Zippers are related to evaluation contexts, which play a key role in his work on refocusing (see, e.g., Danvy and Nielsen 2004), and he has also been known to publish literate scripts from a proof assistant (in his case, Rocq, as in Danvy 2023). We don't claim this paper is as elegant as one of Olivier's, but hope it will be of interest nonetheless.

2 Module Header

We begin with bookkeeping: the module header and imports from the Agda standard library. Agda supports mixfix syntax, so `_≡_` names infix equality.

```
module Zippers where
open import Data.Nat using (ℕ; zero; suc)
open import
  Relation.Binary.PropositionalEquality
  using (_≡_; refl)
open import Data.Empty using (⊥)
```

3 Operator Priorities

We declare in advance binding priorities for infix, multifix, prefix, and postfix operators. A higher number indicates tighter binding, and letters `l` or `r` indicate left or right associativity.

```
infix  4 _∋_ _⊢_ _⊆_ _⊢_↝_ _~_
infixl 5 _,_
infix  5 ƛ_
infixl 7 _·_
infixr 7 _⇒_
infix  8 `_
infix  9 S_
```

4 Lambda Calculus with Delay and Force

We model the source and target language for our optimiser as a simply-typed lambda calculus with delay and force. The delay and force operations are introduced when erasing TPLC (a variant of System F) to UPLC, to represent type abstractions and type applications. The delay operation corresponds to a lambda abstraction over the unit type, and the force operation corresponds to an application to a unit value. For the actual Plutus toolchain, the typed language erases to an untyped language containing delay and force. Here we use a simply-typed language containing delay and force. Requiring that the relations we use preserve types provides a sanity check. Because we use intrinsic typing (see below), this sanity check comes almost for free.

Lambda calculus is typically formulated using named variables and extrinsic typing rules, but when using a proof assistant it is often more convenient to use de Bruijn indices and intrinsic typing rules. With named variables the Church numeral two is written `λs.λz.(s·(s·z))`, whereas with de Bruijn indices it is written `λλ(1·(1·0))`. In the latter, variables names do not appear at point of binding; instead each variable is replaced by a count (starting at zero) of how many binders outward one must step over to find the one that binds this variable. With extrinsic typing, one first gives a syntax of pre-terms and then gives rules assigning types to terms, while with intrinsic typing the syntax of terms and the type rules are defined together. Reynolds (2000) introduced the names intrinsic and extrinsic; the distinction between

the two is sometimes referred to as Curry-style (terms exist prior to types) and Church-style (terms make sense only with their types). The formulation using de Bruijn indices and intrinsic types was first proposed by Altenkirch and Reus (1999). A textbook development in Agda of both the named-variable/extrinsic and the de Bruijn/intrinsic style can be found in PLFA.

We model concepts of interest, such as the types, contexts, variables, and terms of our calculus, as inductive types. In Agda, inductive types are introduced with `data` declarations, similar to those found in Haskell. When we write `Type : Set`, note that `Type` stands for the types of our typed calculus, whereas `Set` is the type of all types of the meta-language Agda—be careful not to confuse the two! Similarly, term can mean either a term of our calculus or a term of the meta-language Agda—again, be careful not to confuse them.

(Readers familiar with Girard's paradox (Girard 1972) will know that taking `Set : Set` leads to a contradiction. Instead Agda has a hierarchy with $\texttt{Set}_0$ `:` $\texttt{Set}_1$, $\texttt{Set}_1$ `:` $\texttt{Set}_2$ and so on. Fortunately, `Set`—an abbreviation for $\texttt{Set}_0$—will suffice for our purposes.)

We let `A`, `B`, `C` (and the same primed) range over types. A type is either a base type `ι`, a function type `A ⇒ B`, or the type of a delayed term, `Delay A`.

```
data Type : Set where

  ι :
        ----
        Type

  _⇒_ :
        (A : Type)
        (B : Type)
    →   ----------
        Type

  Delay :
        (A : Type)
    →   ----------
        Type

variable
  A B C A′ B′ C′ : Type
```

Two dashes indicate the beginning of a comment in Agda, as in Haskell. We take advantage of this to make our declarations mimic the corresponding inference rules. They would mimic them even more closely if the rule for `_⇒_` were instead written, say,

```
  (A : Type)
  (B : Type)
→ --------------
  (A ⇒ B : Type)
```

but that would not be legal Agda.

We let `Γ`, `Δ` range over contexts. Contexts are essentially lists of types, with new types added at the right. A context is either the empty context `∅` or an extended context `Γ , A`.

```
data Context : Set where

  ∅ :
        -------
        Context

  _,_ :
        (Γ : Context)
        (A : Type)
    →   -------------
        Context

variable
  Γ Δ : Context
```

We let `u`, `v`, `x`, `y` range over variables, and write `x : Γ ∋ A` to indicate that in context `Γ` the variable `x` has type `A`. Variables are de Bruijn indices, where `Z` stands for zero (the variable at the right end of the context), and `S` stands for successor (skipping over the variable at the right end of the context).

```
data _∋_ : Context → Type → Set where

  Z :
       ---------
       Γ , A ∋ A

  S_ :
       (x : Γ ∋ B)
    →  -----------
       Γ , A ∋ B

variable
  u v x y : Γ ∋ A
```

We let `L`, `M`, `N` (and the same primed) range over terms, and write `M : Γ ⊢ A` to indicate that in context `Γ` the term `M` has type `A`. A term is either a variable (`′ x`), an abstraction (`ƛ N`), an application (`L · M`), a delay (`delay M`), or a force (`force M`).

```
data _⊢_ : Context → Type → Set where

  `_ :
       (x : Γ ∋ A)
    →  -----------
       Γ ⊢ A

  ƛ_ :
       (N : Γ , A ⊢ B)
    →  ---------------
       Γ ⊢ (A ⇒ B)

  _·_ :
       (L : Γ ⊢ A ⇒ B)
       (M : Γ ⊢ A)
    →  ---------------
```

```
      Γ ⊢ B

  force :
      (M : Γ ⊢ Delay A)
    → ------------------
      Γ ⊢ A

  delay :
      (M : Γ ⊢ A)
    → -----------
      Γ ⊢ Delay A

variable
  L L′ M M′ N N′ : Γ ⊢ A
```

We write `LET M N` as equivalent to `(ƛ N) · M`. For this to be well-typed we should have `M : Γ ⊢ A` and `N : Γ , A ⊢ B`. Using a `pattern` declaration means that `LET` may appear anywhere a constructor may appear.

```
pattern LET M N = (ƛ N) · M
```

We use the following as a running example.

```
Γ₀ : Context
Γ₀ = ∅ , ι ⇒ ι ⇒ ι , ι , ι

M₀ M₀′ : Γ₀ ⊢ ι
M₀ =
  force
    ((ƛ ƛ (delay (` S S S S Z · ` Z · ` S Z)))
      · ` S Z
      · ` Z)
M₀′ =
  (ƛ ƛ (` S S S S Z · ` Z · ` S Z))
     · ` S Z
     · ` Z
```

If we were using named variables instead of de Bruijn indices, term `M₀` would instead be written as

```
M₀ : ∅ , f : ι ⇒ ι ⇒ ι , u : ι , v : ι ⊢ ι
M₀ =
  force
   ((ƛ x ⇒ ƛ y ⇒ (delay (` f · ` y · ` x)))
    · ` u
    · ` v)
```

Note we translate `f` to `S S S S Z` rather than `S S Z` because there are two `ƛ` bindings between `f` and the surrounding context.

5 Renaming and Weakening

Weakening asserts that if a term is well-typed in a context then it is also well-typed in that context extended with an additional variable. Hence if `L : Γ ⊢ B` then `wk L : Γ , A ⊢ B`.

We will introduce equivalences between terms in Section 10. One equivalence we require involves passing an application inside a let binding. We take `(LET M N) · L` to be equivalent to `LET M (N · wk L)`. Here `M : Γ ⊢ A` and `N : Γ , A ⊢ B ⇒ C` and `L : Γ ⊢ B`, and hence `wk L : Γ , A ⊢ B`, which is what is required for the second term in the equivalence to be well-typed.

We let ρ range over renamings, and write `Γ ⊆ Δ` to denote a renaming from Γ to Δ. A renaming is a function that takes every variable in Γ to one in Δ, so if `x : Γ ∋ A` then `ρ x : Δ ∋ A`.

```
_⊆_ : Context → Context → Set
Γ ⊆ Δ = ∀ {A} → Γ ∋ A → Δ ∋ A
```

If we have a renaming `ρ : Γ ⊆ Δ` then `ren ρ` converts a term over context Γ to one over context Δ, that is, if `M : Γ ⊢ A` then `ren ρ M : Δ ⊢ A`. The case for renaming a lambda term `ƛ N : Γ ⊢ A ⇒ B` requires a way to extend a renaming `ρ : Γ ⊆ Δ` to apply to the body of the lambda term `N : Γ , A ⊢ B`. This is provided by the function `ext` (short for extend), which gives `ext ρ : Γ , A ⊆ Δ , A`.

```
ext :
    (ρ : Γ ⊆ Δ)
  → -------------
    Γ , A ⊆ Δ , A
ext ρ Z     = Z
ext ρ (S x) = S (ρ x)

ren :
    (ρ : Γ ⊆ Δ)
    (M : Γ ⊢ A)
  → -----------
    Δ ⊢ A
ren ρ (` x)      = ` ρ x
ren ρ (ƛ N)      = ƛ ren (ext ρ) N
ren ρ (L · M)    = ren ρ L · ren ρ M
ren ρ (force M) = force (ren ρ M)
ren ρ (delay M) = delay (ren ρ M)
```

With these definitions under our belt, it is easy to define weakening. Note that `S_ : Γ ⊆ Γ , A` is a renaming, where `S_` is the successor function on de Bruijn indices. Hence we can weaken the term `L : Γ ⊢ B` by computing `ren S_ L : Γ , A ⊢ B`.

```
wk :
    (L : Γ ⊢ B)
  → -----------
    Γ , A ⊢ B
wk L = ren S_ L
```

Here is a simple example.

```
L₀ : ∅ , ι ⇒ ι , ι ⊢ ι
L₀ = ` S Z · ` Z

L₁ : ∅ , ι ⇒ ι , ι , ι ⊢ ι
L₁ = ` S S Z · ` S Z

_ : wk L₀ ≡ L₁
_ = refl
```

In Agda, one may use `_` as a dummy name that is convenient for examples. If `L` and `M` are terms, one writes `L ≡ M` to stand for the judgement that `L` and `M` are identical. The only constructor for this type is `refl`, short for *reflexive*, and we have `refl : L ≡ L` for any term `L`. The last line typechecks because Agda computes that `wk L₀` evaluates to `L₁`.

6 Relations over Terms and Compatible Closure

Under the propositions-as-types interpretation, a proposition is represented by a type in the meta-language; recall that in Agda the type of types is written `Set`. A proposition is represented by a set of all its possible proofs; a value of this set is referred to as *evidence* for the proposition. If the set is empty the proposition is false, while if the set is non-empty the proposition is true. In particular, if `X` and `Y` are types corresponding to propositions, then `X → Y` corresponds to implication—it takes every proof of `X` into a proof of `Y`.

If `Γ ⊢ A` is the type of terms, then `Γ ⊢ A → Γ ⊢ A → Set` specifies a relation between two terms in the same context with the same type. If the set is non-empty then the relation holds, while if the set is empty then the relation does not hold. We let `R` range over such relations.

```
Relation = ∀ {Γ A} → Γ ⊢ A → Γ ⊢ A → Set

variable
  R : Relation
```

The *compatible closure* of a relation over terms is the smallest relation that contains the original relation and is closed over the syntactic constructs of the language. If `R` is a relation over terms we write `CC R` for its compatible closure. Here `CC` is declared as an inductive type. Constructor `base` injects the original relation into the compatible closure, so that if `s` is a proof that `R M M′` holds, then `base s` is a proof that `CC R M M′` holds. There is also one constructor of the compatible closure for each constructor of the term type. If `x` is a variable then relation `(′ x)` relates term `(′ x)` to itself. Similarly, if `t` relates `N` to `N′` then `(ƛ t)` relates `(ƛ N)` to `(ƛ N′)`. Similarly also, if `r` relates `L` to `L′` and `s` relates `M` to `M′` then `r · s` relates `L · M` to `L′ · M′`. And again, if `s` relates `M` to `M′` then `delay s` relates `delay M` to `delay M′` and `force s` relates `force M` to `force M′`. Here we use the fact that Agda supports overloading of constructors to use `′_`, `ƛ_`, `_·_`, `delay`, `force` as constructors of both the type `Γ ⊢ A` and the type `CC R M M′`.

```
data CC (R : Relation) : Relation where

  base :
      (s : R M M′)
    → ------------
      CC R M M′

  `_ :
      (x : Γ ∋ A)
    → ---------------
      CC R (` x) (` x)

  ƛ_ :
      (t : CC R N N′)
    → -----------------
      CC R (ƛ N) (ƛ N′)

  _·_ :
      (r : CC R L L′)
      (s : CC R M M′)
    → ----------------------
      CC R (L · M) (L′ · M′)

  delay :
      (s : CC R M M′)
    → ------------------------
      CC R (delay M) (delay M′)

  force :
      (s : CC R M M′)
    → ------------------------
      CC R (force M) (force M′)
```

It is easy to show that the compatible closure is reflexive. Regardless of `R`, we have that `CC R M M` holds for every term `M`. The proof is an easy induction over the structure of `M`.

```
reflCC : ∀ (M : Γ ⊢ A) → CC R M M
reflCC (` x)      = ` x
reflCC (ƛ N)      = ƛ (reflCC N)
reflCC (L · M)    = reflCC L · reflCC M
reflCC (force M) = force (reflCC M)
reflCC (delay M) = delay (reflCC M)
```

Here is a simple example. Let `R₀` be the empty relation, that relates no term to any other. Recall that we introduced `L₀` in Section 5. A term is compatible with itself even in the compatible closure of the empty relation.

```
R₀ : Relation
R₀ M M′ = ⊥

_ : L₀ ≡ ` S Z · ` Z
_ = refl

_ : CC R₀ L₀ L₀
_ = reflCC L₀

_ : reflCC {R = R₀} L₀ ≡ ` S Z · ` Z
_ = refl
```

Agda can usually use type inference to determine implicit arguments such as `R`, but here we need to instantiate `R` to `R₀` explicitly. Because of constructor overloading, both `L₀` and `reflCC L₀` are written with the same constructors, as follows: `` ` S Z · ` Z ``.

7 Force-Delay Relation with Two Counters

We now describe the original force-delay relation with two counters.

We write `FDnat m n M M′` to indicate that term `M` relates to term `M′`, where `m` counts the number of `force` operations encountered so far, `n` counts the number of applications encountered so far, and `M` and `M′` are the terms related. Intuitively, both terms have the same lambda abstractions and applications, but matching instances of force and delay in term `M` have been removed in term `M′`. The two terms have the same context but may have different types, because one may have a `force` or `delay` applied when the other does not.

The relation is defined by five inference rules.

```
variable
  m n : ℕ

data FDnat :
       (m : ℕ) (n : ℕ)
       (M : Γ ⊢ A) (M′ : Γ ⊢ A′)
     → Set where

  cc :
       (s : CC (FDnat zero zero) M M′)
    →  ------------------------------
       FDnat zero zero M M′

  force :
       (s : FDnat (suc m) n M M′)
    →  -------------------------
       FDnat m n (force M) M′

  delay :
       (s : FDnat m n M M′)
    →  ---------------------------
       FDnat (suc m) n (delay M) M′

  app :
       (s : FDnat zero zero M M′)
       (r : FDnat m (suc n) L L′)
    →  -------------------------
       FDnat m n (L · M) (L′ · M′)

  abs :
       (t : FDnat m n N N′)
    →  ---------------------------
       FDnat m (suc n) (ƛ N) (ƛ N′)
```

Rule `cc` states that the relation is compatibly closed when both counters are zero. Force and delay must match to enable the optimisation, and similarly for applications and abstractions; if the optimisation does not apply, then compatible closure is used instead. Terms are traversed from outside to inside. Rule `force` states that every time we encounter a force we increment the first counter, and rule `delay` states that every time we encounter a delay we decrement the first counter. Instances of force and delay are cancelled by the optimisation, so appear in the left term but not the right in these two rules. Rule `app` states that every time we encounter an application we increment the second counter, and rule `abs` states every time we encounter a lambda abstraction we decrement the second counter. Applications and lambda abstractions are carried over by the optimisation, so appear in both the left term and the right in these two rules. In the application rule, note that functions are related by `FDnat m n L L′`, while arguments are related by `FDnat zero zero M M′`.

Experimentation shows that the `FDnat` relation adequately characterises the behaviour of the optimisation. That is, if the optimisation is given source term `M` and produces target term `M′`, then `FDnat zero zero M M′` holds. Further, it proves easy to write a decision procedure to determine whether `FDnat` holds for given counters and terms, so it is feasible to use it as a certification relation for verified compilation.

Here is an example.

```
FDnat₀ : FDnat zero zero M₀ M₀′
FDnat₀ =
  force
    (app (cc (` Z))
      (app (cc (` S Z))
        (abs (abs (delay (cc
          (` S S S S Z · ` Z · ` S Z)))))))
```

In fact, Agda computed this example for us. When creating this script interactively, the right-hand side of the equation was given as `?`.

```
FDnat₀ = ?
```

This causes Agda to generate a hole.

```
FDnat₀ = { }0
```

We can ask Agda to fill in the hole automatically by typing `^C ^A`. A little search is required because of potential overlap between the `cc` rules and the other rules, and it times out after one second. We can up the search time to two seconds by filling the hole with the `-t` flag.

```
FDnat₀ = { -t 2 }0
```

Now Agda generates the term above, showing that not much search is required—it is easy to decide whether the relation holds.

Unfortunately, it proved difficult to show this relation sound, that is, that related terms possess identical behaviour. The problem is that the counters throw too much information away. When an application occurs we need to remember the corresponding arguments. We also want to keep track of the interleaving between instances of force and instances of application. It turns out that zippers are perfect for this purpose.

8 Zippers

We write `Γ ⊢ A ↝ B` for a zipper in context `Γ` that wraps a term of type `A` to yield a term of type `B`. We let `z` (and the same primed) range over zippers. A zipper is either the empty zipper `□`, an application zipper `z · M`, or a force zipper `force z`.

```
data _⊢_↝_ : Context → Type → Type → Set where

  □ :
      ---------
      Γ ⊢ A ↝ A

  _·_ :
      (z : Γ ⊢ B ↝ C)
      (M : Γ ⊢ A)
    → ---------------
      Γ ⊢ A ⇒ B ↝ C

  force :
      (z : Γ ⊢ A ↝ C)
    → ---------------
      Γ ⊢ Delay A ↝ C

variable
  z z′ : Γ ⊢ A ↝ B
```

Given a zipper `z : Γ ⊢ A ↝ B` and a term `M : Γ ⊢ A` we can plug the latter into the former, yielding a term `zip z M : Γ ⊢ B`. Because zippers are inside-out evaluation contexts, the `zip` operation is written as a tail recursion, stripping applications and forces off the zipper and adding them to the term.

```
zip : Γ ⊢ A ↝ B → Γ ⊢ A → Γ ⊢ B
zip □       L   = L
zip (z · M) L   = zip z (L · M)
zip (force z) L = zip z (force L)
```

Just as we need weakening for terms we also need weakening for zippers. Weakening for zippers asserts that if a zipper is well-typed in a context then it is also well-typed in that context extended with an additional variable. Hence if `z : Γ ⊢ A ↝ B` then `wkᶻ z : Γ , C ⊢ A ↝ B`. Weakening on a zipper simply applies weakening to each term contained in the zipper.

```
wkᶻ : Γ ⊢ A ↝ B → Γ , C ⊢ A ↝ B
wkᶻ □         = □
wkᶻ (z · M)   = wkᶻ z · wk M
wkᶻ (force z) = force (wkᶻ z)
```

Recall that we introduced `M₀` in Section 4.

```
_ : M₀ ≡
  force
    ((ƛ ƛ (delay (` S S S S Z · ` Z · ` S Z)))
      · ` S Z
      · ` Z)
_ = refl
```

We can factor `M₀` into a zipper and a smaller term as follows.

```
z₀ : ∅ , ι ⇒ ι ⇒ ι , ι , ι
       ⊢ ι ⇒ ι ⇒ Delay ι ↝ ι
z₀ = force □ · ` Z · ` S Z

N₀ : ∅ , ι ⇒ ι ⇒ ι , ι , ι ⊢ ι ⇒ ι ⇒ Delay ι
N₀ = ƛ ƛ (delay (` S S S S Z · ` Z · ` S Z))

_ : zip z₀ N₀ ≡ M₀
_ = refl
```

Note that the constructors of the zipper `z₀` are in the reverse order to the constructors of the final term `M₀`. The `force` on the inside of the zipper corresponds to `force` on the outside of `M₀`, while the two applications in the zipper are reversed compared to their occurrence in `M₀`.

9 Force-Delay Relation with Two Zippers

Having defined zippers, we can now define the force-delay relation with two zippers. Here the work of the two counters is performed by a single zipper, which keeps track of both the applications and the forces encountered as we traverse the term. We need two zippers because there is one for each of the related terms.

We write `FD z z′ M M′` to indicate that that zipper `z` relates to zipper `z′` and term `M` relates to term `M′`. Intuitively, both terms have the same lambda abstractions and applications, but matching instances of force and delay in term `M` have been removed in term `M′`. Similarly, both zippers have the same applications, but instances of force in zipper `z` have been removed in zipper `z′`. The two terms have the same context but may have different types, because one may have a `force` or `delay` applied when the other does not. Similarly for the two zippers. The invariant we expect is that `zip z M` and `zip z′ M′` are equivalent, so plugging the terms into the zippers must result in terms of the same type.

The relation is defined by five inference rules.

```
data FD :
       (z : Γ ⊢ A ↝ B) (z′ : Γ ⊢ A′ ↝ B)
       (M : Γ ⊢ A) (M′ : Γ ⊢ A′)
     → Set where

  cc :
       (s : CC (FD □ □) M M′)
     → ---------------------
       FD □ □ M M′

  force :
       (s : FD (force z) z′ M M′)
     → -------------------------
       FD z z′ (force M) M′

  delay :
       (s : FD z z′ M M′)
     → -------------------------
       FD (force z) z′ (delay M) M′
```

```
app :
      (r : FD (z · M) (z′ · M′) L L′)
  →   ------------------------------
      FD z z′ (L · M) (L′ · M′)

abs :
      (s : FD □ □ M M′)
      (t : FD (wkᶻ z) (wkᶻ z′) N N′)
  →   ---------------------------------
      FD (z · M) (z′ · M′) (ƛ N) (ƛ N′)
```

Rule `cc` states that the relation is compatibly closed when both zippers are empty. Force and delay must match to enable the optimisation, and similarly for applications and abstractions; if the optimisation does not apply, then compatible closure is used instead. Terms are traversed from outside to inside. Rule `force` states that every time we encounter a force we add it to the zipper on the left, and rule `delay` states that every time we encounter a delay we remove a force from the zipper on the left. Instances of force and delay are cancelled by the optimisation, so appear in the left term but not the right in these two rules. Rule `app` states that every time we encounter an application we add it to both zippers, and rule `abs` states every time we encounter a lambda abstraction we remove an application from both zippers. Applications and lambda abstractions are carried over by the optimisation, so appear in both the left term and the right in these two rules. In the abstraction rule, note that functions are related by `FD z z′ L L′`, while arguments are related by `FD □ □ M M′`. Note that whereas in `FDnat` it is the application rule that has two premises (one for the argument), in `FD` it is the *abstraction* rule that has two premises, where the argument is taken from the zipper; this will prove crucial in the proof of correctness.

Using zippers makes it easy to write the rules `force`, `delay`, `app`, and `abs`, which move constructs from the term to the zipper and vice-versa. A zipper is essentially the same as an evaluation context, but evaluation contexts are often written outside-in, which would hinder manipulation; indeed, for this reason some authors make a point of writing evaluation contexts inside-out, closely resembling zippers (e.g., see Danvy and Nielsen 2004).

It is easy to see the correspondence between `FDnat m n M M′` and `FD z z′ M M′`. In particular, `m` counts the number of forces in zipper `z`, there will be no forces in zipper `z′`, and `n` counts the number of applications in both zippers `z` and `z′`. It follows that because `FDnat` adequately characterises the behaviour of the optimisation, so does `FD`. That is, if the the optimisation is given source term `M` and produces target term `M′`, then `FD □ □ M M′` holds. Further, as with `FDnat`, it proves easy to write a decision procedure to determine whether `FD` holds for given zippers and terms, so it is feasible to use it as a certification relation for verified compilation.

Here is an example.

```
FD₀ : FD □ □ M₀ M₀′
FD₀ =
  force
    (app (app
      (abs (cc (` S Z))
        (abs (cc (` S Z))
          (delay (cc
            (` S S S S Z · ` Z · ` S Z)))))))
```

Again, this can be computed for us automatically by Agda.

Unlike `FDnat`, it will be easy to show that `FD` is sound, once we define a suitable notion of what it means for two terms to have identical behaviour.

10 Equivalent Terms

We write `M ~ N` to indicate that `M` and `N` have the same behaviour. The relation is a congruence, meaning it is both an equivalence and compatibly closed.

```
data _~_ : Relation where

  cc :
       (s : CC _~_ M M′)
    →  -----------------
       M ~ M′

  refl~ :
       -----
       M ~ M

  sym~ :
       (r : M ~ N)
    →  -----------
       N ~ M

  trans~ :
       (s : L ~ M)
       (t : M ~ N)
    →  -----------
       L ~ N

  force-delay :
       -------------------
       force (delay M) ~ M

  force-LET :
       ---------------------------------
       force (LET M N) ~ LET M (force N)

  LET-app :
       -------------------------------
       (LET M N) · L ~ LET M (N · wk L)
```

Rule `cc` states that the relation is compatibly closed, while rules `refl~`, `sym~`, and `trans~` state that the relation is reflexive, symmetric, and transitive. Three rules state what we take to be obvious equivalences. Rule `force-delay` states that force of a delay cancels out, rule `force-LET` states that

we can push a force under a let, and rule `LET-app` states that we can push an application under a let. In that last rule, we need to weaken the argument when we push it under the let, to account for the addition let binding; this is why had to go to the effort of defining renaming and weakening.

Agda has a convention for writing chains of equivalences. Here are the necessary definitions, which resemble similar defintions for `_≡_` given in the standard library.

```
module ~-Reasoning where

  infix 1 begin_
  infixr 2 step-~-| step-~-⟩ step-~-⟨
  infix 3 _∎

  begin_ : L ~ M → L ~ M
  begin r = r

  step-~-| : ∀ L → L ~ M → L ~ M
  step-~-| L r = r

  step-~-⟩ : ∀ L → M ~ N → L ~ M → L ~ N
  step-~-⟩ L s r = trans~ r s

  step-~-⟨ : ∀ L → M ~ N → M ~ L → L ~ N
  step-~-⟨ L s r = trans~ (sym~ r) s

  syntax step-~-| L r   = L ~⟨⟩ r
  syntax step-~-⟩ L s r = L ~⟨ r ⟩ s
  syntax step-~-⟨ L s r = L ~⟨ r ⟨ s

  _∎ : ∀ (M : Γ ⊢ A) → M ~ M
  M ∎ = refl~

open ~-Reasoning
```

Here is an example. We can assert equivalence between terms that are definitionally equal (using `~⟨⟩`), and apply equivalences forward (using `~⟨_⟩`), and backward (using `~⟨_⟨`). In Agda, variable names can contain almost any symbol. By convention, we take `L~M` as the name of a judgement `L ~ M`.

```
postulate
  L~M : L ~ M
  N~M : N ~ M

L~N : ∀ {M} → L ~ N
L~N {L = L} {N} {M} =
  begin
    zip □ L
  ~⟨⟩
    L
  ~⟨ L~M ⟩
    M
  ~⟨ N~M ⟨
    N
  ∎

_ : L~N {L = L} {N = N} {M = M} ≡
  trans~ L~M (trans~ (sym~ N~M) refl~)
_ = refl
```

If we ask Agda to prove `M₀ ~ M₀′` using automatic search it fails, even if given 100 seconds. The search space is too large, because there are too many ways that `tran~` can be applied.

11 Soundness

We now show that the `FD` relation is sound, that is, if `FD □ □ M M′` holds then `M ~ M′` holds. Our proof requires a more general result, namely that if `FD z z′ M M′` holds then `zip z M ~ zip z′ M′` holds.

It is easy to push compatibility closure under a zip.

```
cc-zip :
    (z : Γ ⊢ A ⇝ B)
    (s : CC R M M′)
  → ------------------------
    CC R (zip z M) (zip z M′)
cc-zip □ t          = t
cc-zip (z · M) t    = cc-zip z (t · reflCC M)
cc-zip (force z) t = cc-zip z (force t)
```

Hence it is also easy to push equivalence under a zip.

```
zip~ :
     (z : Γ ⊢ A ⇝ B)
     (s : M ~ M′)
  →  ------------------
     zip z M ~ zip z M′
zip~ z r = cc (cc-zip z (base r))
```

It is also easy to see that equivalence is preserved by let.

```
LET~ :
     (s : M ~ M′)
     (t : N ~ N′)
  →  --------------------
     LET M N ~ LET M′ N′
LET~ M~M′ N~N′
  =  cc ((ƛ (base N~N′)) · base M~M′)
```

Further, we can push a zip under a LET. This is easily shown by induction over the structure of the zipper. There are three cases, one for each of the constructors of the zipper.

```
zip-LET :
    (z : Γ ⊢ B ⇝ C)
    (M : Γ ⊢ A)
    (N : Γ , A ⊢ B)
  → ---------------------------------------
    zip z (LET M N) ~ LET M (zip (wkᶻ z) N)
zip-LET □ M N =
  begin
    zip □ (LET M N)
  ~⟨⟩
    LET M (zip □ N)
  ∎
```

```
zip-LET (z · L) M N =
  begin
    zip (z · L) (LET M N)
  ~⟨⟩
    zip z (LET M N · L)
  ~⟨ zip~ z LET-app ⟩
    zip z (LET M (N · wk L))
  ~⟨ zip-LET z M (N · wk L) ⟩
    LET M (zip (wkᶻ z) (N · wk L))
  ~⟨⟩
    LET M (zip (wkᶻ (z · L)) N)
  ∎
zip-LET (force z) M N =
  begin
    zip (force z) (LET M N)
  ~⟨⟩
    zip z (force (LET M N))
  ~⟨ zip~ z force-LET ⟩
    zip z (LET M (force N))
  ~⟨ zip-LET z M (force N) ⟩
    LET M (zip (wkᶻ z) (force N))
  ~⟨⟩
    LET M (zip (force (wkᶻ z)) N)
  ∎
```

With these preliminaries out of the way, we are ready to prove soundness. The proof consists of two propositions, one for the relation `CC (FD □ □)` and one for `FD`. There is a mutual dependence between these, since soundness of the first depends on soundness of the second, and vice versa. Therefore, we begin by assuming soundness of `FD □ □`, which we will later show follows as a special case of the soundness of `FD`.

```
sound□□ :
    (M M′ : Γ ⊢ A)
    (s : FD □ □ M M′)
  → ----------------
    M ~ M′
```

It is easy to show that the compatible closure is sound given the above assumption.

```
soundCC : CC (FD □ □) N N′ → CC _~_ N N′
soundCC (base s)  = base (sound□□ _ _ s)
soundCC (` x)     = ` x
soundCC (ƛ t)     = ƛ (soundCC t)
soundCC (r · s)   = soundCC r · soundCC s
soundCC (delay s) = delay (soundCC s)
soundCC (force s) = force (soundCC s)
```

We now have everything we need to prove soundness. The proof is by induction over the evidence that the relation `FD z z′ M M′` holds. There are five cases, one for each constructor in the relation `FD`.

```
sound :
    (z : Γ ⊢ A ↠ B) (z′ : Γ ⊢ A′ ↠ B)
    (M : Γ ⊢ A) (M′ : Γ ⊢ A′)
    (s : FD z z′ M M′)
  → ---------------------------------
    zip z M ~ zip z′ M′
sound .□ .□ M M′ (cc p) =
  cc (soundCC p)
sound z z′ (force M) M′ (force p) =
  begin
    zip z (force M)
  ~⟨⟩
    zip (force z) M
  ~⟨ sound (force z) z′ M M′ p ⟩
    zip z′ M′
  ∎
sound (force z) z′ (delay M) M′ (delay p) =
  begin
    zip (force z) (delay M)
  ~⟨⟩
    zip z (force (delay M))
  ~⟨ zip~ z force-delay ⟩
    zip z M
  ~⟨ sound z z′ M M′ p ⟩
    zip z′ M′
  ∎
sound z z′ (L · M) (L′ · M′) (app q) =
  begin
    zip z (L · M)
  ~⟨⟩
    zip (z · M) L
  ~⟨ sound (z · M) (z′ · M′) L L′ q ⟩
    zip (z′ · M′) L′
  ~⟨⟩
    zip z′ (L′ · M′)
  ∎
sound (z · M) (z′ · M′) (ƛ N) (ƛ N′) (abs p q) =
  begin
    zip (z · M) (ƛ N)
  ~⟨⟩
    zip z (LET M N)
  ~⟨ zip-LET z M N ⟩
    LET M (zip (wkᶻ z) N)
  ~⟨ LET~ (sound □ □ M M′ p)
          (sound (wkᶻ z) (wkᶻ z′) N N′ q) ⟩
    LET M′ (zip (wkᶻ z′) N′)
  ~⟨ zip-LET z′ M′ N′ ⟨
    zip z′ (LET M′ N′)
  ~⟨⟩
    zip (z′ · M′) (ƛ N′)
  ∎
```

The two dots in the line `sound .□ .□ M M′ (cc s)` tell Agda that other arguments (namely, the last) force the first two arguments to be `□`, which allows Agda to work out that it should match the last argument first. Otherwise, Agda

attempts to match a different argument first, causing two of the equations to be displayed in grey (meaning that they do not hold definitionally).

Finally, we show that the special case of soundness we assumed earlier follows from the general case.

```
sound□□ M M′ s = sound □ □ M M′ s
```

This completes the proof.

Here is an example.

```
M₀~M₀′ : M₀ ~ M₀′
M₀~M₀′ = sound□□ M₀ M₀′ FD₀

_ : M₀~M₀′ ≡
  trans~
  (trans~
    (trans~
      (trans~
        (trans~ (cc (force (base LET-app)))
          (trans~
            (trans~
              (cc (base force-LET))
              (trans~ refl~ refl~))
            refl~))
        (trans~
          (cc ((ƛ base
            (trans~
              (trans~
                (cc (base force-LET))
                (trans~ refl~ refl~))
              (trans~
                (cc ((ƛ base
                  (trans~
                    (cc (base force-delay))
                    (trans~
                      (cc
                        (` S (S (S (S Z)))
                          · ` Z
                          · ` S Z))
                        refl~)))
                      · base (cc (` S Z))))
                (trans~ (sym~ refl~) refl~))))
            · base (cc (` S Z))))
      (trans~
        (sym~
          (trans~
            (cc (base LET-app))
            (trans~ refl~ refl~)))
          refl~))
        refl~)
    refl~)
  refl~
_ = refl
```

The evidence that `M₀ ~ M₀′` is unwieldy. Fortunately, the code above was not written by hand; instead, it was generated by asking Agda to normalise `sound□□ M₀ M₀′ FD₀`. In general, once our decision procedure yields evidence that `FD □ □ M M′`, the soundness proof ensures that `M ~ M′`. We've proved this once and for all, so the fact that evidence for `M ~ M′` is unwieldy hardly matters.

12 Denotational Semantics

As a further step, we could specify a denotational semantics for terms, and show that two terms satisfying `_~_` have identical semantics. It turns out that this is entirely straightforward, so we omit it. Considering the semantics would, in the words of Reynolds, add little to the exposition save length.

13 Conclusion

The public release of IOG's UPLC optimiser includes certification for the force-delay optimisation pass using the zipper-based `FD` relation above, with extensions to handle the full UPLC term syntax. It supports most UPLC optimisation passes: for each included pass, we define a suitable certification relation and a decision procedure that can check whether the source and target terms of the pass satisfy the relation. We are currently completing proofs that the certification relations preserve soundness, as well as defining certification relations and decision procedures for the remaining passes. The proof of soundness for the force-delay optimisation pass depends crucially on our use of zippers as described above, and we expect zippers may also prove useful in establishing certification relations for the remaining passes.

14 Data Availability Statement

The executable Agda script of this paper is available as an artifact in the ACM Digital Library (Wadler *et al* 2025), in the file `Zippers.lagda.md`.

References

Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Computer Science Logic (Lecture Notes in Computer Science)*, Jörg Flum and Mario Rodriguez-Artalejo (Eds.). Springer, Berlin, Heidelberg, 453–468. doi:10.1007/3-540-48168-0_32

James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. 2019. System F in Agda, for Fun and Profit. In *Mathematics of Program Construction*, Graham Hutton (Ed.). Vol. 11825. Springer International Publishing, Cham, 255–297. doi:10.1007/978-3-030-33636-3_10 Series Title: Lecture Notes in Computer Science.

Olivier Danvy. 2023. A Deforestation of Reducts: Refocusing. doi:10.48550/arXiv.2302.10455 arXiv:2302.10455 [cs].

Olivier Danvy and Lasse R. Nielsen. 2004. Refocusing in Reduction Semantics. *BRICS Report Series* 11, 26 (Nov. 2004). doi:10.7146/brics.v11i26.21851

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph. D. Dissertation. Université Paris VII.

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

Wen Kokke, Jeremy G. Siek, and Philip Wadler. 2020. Programming language foundations in Agda. *Science of Computer Programming* 194 (Aug. 2020), 102440. doi:10.1016/j.scico.2020.102440

John C. Reynolds. 2000. The Meaning of Types From Intrinsic to Extrinsic Semantics. *BRICS Report Series* 32 (June 2000). doi:10.7146/brics.v7i32.20167 Number: 32.

Philip Wadler. 2024. Explicit Weakening. In *A Second Soul: Celebrating the Many Languages of Programming*, Annette Bieniusa, Markus Degen, and Stefan Wehr (Eds.). Electronic Proceedings in Theoretical Computer Science, Vol. 413. Open Publishing Association, 15–26. doi:10.4204/EPTCS.413.2

Philip Wadler, Wen Kokke, and Jeremy G Siek. 2018. *Programming Language Foundations in Agda.* https://plfa.inf.ed.ac.uk/

Philip Wadler, Ramsay Taylor, and Jacco O. G. Krijnen. 2025. Executable Agda Script for 'A Tale of Two Zippers'. ACM. doi:10.1145/3747406

Received 2025-06-09; accepted 2025-07-31