CHALMERS | GÖTEBORG UNIVERSITY

Technical Report No. 07-16

A Formalisation of the Correctness Result From

"Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures"

NILS ANDERS DANIELSSON

Department of Computer Science and Engineering
Division of Computing Science
CHALMERS UNIVERSITY OF TECHNOLOGY/
GÖTEBORG UNIVERSITY
Göteborg, Sweden, 2007

Abstract

This document describes how the correctness result in "Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures" has been formalised.

Contents

1 Introduction

This document describes how the correctness result in "Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures" (Danielsson 2008) has been formalised, using the Agda 2 logic (Norell 2007; The Agda Team 2007). This is done by listing the actual (commented) Agda source code used to state this result.

The emphasis is on definitions. Few actual proofs are included, partly since they are relatively uninteresting (simple proofs by induction), but primarily because their correctness can be checked—with a certain level of trust—by running a type checker. The full source code, with all proofs, can (currently) be obtained from `http://www.cs.chalmers.se/~nad/`. This code is accepted by the Agda type checker (which includes a termination checker). Unfortunately Agda currently does not have a completeness checker, so pattern completeness, i.e. exhaustiveness of case analyses, has been checked manually.

This document should not be read without first consulting Danielsson (2008), which motivates and explains many of the definitions here. Some familiarity with Agda or other dependently typed languages may also be necessary in order to understand the definitions.

The following sections contain the Agda source code used for the formalisation. The subsection titles correspond to module names, so the table of contents can be used to locate a particular module.

2 Supporting definitions

This section contains definitions from a supporting library. Some of these definitions have been edited slightly for this document, and only excerpts of the relevant modules have been included. This only applies to this section, though. In the following sections the unedited code of the formalisation is included verbatim.

2.1 Logic

```
infix 4 _≡_ _≅_

-- Propositional, homogeneous equality.

data _≡_ {a : Set} (x : a) : a -> Set where
  ≡-refl : x ≡ x

-- Heterogeneous equality.

data _≅_ {a : Set} (x : a) : {b : Set} -> b -> Set where
  ≅-refl : x ≅ x
```

2.2 Relation.Binary.PropositionalEquality

```
-- The propositional equality is symmetric.

≡-sym : forall {a : Set} {x y : a} -> x ≡ y -> y ≡ x
≡-sym ≡-refl = ≡-refl

-- And transitive.

≡-trans : forall {a : Set} {x y z : a} -> x ≡ y -> y ≡ z -> x ≡ z
≡-trans ≡-refl ≡-refl = ≡-refl

-- And substitutive.

≡-subst :  forall {a : Set} (P : a -> Set) {x y}
        -> x ≡ y -> P x -> P y
≡-subst P ≡-refl p = p

-- And all functions are congruences with respect to this equality.

≡-cong :  forall {a b : Set} (f : a -> b) {x y}
       -> x ≡ y -> f x ≡ f y
≡-cong f ≡-refl = ≡-refl

≡-cong₂ :  forall {a b c : Set} (_•_ : a -> b -> c) {x₁ y₁ x₂ y₂}
        -> x₁ ≡ y₁ -> x₂ ≡ y₂ -> (x₁ • x₂) ≡ (y₁ • y₂)
≡-cong₂ _•_ ≡-refl ≡-refl = ≡-refl
```

2.3 Data.Function

```
infixr 9 _∘_

-- Identity function.

id : {a : Set} -> a -> a
id x = x

-- Composition of functions.

_∘_ : {a b c : Set} -> (b -> c) -> (a -> b) -> (a -> c)
f ∘ g = \x -> f (g x)
```

2.4 Data.Unit

```
-- The unit type. Note that Agda uses η equality for records, so the
-- unique value of the unit type (an empty record) can always be
-- inferred automatically.

record ⊤ : Set where

-- The unit element.

tt : ⊤
tt = record {}
```

2.5 Data.Nat

```
infixl 7 _⊓_
infixl 6 _+_ _∸_
infix  4 _≤_

-- Natural numbers.

data ℕ : Set where
  zero : ℕ
  suc  : ℕ -> ℕ

-- The standard ordering of natural numbers.

data _≤_ : ℕ -> ℕ -> Set where
  z≤n : forall {n}            -> zero  ≤ n
  s≤s : forall {m n} -> m ≤ n -> suc m ≤ suc n

-- Addition.

_+_ : ℕ -> ℕ -> ℕ
zero  + n = n
suc m + n = suc (m + n)
```

```
-- Subtraction (with m ∸ n = 0 if m ≤ n).

_∸_ : ℕ -> ℕ -> ℕ
m     ∸ zero  = m
zero  ∸ suc n = zero
suc m ∸ suc n = m ∸ n

-- Minimum.

_⊓_ : ℕ -> ℕ -> ℕ
zero  ⊓ n     = zero
suc m ⊓ zero  = zero
suc m ⊓ suc n = suc (m ⊓ n)
```

2.6 Data.Nat.Properties

```
-- Some properties of natural numbers.

abstract
  m⊓n+n∸m≡n : forall m n -> (m ⊓ n) + (n ∸ m) ≡ n
  m⊓n+n∸m≡n = ...

  i∸k∸j+j∸k≡i+j∸k
    : forall i j k -> i ∸ (k ∸ j) + (j ∸ k) ≡ i + j ∸ k
  i∸k∸j+j∸k≡i+j∸k = ...
```

2.7 Data.List

```
infixr 5 _::_

-- Lists.

data [_] (a : Set) : Set where
  []  : [ a ]
  _::_ : a -> [ a ] -> [ a ]

-- The standard fold for lists.

foldr : {a b : Set} -> (a -> b -> b) -> b -> [ a ] -> b
foldr c n []       = n
foldr c n (x :: xs) = c x (foldr c n xs)

-- The sum of all the list elements.

sum : [ ℕ ] -> ℕ
sum = foldr _+_ zero
```

3 Basics

This section defines the basic elements of the formalisation: the type system, terms and values (both simple and thunked), and operations such as erasure and substitution.

3.1 Kind

```
------------------------------------------------------------------------
-- Kinds (simple or thunked)
------------------------------------------------------------------------

module Kind where

open import Data.Unit using (⊤; tt)
open import Data.Nat using (ℕ; suc)

-- The type of calculus: thunked or unthunked (simple).

data Kind : Set where
  simple  : Kind
  thunked : Kind

-- In the thunked type system natural numbers are used to keep track
-- of evaluation steps, but not in the simple system. By using Maybe ℕ
-- the two type systems can be merged in a nice way (see
-- TypeSystem.Thunk? and TypeSystem._⊢_). The η equality of the unit
-- type (⊤) is convenient here, since it ensures that the single unit
-- value does not need to be written out when it is hidden, even if
-- nothing depends on it.

Maybe : Set -> Kind -> Set
Maybe a simple  = ⊤
Maybe a thunked = a

-- A lifted version of suc.

suc' : forall k -> Maybe ℕ k -> Maybe ℕ k
suc' simple  _ = tt
suc' thunked n = suc n
```

3.2 TypeSystem

```
------------------------------------------------------------------------
-- The type system of a simply typed λ-calculus with products and
-- natural numbers, and perhaps also thunks
------------------------------------------------------------------------

module TypeSystem where

open import Kind

open import Data.Nat using (ℕ; zero; suc; _+_; _∸_)

infixl 70 _·_
infixr 60 _,_
infix  50 λ_ λ²_
infixl 46 _>>=_
infix  43 ✓_
infix  41 _[_]
infixr 40 _×_ _●×_ _×●_
infixr 30 _→_
infixl 20 _▷_
infix  10 _∋_ _⊢_

------------------------------------------------------------------------
-- Types

-- Types, indexed on their kind (simple or thunked).

data Ty : Kind -> Set where
  Nat   : forall {k} -> Ty k
  _×_   : forall {k} -> Ty k -> Ty k -> Ty k
  _→_   : forall {k} -> Ty k -> Ty k -> Ty k
  Thunk : ℕ -> Ty thunked -> Ty thunked

-- Thunk? n τ is Thunk n τ in the thunked case, but simplifies to τ in
-- the simple case. (Note that Maybe ℕ k simplifies to ⊤ when k =
-- simple, so the superfluous argument n can be inferred automatically
-- in this case.) In general, functions with names ending in a
-- question mark "vanish" in the simple case.

Thunk? : forall {k} -> Maybe ℕ k -> Ty k -> Ty k
Thunk? {k = simple}  n τ = τ
Thunk? {k = thunked} n τ = Thunk n τ

------------------------------------------------------------------------
-- Contexts

-- Contexts used by generalised pay. (Here context is used in the
-- sense of something with holes, not in the sense of a list of
```

```
-- types.)

data PayCtxt : Set where
  ●      : PayCtxt
  const● : Ty thunked -> PayCtxt
  _●×_   : PayCtxt -> Ty thunked -> PayCtxt
  _×●_   : Ty thunked -> PayCtxt -> PayCtxt
  Thunk● : ℕ -> PayCtxt -> PayCtxt

-- Fills the hole in a PayCtxt.

_[_] : PayCtxt -> Ty thunked -> Ty thunked
●          [ τ ] = τ
const● σ   [ τ ] = σ
(C ●× σ)   [ τ ] = C [ τ ] × σ
(σ ×● C)   [ τ ] = σ × C [ τ ]
Thunk● n C [ τ ] = Thunk n (C [ τ ])

------------------------------------------------------------------------
-- Variables

-- Contexts used to keep track of variables.

data Ctxt (k : Kind) : Set where
  -- Empty context.
  ε   : Ctxt k
  -- Context extension.
  _▷_ : Ctxt k -> Ty k -> Ctxt k

-- Type signature for term-like things.

TmLike : Set1
TmLike = forall {k} -> Ctxt k -> Ty k -> Set

-- Variables (de Bruijn indices).

data _∋_ : TmLike where
  -- Variable zero.
  vz : forall {k} {Γ : Ctxt k} {σ}   -> Γ ▷ σ ∋ σ
  -- Variable successor.
  vs : forall {k} {Γ : Ctxt k} {σ τ} -> Γ ∋ τ -> Γ ▷ σ ∋ τ
  -- Pretend that a computation takes more steps than it actually
  -- does. (Note that this constructor only exists in the thunked
  -- setting.)
  gwait∋ :  forall {Γ σ n₁} C n₂
         -> Γ ∋ C [ Thunk n₁ σ ]
         -> Γ ∋ C [ Thunk (n₂ + n₁) σ ]

-- The gwait∋ constructor ("generalised waiting") is included to aid
-- the analysis of code involving generalised pay (gpay). There is
```

```
-- also a corresponding term constructor gwait. These constructors do
-- not correspond to any library primitives. In light of the
-- formalisation it would be sound to include gwait in the library,
-- though.

------------------------------------------------------------------------
-- Terms

-- The term type encodes both simple terms and thunked, run-time
-- terms, depending on the value of the Kind index. The necessary uses
-- of ticks in the run-time terms have been encoded using types. As an
-- example, consider λ_:
--
-- * In the simple case the type of λ_ simplifies to the following
--   type, equivalent to the one used in the paper:
--
--     λ_ :  forall {Γ : Ctxt simple} {n : ⊤} {σ τ}
--        -> Γ ▷ σ ⊢ τ -> Γ ⊢ σ → τ
--
--   (⊤ is the unit type, so the n could be removed. ⊤ also has η
--   equality, so the unique value of n never needs to be written
--   explicitly; it can always be inferred.)
--
-- * In the thunked case the type of λ_ instead simplifies to:
--
--     λ_ :  forall {Γ : Ctxt thunked} {n : ℕ} {σ τ}
--        -> Γ ▷ σ ⊢ Thunk n τ -> Γ ⊢ σ → Thunk (suc n) τ
--
--   The run-time form of lambda is λx. ✓ e. Here e has to have type
--   Thunk n τ for some n and τ; this is reflected in the type of λ_.
--   The type of the entire expression, with the extra tick, then
--   becomes σ → Thunk (suc n) τ (where σ is the type of x); this is
--   also reflected in the type of λ_.

-- Some extensions, mentioned in the paper, are included in the term
-- type. To show that partial applications are OK the constructor s
-- can be partially applied, and I have also included a two-argument
-- lambda. Furthermore general fixpoints are allowed (fix) and the
-- generalised pay (gpay) is included.

data _⊢_ : TmLike where
  -- Lambda calculus.
  var  :  forall {k} {Γ : Ctxt k} {τ}
       -> Γ ∋ τ -> Γ ⊢ τ
  λ_   :  forall {k} {Γ : Ctxt k} {n σ τ}
       -> Γ ▷ σ ⊢ Thunk? n τ -> Γ ⊢ σ → Thunk? (suc' k n) τ
  λ²_  :  forall {k} {Γ : Ctxt k} {n σ₁ σ₂ τ}
       -> Γ ▷ σ₁ ▷ σ₂ ⊢ Thunk? n τ
       -> Γ ⊢ σ₁ → σ₂ → Thunk? (suc' k n) τ
```

```
  _·_  :  forall {k} {Γ : Ctxt k} {σ τ}
       -> Γ ⊢ σ → τ -> Γ ⊢ σ -> Γ ⊢ τ
  fix  :  forall {k} {Γ : Ctxt k} {n σ}
       -> Γ ▷ Thunk? (suc' k n) σ ⊢ Thunk? n σ
       -> Γ ⊢ Thunk? (suc' k n) σ

  -- Natural numbers.
  z      :  forall {k} {Γ : Ctxt k}
         -> Γ ⊢ Nat
  s      :  forall {k} {Γ : Ctxt k}
         -> Γ ⊢ Nat → Nat
  natrec :  forall {k} {Γ : Ctxt k} {n σ}
         -> Γ ⊢ Thunk? n σ
         -> Γ ▷ Nat ▷ Thunk? (suc' k n) σ ⊢ Thunk? n σ
         -> Γ ⊢ Nat → Thunk? (suc' k n) σ

  -- Products.
  _,_     :  forall {k} {Γ : Ctxt k} {σ τ}
          -> Γ ⊢ σ -> Γ ⊢ τ -> Γ ⊢ σ × τ
  uncurry :  forall {k} {Γ : Ctxt k} {n σ₁ σ₂ τ}
          -> Γ ▷ σ₁ ▷ σ₂ ⊢ Thunk? n τ
          -> Γ ⊢ σ₁ × σ₂ → Thunk? (suc' k n) τ

  -- The following constructors only exist in the thunked case (by
  -- virtue of the types assigned to them). Note that force is not
  -- included, since only run-time terms are considered.

  -- Thunk library primitives.
  return :  forall {Γ σ} -> Γ ⊢ σ -> Γ ⊢ Thunk zero σ
  _>>=_  :  forall {Γ σ τ n₁ n₂}
         -> Γ ⊢ Thunk n₁ σ
         -> Γ ⊢ σ → Thunk n₂ τ
         -> Γ ⊢ Thunk (n₁ + n₂) τ
  ✓_     :  forall {Γ σ n}
         -> Γ ⊢ Thunk n σ -> Γ ⊢ Thunk (suc n) σ
  pay    :  forall {Γ σ n₁} n₂
         -> Γ ⊢ Thunk n₁ σ
         -> Γ ⊢ Thunk n₂ (Thunk (n₁ ∸ n₂) σ)
  gpay   :  forall {Γ σ n₁} C n₂
         -> Γ ⊢ C [ Thunk n₁ σ ]
         -> Γ ⊢ Thunk n₂ (C [ Thunk (n₁ ∸ n₂) σ ])
  gwait  :  forall {Γ σ n₁} C n₂
         -> Γ ⊢ C [ Thunk n₁ σ ]
         -> Γ ⊢ C [ Thunk (n₂ + n₁) σ ]

-- The term type includes both a simple pay and a generalised one
-- (gpay). The function pay is a special case of gpay, so it could be
-- omitted, but this arrangement makes it relatively easy to see (some
-- of) the additional complexity incurred by the addition of gpay.
```

3.3 Erasure

```
--------------------------------------------------------------------------------
-- Erasing thunks
--------------------------------------------------------------------------------

-- Functions converting thunked things to the corresponding simple
-- things, with all thunks and library primitives erased.

module Erasure where

open import Kind
open import TypeSystem

open import Logic using (_≡_; ≡-refl)
open import Relation.Binary.PropositionalEquality
  using (≡-subst; ≡-cong)

-- Erases types.

⌜_⌝⋆ : Ty thunked -> Ty simple
⌜ Nat ⌝⋆       = Nat
⌜ σ × τ ⌝⋆     = ⌜ σ ⌝⋆ × ⌜ τ ⌝⋆
⌜ σ → τ ⌝⋆     = ⌜ σ ⌝⋆ → ⌜ τ ⌝⋆
⌜ Thunk n τ ⌝⋆ = ⌜ τ ⌝⋆

-- Thunks inside a pay context are erased, so we get this lemma:

erase-lemma :  forall {m n τ} C
            -> ⌜ C [ Thunk m τ ] ⌝⋆ ≡ ⌜ C [ Thunk n τ ] ⌝⋆
erase-lemma ●            = ≡-refl
erase-lemma (const● σ)   = ≡-refl
erase-lemma (C ●× σ)     = ≡-cong (\τ -> τ × ⌜ σ ⌝⋆) (erase-lemma C)
erase-lemma (σ ×● C)     = ≡-cong (\τ -> ⌜ σ ⌝⋆ × τ) (erase-lemma C)
erase-lemma (Thunk● n C) = erase-lemma C

-- Erases contexts.

⌜_⌝▷ : Ctxt thunked -> Ctxt simple
⌜ ε ⌝▷     = ε
⌜ Γ ▷ σ ⌝▷ = ⌜ Γ ⌝▷ ▷ ⌜ σ ⌝⋆

-- Type signature for erasers of term-like things.

TmEraser : TmLike -> Set
TmEraser _●_ = forall {Γ τ} -> Γ ● τ -> ⌜ Γ ⌝▷ ● ⌜ τ ⌝⋆

-- Erases variables.
```

```
⌜_⌝∋ : TmEraser _∋_
⌜ vz ⌝∋                    = vz
⌜ vs x ⌝∋                  = vs ⌜ x ⌝∋
⌜ gwait∋ {Γ = Γ} C m x ⌝∋ =
  ≡-subst (\τ -> ⌜ Γ ⌝▷ ∋ τ) (erase-lemma C) ⌜ x ⌝∋

-- Erases terms.

⌜_⌝ : TmEraser _⊢_
⌜ var x ⌝                = var ⌜ x ⌝∋
⌜ λ t ⌝                  = λ ⌜ t ⌝
⌜ λ² t ⌝                 = λ² ⌜ t ⌝
⌜ t₁ · t₂ ⌝              = ⌜ t₁ ⌝ · ⌜ t₂ ⌝
⌜ fix t ⌝                = fix ⌜ t ⌝
⌜ z ⌝                    = z
⌜ s ⌝                    = s
⌜ natrec t₁ t₂ ⌝         = natrec ⌜ t₁ ⌝ ⌜ t₂ ⌝
⌜ (t₁ , t₂) ⌝            = (⌜ t₁ ⌝ , ⌜ t₂ ⌝)
⌜ uncurry t ⌝            = uncurry ⌜ t ⌝
⌜ return t ⌝             = ⌜ t ⌝
⌜ t₁ >>= t₂ ⌝            = ⌜ t₂ ⌝ · ⌜ t₁ ⌝
⌜ ✓ t ⌝                  = ⌜ t ⌝
⌜ pay n₂ t ⌝             = ⌜ t ⌝
⌜ gpay {Γ = Γ} C n₂ t ⌝ =
  ≡-subst (\τ -> ⌜ Γ ⌝▷ ⊢ τ) (erase-lemma C) ⌜ t ⌝
⌜ gwait {Γ = Γ} C n₂ t ⌝ =
  ≡-subst (\τ -> ⌜ Γ ⌝▷ ⊢ τ) (erase-lemma C) ⌜ t ⌝
```

3.4 Thunked

```
------------------------------------------------------------------------
-- General (not necessarily run-time) thunked terms
------------------------------------------------------------------------

-- TypeSystem defines the simple terms and the thunked, run-time
-- terms. The general thunked terms are not needed for the
-- formalisation, but included in the paper to make it easier to
-- follow. This module defines the general thunked terms, to show that
-- the definition in the paper is well-formed. It is also shown how
-- run-time terms can be translated into general terms (by making the
-- implicit ticks explicit), and how general terms can be erased.

module Thunked where

open import Kind
open import TypeSystem
open import Erasure

open import Data.Nat using (zero; suc; _+_; _∸_)
open import Logic using (_≡_; ≡-refl)
open import Relation.Binary.PropositionalEquality
  using (≡-subst; ≡-cong; ≡-cong₂)

infixl 70 _·⁺_
infixr 60 _,⁺_
infix  50 λ⁺_ λ²⁺_
infixl 46 _>>=⁺_
infix  43 ✓⁺_
infix  10 _⊢⁺_

-- General thunked terms. (Note that force is included.)

data _⊢⁺_ (Γ : Ctxt thunked) : Ty thunked -> Set where
  -- Lambda calculus.
  var⁺ : forall {τ}       -> Γ ∋ τ -> Γ ⊢⁺ τ
  λ⁺_  : forall {σ τ}     -> Γ ▷ σ ⊢⁺ τ -> Γ ⊢⁺ σ → τ
  λ²⁺_ : forall {σ₁ σ₂ τ} -> Γ ▷ σ₁ ▷ σ₂ ⊢⁺ τ -> Γ ⊢⁺ σ₁ → σ₂ → τ
  _·⁺_ : forall {σ τ}     -> Γ ⊢⁺ σ → τ -> Γ ⊢⁺ σ -> Γ ⊢⁺ τ
  fix⁺ : forall {σ}       -> Γ ▷ σ ⊢⁺ σ -> Γ ⊢⁺ σ

  -- Natural numbers.
  z⁺      :  Γ ⊢⁺ Nat
  s⁺      :  Γ ⊢⁺ Nat → Nat
  natrec⁺ :  forall {σ}
          -> Γ ⊢⁺ σ -> Γ ▷ Nat ▷ σ ⊢⁺ σ -> Γ ⊢⁺ Nat → σ

  -- Products.
  _,⁺_     :  forall {σ τ}     -> Γ ⊢⁺ σ -> Γ ⊢⁺ τ -> Γ ⊢⁺ σ × τ
```

```
  uncurry⁺ :  forall {σ₁ σ₂ τ}
           -> Γ ▷ σ₁ ▷ σ₂ ⊢⁺ τ -> Γ ⊢⁺ σ₁ × σ₂ → τ

  -- Thunk library primitives.
  force⁺  :  forall {n σ} -> Γ ⊢⁺ Thunk n σ -> Γ ⊢⁺ σ
  return⁺ :  forall {σ}   -> Γ ⊢⁺ σ -> Γ ⊢⁺ Thunk zero σ
  _>>=⁺_  :  forall {σ τ n₁ n₂}
          -> Γ ⊢⁺ Thunk n₁ σ -> Γ ⊢⁺ σ → Thunk n₂ τ
          -> Γ ⊢⁺ Thunk (n₁ + n₂) τ
  ✓⁺_     :  forall {σ n}
          -> Γ ⊢⁺ Thunk n σ -> Γ ⊢⁺ Thunk (suc n) σ
  pay⁺    :  forall {σ n₁} n₂
          -> Γ ⊢⁺ Thunk n₁ σ
          -> Γ ⊢⁺ Thunk n₂ (Thunk (n₁ ∸ n₂) σ)
  gpay⁺   :  forall {σ n₁} C n₂
          -> Γ ⊢⁺ C [ Thunk n₁ σ ]
          -> Γ ⊢⁺ Thunk n₂ (C [ Thunk (n₁ ∸ n₂) σ ])
  gwait⁺  :  forall {σ n₁} C n₂
          -> Γ ⊢⁺ C [ Thunk n₁ σ ]
          -> Γ ⊢⁺ C [ Thunk (n₂ + n₁) σ ]

-- This definition exposes the implicit ticks present in the
-- representation of run-time terms.

tmToTm⁺ : forall {Γ τ} -> Γ ⊢ τ -> Γ ⊢⁺ τ
tmToTm⁺ (var x)        = var⁺ x
tmToTm⁺ (λ e)          = λ⁺ (✓⁺ tmToTm⁺ e)
tmToTm⁺ (λ² e)         = λ²⁺ (✓⁺ tmToTm⁺ e)
tmToTm⁺ (e₁ · e₂)      = tmToTm⁺ e₁ ·⁺ tmToTm⁺ e₂
tmToTm⁺ (fix e)        = fix⁺ (✓⁺ tmToTm⁺ e)
tmToTm⁺ z              = z⁺
tmToTm⁺ s              = s⁺
tmToTm⁺ (natrec e₁ e₂) = natrec⁺ (✓⁺ tmToTm⁺ e₁) (✓⁺ tmToTm⁺ e₂)
tmToTm⁺ (e₁ , e₂)      = (tmToTm⁺ e₁ ,⁺ tmToTm⁺ e₂)
tmToTm⁺ (uncurry e)    = uncurry⁺ (✓⁺ tmToTm⁺ e)
tmToTm⁺ (return e)     = return⁺ (tmToTm⁺ e)
tmToTm⁺ (e₁ >>= e₂)    = tmToTm⁺ e₁ >>=⁺ tmToTm⁺ e₂
tmToTm⁺ (✓ e)          = ✓⁺ tmToTm⁺ e
tmToTm⁺ (pay n₂ e)     = pay⁺ n₂ (tmToTm⁺ e)
tmToTm⁺ (gpay C n₂ e)  = gpay⁺ C n₂ (tmToTm⁺ e)
tmToTm⁺ (gwait C n₂ e) = gwait⁺ C n₂ (tmToTm⁺ e)

-- This definition shows that the definition of erasure used in the
-- paper is well-typed.

⌜_⌝⁺ : forall {Γ τ} -> Γ ⊢⁺ τ -> ⌜ Γ ⌝▷ ⊢ ⌜ τ ⌝★
⌜ var⁺ x ⌝⁺                  = var ⌜ x ⌝∋
⌜ λ⁺ t ⌝⁺                    = λ ⌜ t ⌝⁺
⌜ λ²⁺ t ⌝⁺                   = λ² ⌜ t ⌝⁺
⌜ t₁ ·⁺ t₂ ⌝⁺                = ⌜ t₁ ⌝⁺ · ⌜ t₂ ⌝⁺
```

```
⌜ fix⁺ t ⌝⁺                  = fix ⌜ t ⌝⁺
⌜ z⁺ ⌝⁺                      = z
⌜ s⁺ ⌝⁺                      = s
⌜ natrec⁺ t₁ t₂ ⌝⁺           = natrec ⌜ t₁ ⌝⁺ ⌜ t₂ ⌝⁺
⌜ (t₁ ,⁺ t₂) ⌝⁺              = (⌜ t₁ ⌝⁺ , ⌜ t₂ ⌝⁺)
⌜ uncurry⁺ t ⌝⁺              = uncurry ⌜ t ⌝⁺
⌜ force⁺ t ⌝⁺                = ⌜ t ⌝⁺
⌜ return⁺ t ⌝⁺               = ⌜ t ⌝⁺
⌜ t₁ >>=⁺ t₂ ⌝⁺              = ⌜ t₂ ⌝⁺ · ⌜ t₁ ⌝⁺
⌜ ✓⁺ t ⌝⁺                    = ⌜ t ⌝⁺
⌜ pay⁺ n₂ t ⌝⁺               = ⌜ t ⌝⁺
⌜_⌝⁺ {Γ = Γ} (gpay⁺ C n₂ t)  =
  ≡-subst (\τ -> ⌜ Γ ⌝▷ ⊢ τ) (erase-lemma C) ⌜ t ⌝⁺
⌜_⌝⁺ {Γ = Γ} (gwait⁺ C n₂ t) =
  ≡-subst (\τ -> ⌜ Γ ⌝▷ ⊢ τ) (erase-lemma C) ⌜ t ⌝⁺

-- The definition of erasure actually used is equivalent to the
-- composition of the definitions above.

⌜_⌝-lemma : forall {Γ τ} (e : Γ ⊢ τ) -> ⌜ e ⌝ ≡ ⌜ tmToTm⁺ e ⌝⁺
⌜_⌝-lemma (var x)        = ≡-refl
⌜_⌝-lemma (λ t)          = ≡-cong λ_ (⌜_⌝-lemma t)
⌜_⌝-lemma (λ² t)         = ≡-cong λ²_ (⌜_⌝-lemma t)
⌜_⌝-lemma (t₁ · t₂)      = ≡-cong₂ _·_ (⌜_⌝-lemma t₁) (⌜_⌝-lemma t₂)
⌜_⌝-lemma (fix t)        = ≡-cong fix (⌜_⌝-lemma t)
⌜_⌝-lemma z              = ≡-refl
⌜_⌝-lemma s              = ≡-refl
⌜_⌝-lemma (natrec t₁ t₂) = ≡-cong₂ natrec (⌜_⌝-lemma t₁) (⌜_⌝-lemma t₂)
⌜_⌝-lemma (t₁ , t₂)      = ≡-cong₂ _,_ (⌜_⌝-lemma t₁) (⌜_⌝-lemma t₂)
⌜_⌝-lemma (uncurry t)    = ≡-cong uncurry (⌜_⌝-lemma t)
⌜_⌝-lemma (return t)     = ⌜_⌝-lemma t
⌜_⌝-lemma (t₁ >>= t₂)    = ≡-cong₂ _·_ (⌜_⌝-lemma t₂) (⌜_⌝-lemma t₁)
⌜_⌝-lemma (✓ t)          = ⌜_⌝-lemma t
⌜_⌝-lemma (pay n₂ t)     = ⌜_⌝-lemma t
⌜_⌝-lemma (gpay {σ = σ} {n₁ = n₁} C n₂ t)
  with ⌜ C [ Thunk (n₁ ∸ n₂) σ ] ⌝⋆
     | erase-lemma {m = n₁} {n = n₁ ∸ n₂} {τ = σ} C
... | ._ | ≡-refl = ⌜_⌝-lemma t
⌜_⌝-lemma (gwait {σ = σ} {n₁ = n₁} C n₂ t)
  with ⌜ C [ Thunk (n₂ + n₁) σ ] ⌝⋆
     | erase-lemma {m = n₁} {n = n₂ + n₁} {τ = σ} C
... | ._ | ≡-refl = ⌜_⌝-lemma t
```

3.5 Substitution

```
------------------------------------------------------------------------
-- Substitutions
------------------------------------------------------------------------

module Substitution where

open import Kind
open import TypeSystem
open import Erasure as E

open import Data.Function using (id)
open import Data.Nat using (_+_)

------------------------------------------------------------------------
-- General substitution machinery

-- Uses idea from "Type-Preserving Renaming and Substitution" by Conor
-- McBride to enable defining both term and variable substitutions
-- without having to write several functions which traverse terms.

-- Definition of what a substitution is.

module SubstDef (_•_ : TmLike) -- The substitution replaces variables
                               -- with this kind of "term".
                where

  infix  10 _⇒_
  infixl 20 _▷_

  -- Substitutions of type Γ ⇒ Δ, when applied, take things with
  -- variables in Γ and give things with variables in Δ. (This concept
  -- is sometimes written with the arrow in the other direction.)

  data _⇒_ {k : Kind} : Ctxt k -> Ctxt k -> Set where
    ∅   : forall {Δ}     -> ε ⇒ Δ
    _▷_ : forall {Γ Δ τ} -> Γ ⇒ Δ -> Δ • τ -> Γ ▷ τ ⇒ Δ

  -- The type of functions applying substitutions to things.

  Applier : TmLike -> TmLike -> Set
  Applier _•₁_ _•₂_ =    forall {k} {Γ Δ : Ctxt k} {τ}
                      -> Γ •₁ τ -> Γ ⇒ Δ -> Δ •₂ τ

  -- You get lots of substitution machinery defined for you (see
  -- below) if you define a SubstKit, ...
```

```
  record SubstKit : Set where
    vr     :  forall {k} {Γ : Ctxt k} {τ}   -> Γ ∋ τ -> Γ • τ
    weaken :  forall {k} {Γ : Ctxt k} {σ τ} -> Γ • τ -> Γ ▷ σ • τ
    tm     :  forall {k} {Γ : Ctxt k} {τ}   -> Γ • τ -> Γ ⊢ τ
    wait   :  forall {Γ τ n₁} C n₂
           -> Γ • C [ Thunk n₁ τ ] -> Γ • C [ Thunk (n₂ + n₁) τ ]

  -- ...and more if you define a SubstKit⁺.

  record SubstKit⁺ : Set where
    kit  : SubstKit
    _/•_ : Applier _•_ _•_

open SubstDef using (SubstKit; SubstKit⁺)

-- Various substitution-related functions.

module Subst {_•_ : TmLike}
             (kit : SubstKit _•_)
             where

  private
    open module D = SubstDef _•_ public
    open module K = SubstKit kit

  infix  70 _↑
  infixl 48 _/⊢_ _/∋_

  -- Weakening of substitutions.

  wk⇒ : forall {k} {Γ Δ : Ctxt k} {σ} -> Γ ⇒ Δ -> Γ ⇒ Δ ▷ σ
  wk⇒ ∅       = ∅
  wk⇒ (ρ ▷ t) = wk⇒ ρ ▷ weaken t

  -- Lifting of substitutions.

  _↑ : forall {k} {Γ Δ : Ctxt k} {σ} -> Γ ⇒ Δ -> Γ ▷ σ ⇒ Δ ▷ σ
  ρ ↑ = wk⇒ ρ ▷ vr vz

  -- Identity.

  idˢ : forall {k} {Γ : Ctxt k} -> Γ ⇒ Γ
  idˢ {Γ = ε}     = ∅
  idˢ {Γ = Γ ▷ τ} = idˢ ↑

  -- Weakening.

  wk : forall {k} {Γ : Ctxt k} {σ} -> Γ ⇒ Γ ▷ σ
  wk = wk⇒ idˢ
```

```
  -- Substitution of a single variable.

  sub : forall {k} {Γ : Ctxt k} {τ} -> Γ • τ -> Γ ▷ τ ⇒ Γ
  sub t = id^s ▷ t

  -- Substitution application for variables.

  _/∋_ : Applier _∋_ _•_
  vz           /∋ (ρ ▷ t) = t
  vs x         /∋ (ρ ▷ t) = x /∋ ρ
  gwait∋ C m x /∋ ρ       = wait C m (x /∋ ρ)

  -- Substitution application for terms.

  _/⊢_ : Applier _⊢_ _⊢_
  var x        /⊢ ρ = tm (x /∋ ρ)
  λ t          /⊢ ρ = λ (t /⊢ ρ ↑)
  λ² t         /⊢ ρ = λ² (t /⊢ ρ ↑ ↑)
  t₁ · t₂      /⊢ ρ = (t₁ /⊢ ρ) · (t₂ /⊢ ρ)
  fix t        /⊢ ρ = fix (t /⊢ ρ ↑)
  z            /⊢ ρ = z
  s            /⊢ ρ = s
  natrec t₁ t₂ /⊢ ρ = natrec (t₁ /⊢ ρ) (t₂ /⊢ ρ ↑ ↑)
  (t₁ , t₂)    /⊢ ρ = (t₁ /⊢ ρ) , (t₂ /⊢ ρ)
  uncurry t    /⊢ ρ = uncurry (t /⊢ ρ ↑ ↑)
  return t     /⊢ ρ = return (t /⊢ ρ)
  (t₁ >>= t₂)  /⊢ ρ = t₁ /⊢ ρ >>= t₂ /⊢ ρ
  (✓ t)        /⊢ ρ = ✓ t /⊢ ρ
  pay n t      /⊢ ρ = pay n (t /⊢ ρ)
  gpay C n t   /⊢ ρ = gpay C n (t /⊢ ρ)
  gwait C n t  /⊢ ρ = gwait C n (t /⊢ ρ)

-- Subst⁺ defines composition. This function uses a stronger
-- substitution kit, which is sometimes defined using the Subst module
-- (see the examples below); this is the reason for placing
-- composition in a separate module.

module Subst⁺ {_•_  : TmLike}
              (kit⁺ : SubstKit⁺ _•_)
              where

  private
    open module K = SubstDef.SubstKit⁺ _•_ kit⁺
    open module S = Subst kit public

  infixl 70 _∘^s_

  _∘^s_ : forall {k} {Γ Δ X : Ctxt k} -> Γ ⇒ Δ -> Δ ⇒ X -> Γ ⇒ X
  ∅         ∘^s ρ₂ = ∅
  (ρ₁ ▷ t) ∘^s ρ₂ = ρ₁ ∘^s ρ₂ ▷ (t /• ρ₂)
```

```
-- Erasing terms in substitutions.

module Erasure (• : TmLike) (⌜_⌝• : TmEraser •) where

  private
    open module S = SubstDef •

  ⌜_⌝⇒ : forall {Γ Δ} -> Γ ⇒ Δ -> ⌜ Γ ⌝▷ ⇒ ⌜ Δ ⌝▷
  ⌜ ∅ ⌝⇒     = ∅
  ⌜ ρ ▷ t ⌝⇒ = ⌜ ρ ⌝⇒ ▷ ⌜ t ⌝•

------------------------------------------------------------------------
-- Instantiations of the general structures above

-- Variables.

varKit : SubstKit _∋_
varKit = record
  { vr     = id
  ; weaken = vs
  ; tm     = var
  ; wait   = gwait∋
  }

varKit⁺ : SubstKit⁺ _∋_
varKit⁺ = record { kit  = varKit
                 ; _/•_ = Subst._/∋_ varKit
                 }

module VarSubst where
  private
    open module S  = Subst⁺ varKit⁺ public
    open module SE = Erasure _∋_ ⌜_⌝∋ public

-- Terms.

-- Note that term substitutions are not used by the formalisation.
-- They are included to illustrate the utility of the machinery
-- defined above, and to show that it is possible to define full term
-- substitutions.

tmKit : SubstKit _⊢_
tmKit = record
  { vr     = var
  ; weaken = \t -> t /⊢ wk
  ; tm     = id
  ; wait   = gwait
  }
  where open module VS = VarSubst
```

```
tmKit⁺ : SubstKit⁺ _⊢_
tmKit⁺ = record { kit  = tmKit
                ; _/•_ = Subst._/⊢_ tmKit
                }

module TmSubst where
  private
    open module S  = Subst⁺ tmKit⁺ public
    open module SE = Erasure _⊢_ ⌜_⌝ public
```

3.6 Value

```
------------------------------------------------------------------------
-- Values
------------------------------------------------------------------------

module Value where

open import Kind
open import TypeSystem
open import Erasure
open import Substitution
open VarSubst

open import Data.Nat using (zero; suc; _+_)

infixr 60 _,⇓_
infix  50 λ⇓_ λ²⇓_
infixl 48 _/⊢⇓_
infix  10 _⊢⇓_

-- Values. A value is the result of evaluating a term in a "full" heap
-- (where all variables are bound to something) to WHNF. Whenever
-- possible these constructors take variables as arguments, to ensure
-- full sharing if they are evaluated further.

-- Just as before simple values and thunked, run-time values share the
-- same definition.

data _⊢⇓_ : TmLike where
  -- Lambda.
  λ⇓_  :  forall {k} {Γ : Ctxt k} {σ τ n}
       -> Γ ▷ σ ⊢ Thunk? n τ
       -> Γ ⊢⇓ σ → Thunk? (suc' k n) τ
  λ²⇓_ :  forall {k} {Γ : Ctxt k} {σ₁ σ₂ τ n}
       -> Γ ▷ σ₁ ▷ σ₂ ⊢ Thunk? n τ
       -> Γ ⊢⇓ σ₁ → σ₂ → Thunk? (suc' k n) τ

  -- Natural numbers. An unapplied s (the successor constructor) is
  -- represented by s⇓, while s₁⇓ is used when s has been applied to
  -- something.
  z⇓      :  forall {k} {Γ : Ctxt k} -> Γ ⊢⇓ Nat
  s⇓      :  forall {k} {Γ : Ctxt k} -> Γ ⊢⇓ Nat → Nat
  s₁⇓     :  forall {k} {Γ : Ctxt k} -> Γ ∋ Nat -> Γ ⊢⇓ Nat
  natrec⇓ :  forall {k} {Γ : Ctxt k} {σ n}
          -> Γ ∋ Thunk? n σ
          -> Γ ▷ Nat ▷ Thunk? (suc' k n) σ ⊢ Thunk? n σ
          -> Γ ⊢⇓ Nat → Thunk? (suc' k n) σ
```

```
  -- Products.
  _,⇓_     :  forall {k} {Γ : Ctxt k} {σ τ}
           -> Γ ∋ σ -> Γ ∋ τ -> Γ ⊢⇓ σ × τ
  uncurry⇓ :  forall {k} {Γ : Ctxt k} {σ₁ σ₂ τ n}
           -> Γ ▷ σ₁ ▷ σ₂ ⊢ Thunk? n τ
           -> Γ ⊢⇓ σ₁ × σ₂ → Thunk? (suc' k n) τ

  -- Thunks.
  return⋆⇓ : forall {Γ σ n} -> Γ ⊢⇓ σ -> Γ ⊢⇓ Thunk n σ

-- gwait for values. This definition takes advantage of the fact that
-- variables can be annotated using gwait∋.

gwait⇓ :  forall {Γ σ n₁} C n₂
       -> Γ ⊢⇓ C [ Thunk n₁ σ ]
       -> Γ ⊢⇓ C [ Thunk (n₂ + n₁) σ ]
gwait⇓ ●            m (return⋆⇓ v) = return⋆⇓ v
gwait⇓ (const● σ)   m v            = v
gwait⇓ (C ●× σ)     m (x ,⇓ y)     = (gwait∋ C m x ,⇓ y)
gwait⇓ (σ ×● C)     m (x ,⇓ y)     = (x ,⇓ gwait∋ C m y)
gwait⇓ (Thunk● n C) m (return⋆⇓ v) = return⋆⇓ (gwait⇓ C m v)

-- return⋆ for terms.

return⋆ : forall {Γ τ n} -> Γ ⊢ τ -> Γ ⊢ Thunk n τ
return⋆ {n = zero}  e = return e
return⋆ {n = suc n} e = ✓ return⋆ e

-- Converts a value to the corresponding term.

valToTm : forall {k} {Γ : Ctxt k} {τ} -> Γ ⊢⇓ τ -> Γ ⊢ τ
valToTm (λ⇓ t)          = λ t
valToTm (λ²⇓ t)         = λ² t
valToTm z⇓              = z
valToTm s⇓              = s
valToTm (s₁⇓ x)         = s · var x
valToTm (natrec⇓ x₁ t₂) = natrec (var x₁) t₂
valToTm (x₁ ,⇓ x₂)      = (var x₁ , var x₂)
valToTm (uncurry⇓ t)    = uncurry t
valToTm (return⋆⇓ v)    = return⋆ (valToTm v)

-- Variable substitutions can be applied to values.

_/⊢⇓_ : Applier _⊢⇓_ _⊢⇓_
λ⇓ t          /⊢⇓ ρ = λ⇓ (t /⊢ ρ ↑)
λ²⇓ t         /⊢⇓ ρ = λ²⇓ (t /⊢ ρ ↑ ↑)
z⇓            /⊢⇓ ρ = z⇓
s⇓            /⊢⇓ ρ = s⇓
s₁⇓ x         /⊢⇓ ρ = s₁⇓ (x /∋ ρ)
natrec⇓ x₁ t₂ /⊢⇓ ρ = natrec⇓ (x₁ /∋ ρ) (t₂ /⊢ ρ ↑ ↑)
```

```
(x₁ ,⇓ x₂)     /⊢⇓ ρ = (x₁ /∋ ρ) ,⇓ (x₂ /∋ ρ)
uncurry⇓ t     /⊢⇓ ρ = uncurry⇓ (t /⊢ ρ ↑ ↑)
return⋆⇓ v     /⊢⇓ ρ = return⋆⇓ (v /⊢⇓ ρ)

-- Erases the value.

⌜_⌝⇓ : TmEraser _⊢⇓_
⌜ λ⇓ e ⌝⇓           = λ⇓ ⌜ e ⌝
⌜ λ²⇓ e ⌝⇓          = λ²⇓ ⌜ e ⌝
⌜ z⇓ ⌝⇓             = z⇓
⌜ s⇓ ⌝⇓             = s⇓
⌜ s₁⇓ x ⌝⇓          = s₁⇓ ⌜ x ⌝∋
⌜ natrec⇓ x₁ e₂ ⌝⇓ = natrec⇓ ⌜ x₁ ⌝∋ ⌜ e₂ ⌝
⌜ x ,⇓ y ⌝⇓         = ⌜ x ⌝∋ ,⇓ ⌜ y ⌝∋
⌜ uncurry⇓ e ⌝⇓     = uncurry⇓ ⌜ e ⌝
⌜ return⋆⇓ v ⌝⇓     = ⌜ v ⌝⇓
```

4 Simple operational semantics

This section defines the simple operational semantics.

4.1 Heap

```
------------------------------------------------------------------------
-- Heaps
------------------------------------------------------------------------

open import TypeSystem

module Heap (_•_ : TmLike) where

open import Kind

infixl 20 _▷ʰ_

data Heap {k : Kind} : Ctxt k -> Set where
  ∅ʰ   : Heap ε
  _▷ʰ_ : forall {Γ τ} -> Heap Γ -> Γ • τ -> Heap (Γ ▷ τ)

-- Note that Heap Γ ≇ Γ ⇒ ε; if that were the case the _▷ʰ_
-- constructor would have type
--
--   _▷ʰ_ : forall {Γ τ} -> Heap Γ -> ε • τ -> Heap (Γ ▷ τ),
--
-- i.e. all elements in the heap would have to be closed.
```

4.2 Heap.Erasure

```
--------------------------------------------------------------------------
-- Erasing heaps
--------------------------------------------------------------------------

open import TypeSystem
open import Erasure

module Heap.Erasure (• : TmLike) (⌜_⌝• : TmEraser •) where

import Heap
private
  open module H = Heap •

⌜_⌝ʰ : forall {Γ} -> Heap Γ -> Heap ⌜ Γ ⌝▷
⌜ ∅ʰ ⌝ʰ      = ∅ʰ
⌜ Σ ▷ʰ t ⌝ʰ = ⌜ Σ ⌝ʰ ▷ʰ ⌜ t ⌝•
```

4.3 OperationalSemantics.Binding

```
------------------------------------------------------------------------
-- Heap bindings
------------------------------------------------------------------------

module OperationalSemantics.Binding where

open import Kind
open import TypeSystem
open import Erasure
open import Substitution
open VarSubst
open import Value
import Heap
import Heap.Erasure

open import Data.Nat using (ℕ; _∸_)
open import Data.List using ([]; _∷_; foldr) renaming ([_] to List)
open import Logic using (_≡_; ≡-refl)
open import Relation.Binary.PropositionalEquality
  using (≡-subst; ≡-cong; ≡-sym)

infixl 48 _/↦_
infix  30 ↦_ ↦ⁿ_
infix  10 _↦_

------------------------------------------------------------------------
-- Multiple applications of Thunk

-- Thunks (n₁ ∷ n₂ ∷ []) τ ≡ Thunk n₁ (Thunk n₂ τ).

Thunks : List ℕ -> Ty thunked -> Ty thunked
Thunks ns τ = foldr Thunk τ ns

Thunks-lemma : forall ns {τ} -> ⌜ Thunks ns τ ⌝⋆ ≡ ⌜ τ ⌝⋆
Thunks-lemma []       = ≡-refl
Thunks-lemma (n ∷ ns) = Thunks-lemma ns

-- Generalised variant which vanishes in the simple case.

Thunks? : forall {k} -> Maybe (List ℕ) k -> Ty k -> Ty k
Thunks? {k = simple}  ns τ = τ
Thunks? {k = thunked} ns τ = Thunks ns τ

-- Thunks● (n₁ ∷ n₂ ∷ []) C ≡ Thunk● n₁ (Thunk● n₂ C).

Thunks● : List ℕ -> PayCtxt -> PayCtxt
Thunks● ns C = foldr Thunk● C ns
```

```
Thunks●-lemma :  forall ns C {τ}
              -> Thunks ns (C [ τ ]) ≡ Thunks● ns C [ τ ]
Thunks●-lemma []       C = ≡-refl
Thunks●-lemma (n :: ns) C = ≡-cong (Thunk n) (Thunks●-lemma ns C)

-- Removes one application of return⋆⇓ for every element in ns.

drop-return⋆⇓ :  forall {Γ τ} ns
              -> Γ ⊢⇓ Thunks ns τ -> Γ ⊢⇓ τ
drop-return⋆⇓ []       v            = v
drop-return⋆⇓ (n :: ns) (return⋆⇓ v) = drop-return⋆⇓ ns v

------------------------------------------------------------------------
-- Heap bindings

-- The heap bindings carry credit (in "ns"; see MainResult),
-- corresponding to terms which have been paid off but not yet
-- evaluated. The type of the binding reflects this credit.

data _↦_ : TmLike where
  ↦ⁿˢ :  forall {k} {Γ : Ctxt k} {τ} ns
      -> Γ ⊢ Thunks? ns τ -> Γ ↦ τ

-- Some special cases (these two cases could replace ↦ⁿˢ if there was
-- no generalised pay):

-- Ordinary binding.

↦_ : forall {k} {Γ : Ctxt k} {τ} -> Γ ⊢ τ -> Γ ↦ τ
↦_ {k = simple}  e = ↦ⁿˢ _  e
↦_ {k = thunked} e = ↦ⁿˢ [] e

-- A binding which has been (partially) paid off.

↦ⁿ_ : forall {Γ τ n} -> Γ ⊢ Thunk n τ -> Γ ↦ τ
↦ⁿ_ {n = n} e = ↦ⁿˢ (n :: []) e

-- Erases the heap bindings.

⌜_⌝↦ : TmEraser _↦_
⌜ ↦ⁿˢ ns e ⌝↦ = ↦ ≡-subst (_⊢_ _) (Thunks-lemma ns) ⌜ e ⌝

open module Heap↦ = Heap _↦_
module Heap↦Erasure = Heap.Erasure _↦_ ⌜_⌝↦

-- Variable substitutions can be applied to heap bindings.

_/↦_ : forall {k} {Γ Δ : Ctxt k} {τ} -> Γ ↦ τ -> Γ ⇒ Δ -> Δ ↦ τ
↦ⁿˢ ns e /↦ ρ = ↦ⁿˢ ns (e /⊢ ρ)
```

```
-- Paying off bindings.

gpay↦ :  forall {Γ σ n} C m
      -> Γ ↦ C [ Thunk n σ ] -> Γ ↦ C [ Thunk (n ∸ m) σ ]
gpay↦ C m (↦ⁿˢ ns e) =
  ↦ⁿˢ (m :: ns) (cast₁ (gpay (Thunks● ns C) m (cast₂ e)))
  where
  cast₁ = ≡-subst (_⊢_ _) (≡-sym (Thunks●-lemma (m :: ns) C))
  cast₂ = ≡-subst (_⊢_ _) (Thunks●-lemma ns C)
```

4.4 OperationalSemantics.Simple

```
------------------------------------------------------------------------
-- Lazy operational semantics
------------------------------------------------------------------------

-- The operational semantics is inspired by Lanchbury's natural
-- semantics for lazy evaluation.

-- The evaluation relation is annotated by the number of steps
-- necessary to evaluate a term to weak head normal form. Note that
-- reindexing of de Bruijn indices is not counted, since this is not
-- typically part of efficient implementations. Similarly,
-- substituting a variable for another, adding heap bindings, or
-- looking up variables are not counted.

-- Note that the type of the relation ensures that the semantics is
-- well-scoped and well-typed, and that the result is in weak head
-- normal form.

-- The semantics is indexed on a substitution (of type Γ ⇒ Δ). This is
-- a variable substitution, i.e. a renaming. The resulting heap may
-- have more bindings than the initial one, and this substitution
-- describes how one can "weaken" the initial context to arrive at the
-- final one.

-- The inference rules in comments below describe the rules using
-- ordinary variables. They assume that all variables are suitably
-- fresh.

module OperationalSemantics.Simple where

open import Kind
open import TypeSystem
open import Substitution
open VarSubst
open import Value
open import OperationalSemantics.Binding
open Heap→

open import Data.Nat using (ℕ; zero; suc; _+_)

infix 10 _|_⇓_⇖_|_✓_ _|_•⇓_⇖_|_✓_
```

```
mutual

  data _|_⇓_⇜_|_✓_
    :  forall {Γ Δ : Ctxt simple} {τ}
    -> Heap Γ -> Γ ⊢ τ
    -> Γ ⇒ Δ -> Heap Δ -> Δ ⊢⇓ τ
    -> ℕ
    -> Set where

    -- Values.

    -- ———————————————————————
    --  Σ | λx.t ⇓ Σ | λx.t ✓ 0

    eval-λ
      :  forall {Γ σ τ Σ} {e : Γ ▷ σ ⊢ τ}
      -> Σ | λ e ⇓ id^s ⇜ Σ | λ⇓ e ✓ zero

    -- ———————————————————————
    --  Σ | λxy.t ⇓ Σ | λxy.t ✓ 0

    eval-λ²
      :  forall {Γ σ₁ σ₂ τ Σ} {e : Γ ▷ σ₁ ▷ σ₂ ⊢ τ}
      -> Σ | λ² e ⇓ id^s ⇜ Σ | λ²⇓ e ✓ zero

    -- ———————————————————
    --  Σ | z ⇓ Σ | z ✓ 0

    eval-z
      :  forall {Γ} {Σ : Heap Γ}
      -> Σ | z ⇓ id^s ⇜ Σ | z⇓ ✓ zero

    -- ———————————————————
    --  Σ | s ⇓ Σ | s ✓ 0

    eval-s
      :  forall {Γ} {Σ : Heap Γ}
      -> Σ | s ⇓ id^s ⇜ Σ | s⇓ ✓ zero

    -- ———————————————————————————————————————————————————————————————
    --  Σ | natrec t₁ (λxy.t₂) ⇓ Σ, x₁ ↦ t₁ | natrec x₁ (λxy.t₂) ✓ 0

    eval-natrec
      :  forall {Γ σ Σ} {e₁ : Γ ⊢ σ} {e₂}
      -> Σ | natrec e₁ e₂ ⇓ wk ⇜ Σ ▷^h ↦ e₁
                          | natrec⇓ vz (e₂ /⊢ wk ↑ ↑) ✓ zero
```

```
-- ------------------------------------------------------------
--  Σ | (t₁ , t₂) ⇓ Σ, x ↦ t₁, y ↦ t₂ | (x , y) ✓ 0

eval-,
  :  forall {Γ σ₁ σ₂ Σ} {e₁ : Γ ⊢ σ₁} {e₂ : Γ ⊢ σ₂}
  -> Σ | (e₁ , e₂) ⇓ wk ∘ˢ wk ∝ Σ ▷ʰ ↦ e₁ ▷ʰ ↦ e₂ /⊢ wk
                  | (vs vz ,⇓ vz) ✓ zero

-- ------------------------------------------------------------
--  Σ | uncurry (λxy.t) ⇓ Σ | uncurry (λxy.t) ✓ 0

eval-uncurry
  :  forall {Γ σ₁ σ₂ τ Σ} {e : Γ ▷ σ₁ ▷ σ₂ ⊢ τ}
  -> Σ | uncurry e ⇓ idˢ ∝ Σ | uncurry⇓ e ✓ zero

-- Variables.

--             Σ₁ | t ⇓ Σ₂             | v ✓ n
-- ------------------------------------------------------
--  Σ₁, x ↦ t, Σ' | x ⇓ Σ₂, x ↦ v, Σ' | v ✓ n

eval-vz
  :  forall {Γ Δ τ n ρ Σ₁ Σ₂} {e₁ : Γ ⊢ τ} {v : Δ ⊢⇓ τ}
  -> Σ₁          | e₁     ⇓ ρ   ∝ Σ₂                | v        ✓ n
  -> Σ₁ ▷ʰ ↦ e₁ | var vz ⇓ ρ ↑ ∝ Σ₂ ▷ʰ ↦ valToTm v | v /⊢⇓ wk ✓ n

eval-vs
  :  forall {Γ Δ σ τ n ρ Σ₁ Σ₂}
            {x : Γ ∋ τ} {b : Γ ↦ σ} {v : Δ ⊢⇓ _}
  -> Σ₁       | var x      ⇓ ρ   ∝ Σ₂           | v        ✓ n
  -> Σ₁ ▷ʰ b | var (vs x) ⇓ ρ ↑ ∝ Σ₂ ▷ʰ b /↦ ρ | v /⊢⇓ wk ✓ n

-- Fixpoints.

--  Σ₁, x ↦ fix (λx.t) | t          ⇓ Σ₂ | v ✓ n
-- ------------------------------------------------------
--  Σ₁                 | fix (λx.t) ⇓ Σ₂ | v ✓ 1 + n

eval-fix
  :  forall {Γ Δ σ n Σ₁ Σ₂} {ρ : _ ⇒ Δ} {e : Γ ▷ σ ⊢ σ} {v}
  -> Σ₁ ▷ʰ ↦ fix e | e     ⇓ ρ       ∝ Σ₂ | v ✓ n
  -> Σ₁            | fix e ⇓ wk ∘ˢ ρ ∝ Σ₂ | v ✓ suc n
```

```
  -- Applications.

  --  Σ₁           | t₁      ⇓ Σ₂ | v₁ ✓ n₁
  --  Σ₂, x₂ ↦ t₂ | v₁ • x₂ ⇓ Σ₃ | v  ✓ n₂
  -- ──────────────────────────────────────
  --  Σ₁           | t₁ · t₂ ⇓ Σ₂ | v  ✓ n₁ + n₂

  eval-·
    :  forall {Γ Δ X σ τ n₁ n₂ Σ₁ Σ₂ Σ₃}
              {ρ₁ : Γ ⇒ Δ} {ρ₂ : _ ⇒ X}
              {e₁ : Γ ⊢ σ → τ} {e₂ : Γ ⊢ σ} {v v₁}
    -> Σ₁               | e₁      ⇓ ρ₁          ⋉ Σ₂ | v₁ ✓ n₁
    -> Σ₂ ▷ʰ ↦ e₂ /⊢ ρ₁ | v₁     •⇓ ρ₂          ⋉ Σ₃ | v  ✓ n₂
    -> Σ₁               | e₁ · e₂ ⇓ wk⇒ ρ₁ ○ˢ ρ₂ ⋉ Σ₃ | v  ✓ (n₁ + n₂)

data _|_•⇓_⋉_|_✓_
  :  forall {Γ Δ : Ctxt simple} {σ τ}
  -> Heap (Γ ▷ σ) -> Γ ⊢⇓ σ → τ
  -> Γ ▷ σ ⇒ Δ -> Heap Δ -> Δ ⊢⇓ τ
  -> ℕ
  -> Set where

  -- ──────────────────────────
  --  Σ | s • x₂ ⇓ Σ | s x₂ ✓ 0

  eval-·-s
    :  forall {Γ} {Σ : Heap (Γ ▷ Nat)}
    -> Σ | s⇓ •⇓ idˢ ⋉ Σ | s₁⇓ vz ✓ zero

  --  Σ₁ | t₁[x := x₂] ⇓ Σ₂ | v ✓ n
  -- ──────────────────────────────────
  --  Σ₁ | λx.t₁ • x₂  ⇓ Σ₂ | v ✓ 1 + n

  eval-·-λ
    :  forall {Γ Δ n σ τ Σ₁ Σ₂} {ρ : _ ⇒ Δ} {e₁ : Γ ▷ σ ⊢ τ} {v}
    -> Σ₁ | e₁     ⇓ ρ ⋉ Σ₂ | v ✓ n
    -> Σ₁ | λ⇓ e₁ •⇓ ρ ⋉ Σ₂ | v ✓ suc n

  -- Note that nothing is charged for the following application; only
  -- when all arguments have been supplied (and hence evaluation of
  -- the right hand side can commence) is something charged. If you do
  -- not agree with this cost model you should avoid partial
  -- applications.

  -- ──────────────────────────────────────
  --  Σ₁ | λxy.t₁ • x₂ ⇓ Σ₁ | λy.t₁[x := x₂] ✓ 0

  eval-·-λ²
    :  forall {Γ σ₁ σ₂ τ Σ} {e₁ : Γ ▷ σ₁ ▷ σ₂ ⊢ τ}
    -> Σ | λ²⇓ e₁ •⇓ idˢ ⋉ Σ | λ⇓ e₁ ✓ zero
```

```
--  Σ₁ | x₂                          ⇓ Σ₂ | z ✓ n₁
--  Σ₂ | x₁¹                         ⇓ Σ₃ | v ✓ n₂
-- ──────────────────────────────────────────────────────
--  Σ₁ | natrec x₁¹ (λxy.t₁²) • x₂ ⇓ Σ₃ | v ✓ 1 + n₁ + n₂

eval-·-natrec-zero
  :  forall {Γ₁ Γ₂ Γ₃ n₁ n₂ τ ρ₁ Σ₁ Σ₂ Σ₃}
            {ρ₂ : Γ₂ ⇒ Γ₃} {x₁¹ : Γ₁ ∋ τ} {e₁²} {v}
  -> let x₁¹' = var x₁¹ /⊢ wk ∘ˢ ρ₁
         ρ₃   = ρ₁ ∘ˢ ρ₂
     in
     Σ₁ | var vz           ⇓ ρ₁ ∝ Σ₂ | z⇓ ✓ n₁
  -> Σ₂ | x₁¹',            ⇓ ρ₂ ∝ Σ₃ | v  ✓ n₂
  -> Σ₁ | natrec⇓ x₁¹ e₁² •⇓ ρ₃ ∝ Σ₃ | v  ✓ suc (n₁ + n₂)

--  Σ₁  | x₂                          ⇓ Σ₂ | s x₃ ✓ n₁
--  Σ₂, y ↦ natrec x₁¹ (λxy.t₁²) · x₃
--      | t₁²[x := x₃]                ⇓ Σ₃ | v    ✓ n₂
-- ──────────────────────────────────────────────────────────
--  Σ₁  | natrec x₁¹ (λxy.t₁²) • x₂ ⇓ Σ₃ | v    ✓ 1 + n₁ + n₂

eval-·-natrec-suc
  :  forall {Γ₁ Γ₂ Γ₃ n₁ n₂ τ ρ₁ ρ₂ Σ₁ Σ₂ Σ₃}
            {x₁¹ : Γ₁ ∋ τ} {e₁²}
            {x₂ : Γ₂ ∋ Nat} {v : Γ₃ ⊢⇓ τ}
  -> let Σ₂'  = Σ₂ ▷ʰ ↦ (natrec (var x₁¹) e₁² /⊢ (wk ∘ˢ ρ₁)) · var x₂
         e₁²' = e₁² /⊢ (wk ∘ˢ ρ₁) ↑ ↑ /⊢ sub x₂ ↑
         ρ₃   = wk⇒ ρ₁ ∘ˢ ρ₂
     in
     Σ₁  | var vz           ⇓ ρ₁ ∝ Σ₂ | s₁⇓ x₂ ✓ n₁
  -> Σ₂' | e₁²',            ⇓ ρ₂ ∝ Σ₃ | v    ✓ n₂
  -> Σ₁  | natrec⇓ x₁¹ e₁² •⇓ ρ₃ ∝ Σ₃ | v    ✓ suc (n₁ + n₂)

--  Σ₁ | x₂                    ⇓ Σ₂ | (x₃ , y₃) ✓ n₁
--  Σ₂ | t₁[x := x₃, y := y₃]  ⇓ Σ₃ | v          ✓ n₂
-- ──────────────────────────────────────────────────────
--  Σ₁ | uncurry (λxy.t₁) • x₂ ⇓ Σ₃ | v          ✓ 1 + n₁ + n₂

eval-·-uncurry
  :  forall {Γ₁ Γ₂ Γ₃ n₁ n₂ σ₁ σ₂ τ ρ₁ ρ₂ Σ₁ Σ₂ Σ₃}
            {e₁ : Γ₁ ▷ σ₁ ▷ σ₂ ⊢ τ}
            {x₂ : Γ₂ ∋ σ₁} {y₂ : Γ₂ ∋ σ₂} {v : Γ₃ ⊢⇓ τ}
  -> let e₁' = e₁ /⊢ (wk ∘ˢ ρ₁) ↑ ↑ /⊢ sub x₂ ↑ /⊢ sub y₂ in
     Σ₁ | var vz       ⇓ ρ₁       ∝ Σ₂ | (x₂ ,⇓ y₂) ✓ n₁
  -> Σ₂ | e₁'          ⇓ ρ₂       ∝ Σ₃ | v          ✓ n₂
  -> Σ₁ | uncurry⇓ e₁ •⇓ ρ₁ ∘ˢ ρ₂ ∝ Σ₃ | v          ✓ suc (n₁ + n₂)
```

5 Thunked operational semantics

This section defines the thunked operational semantics. Note that it is not necessary to understand this definition in order to trust the correctness proof, since the thunked semantics is sound and complete with respect to the simple one (see Section 6). There is also a simplified (but less compositional) correctness result which does not mention the thunked semantics at all.

5.1 PayCtxt

```
------------------------------------------------------------------------
-- PayCtxt-related functions
------------------------------------------------------------------------

module PayCtxt where

open import Kind
open import TypeSystem

open import Logic using (_≡_; ≡-refl)
open import Relation.Binary.PropositionalEquality using (≡-cong)

------------------------------------------------------------------------
-- Composing contexts

infixr 42 _○_

_○_ : PayCtxt → PayCtxt → PayCtxt
●           ○ C₂ = C₂
const● σ    ○ C₂ = const● σ
(C₁ ●× σ)   ○ C₂ = (C₁ ○ C₂) ●× σ
(σ ×● C₁)   ○ C₂ = σ ×● (C₁ ○ C₂)
Thunk● n C₁ ○ C₂ = Thunk● n (C₁ ○ C₂)

○-lemma : ∀ C₁ C₂ {σ} → C₁ ○ C₂ [ σ ] ≡ C₁ [ C₂ [ σ ] ]
○-lemma ●             C₂ = ≡-refl
○-lemma (const● σ)    C₂ = ≡-refl
○-lemma (C₁ ●× σ)     C₂ = ≡-cong (λ τ → τ × σ) (○-lemma C₁ C₂)
○-lemma (σ ×● C₁)     C₂ = ≡-cong (λ τ → σ × τ) (○-lemma C₁ C₂)
○-lemma (Thunk● n C₁) C₂ = ≡-cong (Thunk n) (○-lemma C₁ C₂)

------------------------------------------------------------------------
-- "Inverting" contexts

-- "Inverting" here means swapping the order of the context elements,
-- so that the outermost context constructor is the first one applied
-- to the hole-filling type. In other words, the context is turned
-- outside-in.
```

```
-- Instantiates a context outside-in.

infix 41 _⁻¹[_]

_⁻¹[_] : PayCtxt -> Ty thunked -> Ty thunked
●          ⁻¹[ τ ] = τ
const● σ   ⁻¹[ τ ] = σ
(C ●× σ)   ⁻¹[ τ ] = C ⁻¹[ τ × σ ]
(σ ×● C)   ⁻¹[ τ ] = C ⁻¹[ σ × τ ]
Thunk● n C ⁻¹[ τ ] = C ⁻¹[ Thunk n τ ]

-- "Inverts" a context and composes the result with another.

infixr 42 _⁻¹∘_

_⁻¹∘_ : PayCtxt -> PayCtxt -> PayCtxt
●           ⁻¹∘ C₂ = C₂
const● σ    ⁻¹∘ C₂ = const● σ
(C₁ ●× σ)   ⁻¹∘ C₂ = C₁ ⁻¹∘ (C₂ ●× σ)
(σ ×● C₁)   ⁻¹∘ C₂ = C₁ ⁻¹∘ (σ ×● C₂)
Thunk● n C₁ ⁻¹∘ C₂ = C₁ ⁻¹∘ Thunk● n C₂

-- "Inverts" a context.

infix 43 _⁻¹

_⁻¹ : PayCtxt -> PayCtxt
C ⁻¹ = C ⁻¹∘ ●

-- The definitions are closely related.

⁻¹[]-lemma : forall C C' {σ} -> C ⁻¹[ C' [ σ ] ] ≡ C ⁻¹∘ C' [ σ ]
⁻¹[]-lemma ●            C' = ≡-refl
⁻¹[]-lemma (const● σ)   C' = ≡-refl
⁻¹[]-lemma (C ●× σ)     C' = ⁻¹[]-lemma C (C' ●× σ)
⁻¹[]-lemma (σ ×● C)     C' = ⁻¹[]-lemma C (σ ×● C')
⁻¹[]-lemma (Thunk● n C) C' = ⁻¹[]-lemma C (Thunk● n C')

⁻¹-lemma : forall C {σ} -> C ⁻¹[ σ ] ≡ C ⁻¹ [ σ ]
⁻¹-lemma C = ⁻¹[]-lemma C ●

⁻¹∘-∘-assoc : forall C₁ C₂ C₃ -> C₁ ⁻¹∘ (C₂ ∘ C₃) ≡ (C₁ ⁻¹∘ C₂) ∘ C₃
⁻¹∘-∘-assoc ●            C₂ C₃ = ≡-refl
⁻¹∘-∘-assoc (const● σ)   C₂ C₃ = ≡-refl
⁻¹∘-∘-assoc (C₁ ●× σ)    C₂ C₃ = ⁻¹∘-∘-assoc C₁ (C₂ ●× σ) C₃
⁻¹∘-∘-assoc (σ ×● C₁)    C₂ C₃ = ⁻¹∘-∘-assoc C₁ (σ ×● C₂) C₃
⁻¹∘-∘-assoc (Thunk● n C₁) C₂ C₃ = ⁻¹∘-∘-assoc C₁ (Thunk● n C₂) C₃

⁻¹∘-lemma : forall C₁ C₂ -> C₁ ⁻¹∘ C₂ ≡ C₁ ⁻¹ ∘ C₂
⁻¹∘-lemma C₁ C₂ = ⁻¹∘-∘-assoc C₁ ● C₂
```

5.2 TypeTriple

```
------------------------------------------------------------------------
-- Types C [ Thunk n σ ] represented as triples ⟨ C , n , σ ⟩
------------------------------------------------------------------------

-- The contribution of this module is a data type enumerating all ways
-- in which C₁ [ Thunk n₁ σ₁ ] can be equal to C₂ [ Thunk n₂ σ₂ ].

module TypeTriple where

open import Kind
open import TypeSystem

open import Data.Nat using (ℕ)
open import Logic using (_≡_; ≡-refl)
open import Relation.Binary.PropositionalEquality using (≡-cong)

------------------------------------------------------------------------
-- Type triples

data TypeTriple : Set where
  ⟨_,_,_⟩ : PayCtxt -> ℕ -> Ty thunked -> TypeTriple

-- The semantics of a type triple.

⟦_⟧ : TypeTriple -> Ty thunked
⟦ ⟨ C , n , σ ⟩ ⟧ = C [ Thunk n σ ]

-- Equality between type triples.

infix 10 _≙_

data _≙_ : TypeTriple -> TypeTriple -> Set where
  ≙c-   :  forall {C n₁ σ₁ n₂ σ₂}
        -> ⟨ const● (C [ Thunk n₂ σ₂ ]) , n₁ , σ₁ ⟩ ≙ ⟨ C , n₂ , σ₂ ⟩
  ≙-c   :  forall {C n₁ σ₁ n₂ σ₂}
        -> ⟨ C , n₁ , σ₁ ⟩ ≙ ⟨ const● (C [ Thunk n₁ σ₁ ]) , n₂ , σ₂ ⟩

  ≙●    :  forall {n σ} -> ⟨ ● , n , σ ⟩ ≙ ⟨ ● , n , σ ⟩
  ≙●T   :  forall {C n₁ n₂ σ}
        -> ⟨ ● , n₁ , C [ Thunk n₂ σ ] ⟩ ≙ ⟨ Thunk● n₁ C , n₂ , σ ⟩
  ≙T●   :  forall {C n₁ n₂ σ}
        -> ⟨ Thunk● n₁ C , n₂ , σ ⟩ ≙ ⟨ ● , n₁ , C [ Thunk n₂ σ ] ⟩
  ≙TT   :  forall {C₁ C₂ n n₁ n₂ σ₁ σ₂}
        -> ⟨ C₁ , n₁ , σ₁ ⟩ ≙ ⟨ C₂ , n₂ , σ₂ ⟩
        -> ⟨ Thunk● n C₁ , n₁ , σ₁ ⟩ ≙ ⟨ Thunk● n C₂ , n₂ , σ₂ ⟩
```

```
  ≜●×²  :  forall {C₁ C₂ n₁ n₂ σ σ₁ σ₂}
        -> ⟨ C₁ , n₁ , σ₁ ⟩ ≜ ⟨ C₂ , n₂ , σ₂ ⟩
        -> ⟨ C₁ ●× σ , n₁ , σ₁ ⟩ ≜ ⟨ C₂ ●× σ , n₂ , σ₂ ⟩
  ≜×●²  :  forall {C₁ C₂ n₁ n₂ σ σ₁ σ₂}
        -> ⟨ C₁ , n₁ , σ₁ ⟩ ≜ ⟨ C₂ , n₂ , σ₂ ⟩
        -> ⟨ σ ×● C₁ , n₁ , σ₁ ⟩ ≜ ⟨ σ ×● C₂ , n₂ , σ₂ ⟩
  ≜●××● :  forall {C₁ C₂ n₁ n₂ σ₁ σ₂}
        -> ⟨ C₁ ●× C₂ [ Thunk n₂ σ₂ ] , n₁ , σ₁ ⟩ ≜
           ⟨ C₁ [ Thunk n₁ σ₁ ] ×● C₂ , n₂ , σ₂ ⟩
  ≜×●●× :  forall {C₁ C₂ n₁ n₂ σ₁ σ₂}
        -> ⟨ C₁ [ Thunk n₁ σ₁ ] ×● C₂ , n₂ , σ₂ ⟩ ≜
           ⟨ C₁ ●× C₂ [ Thunk n₂ σ₂ ] , n₁ , σ₁ ⟩

------------------------------------------------------------------------
-- The equality is meaningful

-- First some boring lemmas.

private

  drop-Thunk¹ :  forall {n₁ n₂ σ₁ σ₂}
              -> Thunk n₁ σ₁ ≡ Thunk n₂ σ₂ -> n₁ ≡ n₂
  drop-Thunk¹ ≡-refl = ≡-refl

  drop-Thunkʳ :  forall {n₁ n₂ σ₁ σ₂}
              -> Thunk n₁ σ₁ ≡ Thunk n₂ σ₂ -> σ₁ ≡ σ₂
  drop-Thunkʳ ≡-refl = ≡-refl

  drop-×¹ :  forall {k} {σ₁¹ σ₂¹ σ₁² σ₂² : Ty k}
          -> σ₁¹ × σ₂¹ ≡ σ₁² × σ₂² -> σ₁¹ ≡ σ₁²
  drop-×¹ ≡-refl = ≡-refl

  drop-×ʳ :  forall {k} {σ₁¹ σ₂¹ σ₁² σ₂² : Ty k}
          -> σ₁¹ × σ₂¹ ≡ σ₁² × σ₂² -> σ₂¹ ≡ σ₂²
  drop-×ʳ ≡-refl = ≡-refl

-- Soundness.

sound : forall {tt₁ tt₂} -> tt₁ ≜ tt₂ -> ⟦ tt₁ ⟧ ≡ ⟦ tt₂ ⟧
sound ≜c-               = ≡-refl
sound ≜-c               = ≡-refl
sound ≜●                = ≡-refl
sound ≜●T               = ≡-refl
sound ≜T●               = ≡-refl
sound (≜TT eq)          = ≡-cong (Thunk _) (sound eq)
sound (≜●×² {σ = σ} eq) = ≡-cong (\τ -> τ × σ) (sound eq)
sound (≜×●² {σ = σ} eq) = ≡-cong (\τ -> σ × τ) (sound eq)
sound ≜●××●             = ≡-refl
sound ≜×●●×             = ≡-refl
```

```
-- Completeness.

complete : forall tt₁ tt₂ -> ⟦ tt₁ ⟧ ≡ ⟦ tt₂ ⟧ -> tt₁ ≜ tt₂
complete ⟨ C₁ , _ , _ ⟩ ⟨ C₂ , _ , _ ⟩ eq = complete' C₁ C₂ eq
  where
  complete' :  forall C₁ {n₁ σ₁} C₂ {n₂ σ₂}
            -> C₁ [ Thunk n₁ σ₁ ] ≡ C₂ [ Thunk n₂ σ₂ ]
            -> ⟨ C₁ , n₁ , σ₁ ⟩ ≜ ⟨ C₂ , n₂ , σ₂ ⟩
  complete' (const● ._)    _              ≡-refl = ≜c-
  complete' _              (const● ._)    ≡-refl = ≜-c

  complete' ●              ●              ≡-refl = ≜●
  complete' ●              (Thunk● _ _)   ≡-refl = ≜●T
  complete' (Thunk● _ _)   ●              ≡-refl = ≜T●
  complete' (Thunk● n₁ C₁) (Thunk● n₂ C₂) eq with n₁ | drop-Thunk¹ eq
  ... | .n₂ | ≡-refl = ≜TT (complete' C₁ C₂ (drop-Thunkʳ eq))

  complete' (C₁ ●× σ₁) (C₂ ●× σ₂) eq with σ₁ | drop-×ʳ eq
  ... | .σ₂ | ≡-refl = ≜●×² (complete' C₁ C₂ (drop-×¹ eq))
  complete' (σ₁ ×● C₁) (σ₂ ×● C₂) eq with σ₁ | drop-×¹ eq
  ... | .σ₂ | ≡-refl = ≜×●² (complete' C₁ C₂ (drop-×ʳ eq))
  complete' (C₁ ●× ._) (._ ×● C₂) ≡-refl = ≜●××●
  complete' (._ ×● C₂) (C₁ ●× ._) ≡-refl = ≜×●●×

  complete' ●            (_ ●× _)     ()
  complete' ●            (_ ×● _)     ()
  complete' (Thunk● _ _) (_ ●× _)     ()
  complete' (Thunk● _ _) (_ ×● _)     ()
  complete' (_ ●× _)     ●            ()
  complete' (_ ●× _)     (Thunk● _ _) ()
  complete' (_ ×● _)     ●            ()
  complete' (_ ×● _)     (Thunk● _ _) ()
```

5.3 OperationalSemantics.GPay

```
------------------------------------------------------------------------
-- Calculating the result of generalised pay
------------------------------------------------------------------------

-- When gpay is executed in the thunked semantics the resulting value
-- has to be transformed to be of the right type, and the heap has to
-- be updated to ensure that the correct credit is recorded. In
-- nontrivial cases this is done by reducing the tick count of a
-- variable in the value (by modifying the context and heap; this
-- yields credit in the heap), and restoring the tick count of all
-- other uses of this variable by inserting gwait∋. This module
-- defines this operation.

module OperationalSemantics.GPay where

open import Kind
open import TypeSystem
open import Substitution
open VarSubst
open import Value
open import TypeTriple
open import OperationalSemantics.Binding
open Heap→
open import PayCtxt hiding (_∘_)

open import Data.Function using (id; _∘_)
open import Data.Nat using (ℕ; zero; _+_; _∸_; _⊓_)
open import Data.List using (_∷_; [])
open import Logic using (_≡_; ≡-refl)
open import Relation.Binary.PropositionalEquality
  using (≡-sym; ≡-trans; ≡-subst; ≡-cong)
open import Data.Nat.Properties using (m⊓n+n∸m≡n; i∸k∸j+j∸k≡i+j∸k)

------------------------------------------------------------------------
-- Lemmas used below

abstract

  ⁻¹_[]-lemma :  forall C₁ C₂ {σ}
              -> C₁ ⁻¹ [ C₂ [ σ ] ] ≡ C₁ ⁻¹∘ C₂ [ σ ]
  ⁻¹_[]-lemma C₁ C₂ = ≡-trans (≡-sym (⁻¹-lemma C₁)) (⁻¹[]-lemma C₁ C₂)

  ⁻¹-const●-lemma
    :  forall C C₂ {n₁ n₂ σ₁ σ₂}
    -> C ⁻¹∘ const● (C₂ [ Thunk n₂ σ₂ ]) [ Thunk n₁ σ₁ ] ≡
       C ⁻¹∘ C₂ [ Thunk n₂ σ₂ ]
  ⁻¹-const●-lemma C C₂ =
    ≡-trans (≡-sym (⁻¹[]-lemma C _)) (⁻¹[]-lemma C _)
```

```
  ×-lemma
    :  forall C C₁ C₂ {n₁ n₂ σ₁ σ₂}
    -> C ⁻¹○ (C₁ [ Thunk n₂ σ₁ ] ×● C₂) [ Thunk n₁ σ₂ ] ≡
       C ⁻¹○ (C₁ ●× C₂ [ Thunk n₁ σ₂ ]) [ Thunk n₂ σ₁ ]
  ×-lemma C C₁ C₂ =
    ≡-trans (≡-sym (⁻¹[]-lemma C _)) (⁻¹[]-lemma C _)

------------------------------------------------------------------------
-- Pay results

-- This type represents the result of evaluating an application of
-- gpay. It includes a new context, heap and value (or variable), plus
-- an accompanying substitution which takes other things to the new
-- context.

record GPay (_●_ : Ctxt thunked -> Ty thunked -> Set)
            (Γ : Ctxt thunked) (C : PayCtxt)
            (n m : ℕ) (σ : Ty thunked)
            : Set where
  Δ  : Ctxt thunked
  ρ  : Γ ⇒ Δ
  Σ₂ : Heap Δ
  v₂ : Δ ● C [ Thunk (n ∸ m) σ ]

open GPay

-- Updates the "value" field.

update-v₂
  :  forall {Γ} {_●¹_ C¹ n¹ m¹ σ¹}
                {_●²_ C² n² m² σ²}
  -> (r : GPay _●¹_ Γ C¹ n¹ m¹ σ¹)
  -> (   Δ r ●¹ C¹ [ Thunk (n¹ ∸ m¹) σ¹ ]
      -> Δ r ●² C² [ Thunk (n² ∸ m²) σ² ])
  -> GPay _●²_ Γ C² n² m² σ²
update-v₂ r f = record
  { Δ  = Δ r
  ; ρ  = ρ r
  ; Σ₂ = Σ₂ r
  ; v₂ = f (v₂ r)
  }

------------------------------------------------------------------------
-- Calculating the result of gpay: variable cases

mutual

  -- The vz case of gpayResult∋ is where things really happen. Most
  -- other cases merely propagate this new information.
```

```
-- See gpayResult below for an explanation of this function's
-- arguments.

gpayResult∋ :  forall C (m : ℕ) {Γ τ n σ}
            -> Heap Γ
            -> Γ ∋ τ -> τ ≡ C [ Thunk n σ ]
            -> GPay _∋_ Γ C n m σ
gpayResult∋ C₂ m Σ₁ (gwait∋ C₁ n x₁) eq =
  gpayResult∋-gwait • m n _ _ Σ₁ x₁
                    (complete ⟨ C₁ , _ , _ ⟩ ⟨ C₂ , _ , _ ⟩ eq)

gpayResult∋ C m .{Γ = Γ ▷ C [ Thunk n σ ]} {n = n} {σ = σ}
            (Σ₁ ▷ʰ b) (vz {Γ = Γ}) ≡-refl = record
    -- Reduce the tick count of this variable.
  { Δ  = Γ ▷ C [ Thunk (n ∸ m) σ ]
    -- The substitution ρ applies gwait∋ to occurrences of this
    -- variable.
  ; ρ  = wk ▷ cast (gwait∋ {n₁ = n ∸ m} C (m ⊓ n) vz)
    -- The binding in the heap is paid off.
  ; Σ₂ = Σ₁ ▷ʰ gpay↦ C m b
  ; v₂ = vz
  }
  where
  cast = ≡-subst (_∋_ _) (≡-cong (\i -> C [ Thunk i σ ])
                                 (m⊓n+n∸m≡n m n))

gpayResult∋ C m (Σ₁ ▷ʰ b) (vs {σ = σ} x₁) ≡-refl = record
  { Δ  = Δ r ▷ σ
  ; ρ  = ρ r ↑
    -- Note that the substitution is applied to all outer bindings.
  ; Σ₂ = Σ₂ r ▷ʰ b /↦ ρ r
  ; v₂ = vs (v₂ r)
  }
  where r = gpayResult∋ C m Σ₁ x₁ ≡-refl

-- This function takes care of propagating the information through
-- applications of gwait∋.

gpayResult∋-gwait
  :  forall C {C₁ C₂} (m n n₁ n₂ : ℕ) {Γ σ₁ σ₂} (Σ₁ : Heap Γ)
  -> (x₁ : Γ ∋ C ⁻¹∘ C₁ [ Thunk n₁ σ₁ ])
  -> (eq : ⟨ C₁ , n + n₁ , σ₁ ⟩ ≜ ⟨ C₂ , n₂ , σ₂ ⟩)
  -> GPay _∋_ Γ (C ⁻¹∘ C₂) n₂ m σ₂
gpayResult∋-gwait C m n n₁ n₂ Σ₁ x₁ (≜c- {C = C₂}) =
  update-v₂ (gpayResult∋ (C ⁻¹∘ C₂) m Σ₁ x₁ (⁻¹-const•-lemma C C₂))
            (gwait∋ (C ⁻¹∘ C₂) zero)
```

```
gpayResult∋-gwait C m n n₁ n₂ Σ₁ x₁ (≙-c {C = C₁} {σ₁ = σ₁}) =
  update-v₂ (gpayResult∋ (C ⁻¹∘ const● (C₁ [ Thunk n₁ σ₁ ]))
                         m {n = n₂} {σ = σ₁} Σ₁ x₁
                         (≡-sym (⁻¹-const●-lemma C C₁)))
            (cast₁ ∘ gwait∋ (C ⁻¹∘ C₁) n ∘ cast₂)
  where
  cast₁ = ≡-subst (_∋_ _) (≡-sym (⁻¹-const●-lemma C C₁))
  cast₂ = ≡-subst (_∋_ _) (⁻¹-const●-lemma C C₁)

gpayResult∋-gwait C m n n₁ .(n + n₁) Σ₁ x₁ (≙● {σ = σ}) =
  update-v₂ r (cast ∘ gwait∋ (C ⁻¹) (n ∸ (m ∸ n₁)))
  where
  r    = gpayResult∋ (C ⁻¹) m Σ₁ x₁ ≡-refl
  cast = ≡-subst (_∋_ _) (≡-cong (\i -> C ⁻¹ [ Thunk i σ ])
                                 (i∸k∸j+j∸k≡i+j∸k n n₁ m))

gpayResult∋-gwait C m n n₁ n₂ Σ₁ x₁ (≙●T {C = C'}) =
  update-v₂ (gpayResult∋ (C ⁻¹∘ Thunk● n₁ C') m Σ₁ x₁
                         (⁻¹_[]-lemma C (Thunk● n₁ C')))
            (cast₁ ∘ gwait∋ (C ⁻¹) n ∘ cast₂)
  where
  cast₁ = ≡-subst (_∋_ _) (⁻¹_[]-lemma C (Thunk● (n + n₁) C'))
  cast₂ = ≡-subst (_∋_ _) (≡-sym (⁻¹_[]-lemma C (Thunk● n₁ C')))

gpayResult∋-gwait C m n n₁ n₂ Σ₁ x₁ (≙T● {C = C'}) =
  update-v₂ (gpayResult∋ (C ⁻¹) m Σ₁ x₁
                         (≡-sym (⁻¹_[]-lemma C (Thunk● n₂ C'))))
            (cast₁ ∘ gwait∋ (C ⁻¹∘ Thunk● (n₂ ∸ m) C') n ∘ cast₂)
  where
  cast₁ = ≡-subst (_∋_ _) (≡-sym (⁻¹_[]-lemma C (Thunk● (n₂ ∸ m) C')))
  cast₂ = ≡-subst (_∋_ _) (⁻¹_[]-lemma C (Thunk● (n₂ ∸ m) C'))

gpayResult∋-gwait C m n n₁ n₂ Σ₁ x₁ (≙TT {n = n'} eq) =
  gpayResult∋-gwait (Thunk● n' C) m n n₁ n₂ Σ₁ x₁ eq

gpayResult∋-gwait C m n n₁ n₂ Σ₁ x₁ (≙●×² {σ = σ} eq) =
  gpayResult∋-gwait (C ●× σ) m n n₁ n₂ Σ₁ x₁ eq

gpayResult∋-gwait C m n n₁ n₂ Σ₁ x₁ (≙×●² {σ = σ} eq) =
  gpayResult∋-gwait (σ ×● C) m n n₁ n₂ Σ₁ x₁ eq

gpayResult∋-gwait C m n n₁ n₂ Σ₁ x₁
                  (≙●××● {C₁ = C₁} {C₂ = C₂} {σ₁ = σ₁} {σ₂ = σ₂}) =
  update-v₂ (gpayResult∋ (C ⁻¹∘ (C₁ [ Thunk n₁ σ₁ ] ×● C₂)) m Σ₁ x₁
                         (≡-sym (×-lemma C C₁ C₂)))
            (cast₁ ∘ gwait∋ (C ⁻¹∘ (C₁ ●× C₂ [ Thunk (n₂ ∸ m) σ₂ ])) n
                   ∘ cast₂)
  where
  cast₁ = ≡-subst (_∋_ _) (≡-sym (×-lemma C C₁ C₂))
  cast₂ = ≡-subst (_∋_ _) (×-lemma C C₁ C₂)
```

```
  gpayResult∋-gwait C m n n₁ n₂ Σ₁ x₁
                   (≙×●●× {C₁ = C₁} {C₂ = C₂} {σ₁ = σ₁} {σ₂ = σ₂}) =
    update-v₂ (gpayResult∋ (C ⁻¹∘ (C₁ ●× C₂ [ Thunk n₁ σ₂ ])) m Σ₁ x₁
                           (×-lemma C C₁ C₂))
              (cast₁ ∘ gwait∋ (C ⁻¹∘ (C₁ [ Thunk (n₂ ∸ m) σ₁ ] ×● C₂)) n
                     ∘ cast₂)
    where
    cast₁ = ≡-subst (_∋_ _) (×-lemma C C₁ C₂)
    cast₂ = ≡-subst (_∋_ _) (≡-sym (×-lemma C C₁ C₂))

------------------------------------------------------------------------
-- Calculating the result of gpay

-- This function calculates the result of evaluating gpay C m e in the
-- heap Σ, given the heap and value which are the result of evaluating
-- e in Σ. The functions above are used when a variable inside the
-- value is paid off.

gpayResult :  forall {Γ} C m (Σ₁ : Heap Γ) {n σ}
           -> Γ ⊢⇓ C [ Thunk n σ ]
           -> GPay _⊢⇓_ Γ C n m σ
gpayResult ● m Σ₁ (return⋆⇓ v₁) = record
  { Δ  = _
  ; ρ  = idˢ
  ; Σ₂ = Σ₁
  ; v₂ = return⋆⇓ v₁
  }

gpayResult (const● σ) m Σ₁ v₁ = record
  { Δ  = _
  ; ρ  = idˢ
  ; Σ₂ = Σ₁
  ; v₂ = v₁
  }

gpayResult (C ●× σ₂) m Σ₁ (x₁ ,⇓ y₁) =
  update-v₂ r (\x₂ -> (x₂ ,⇓ (y₁ /∋ ρ r)))
  where r = gpayResult∋ C m Σ₁ x₁ ≡-refl

gpayResult (σ₁ ×● C) m Σ₁ (x₁ ,⇓ y₁) =
  update-v₂ r (\y₂ -> ((x₁ /∋ ρ r) ,⇓ y₂))
  where r = gpayResult∋ C m Σ₁ y₁ ≡-refl

gpayResult (Thunk● k C) m Σ₁ (return⋆⇓ v₁) =
  update-v₂ (gpayResult C m Σ₁ v₁) return⋆⇓
```

5.4 OperationalSemantics.Thunked

```
------------------------------------------------------------------------
-- Annotated lazy operational semantics
------------------------------------------------------------------------

-- The thunked semantics is very similar to the simple one. The main
-- difference is that it also handles the library primitives.

-- Note that in the thunked case the substitution which is an index to
-- the semantics can, in addition to renaming variables, also reduce
-- the tick counts of heap elements (due to the presence of gwaitə).
-- This is part of the scheme used to give a semantics for gpay, as
-- explained in OperationalSemantics.GPay.

-- Note also that it is possible to merge the simple and thunked
-- operational semantics, just as the simple terms and the thunked,
-- run-time terms were merged into a single definition. This is not
-- done here, for two reasons:
--
-- * I tried, and some proofs became more complicated.
--
-- * More importantly, now the simplified correctness result can be
--   stated without referring to the thunked semantics (see
--   MainResults).

module OperationalSemantics.Thunked where

open import Kind
open import TypeSystem
open import Substitution
open VarSubst
open import Value
open import OperationalSemantics.Binding
open Heap→
open import OperationalSemantics.GPay

open import Data.Nat using (ℕ; zero; suc; _+_)

infix 10 _|_⇓⋉_|_✓_ _|_•⇓⋉_|_✓_

mutual

  data _|_⇓⋉_|_✓_
    :  forall {Γ Δ : Ctxt thunked} {τ}
    -> Heap Γ -> Γ ⊢ τ
    -> Γ ⇒ Δ -> Heap Δ -> Δ ⊢⇓ τ
    -> ℕ
    -> Set where
```

```
-- Values.

eval-λ
  :  forall {Γ σ n τ Σ} {e : Γ ▷ σ ⊢ Thunk n τ}
  -> Σ | λ e ⇊ id^s ∝ Σ | λ⇓ e ✓ zero

eval-λ²
  :  forall {Γ σ₁ σ₂ n τ Σ} {e : Γ ▷ σ₁ ▷ σ₂ ⊢ Thunk n τ}
  -> Σ | λ² e ⇊ id^s ∝ Σ | λ²⇓ e ✓ zero

eval-z
  :  forall {Γ} {Σ : Heap Γ}
  -> Σ | z ⇊ id^s ∝ Σ | z⇓ ✓ zero

eval-s
  :  forall {Γ} {Σ : Heap Γ}
  -> Σ | s ⇊ id^s ∝ Σ | s⇓ ✓ zero

eval-natrec
  :  forall {Γ σ n Σ} {e₁ : Γ ⊢ Thunk n σ} {e₂}
  -> Σ | natrec e₁ e₂ ⇊ wk ∝ Σ ▷^h ↦ e₁
                      | natrec⇓ vz (e₂ /⊢ wk ↑ ↑) ✓ zero

eval-,
  :  forall {Γ σ₁ σ₂ Σ} {e₁ : Γ ⊢ σ₁} {e₂ : Γ ⊢ σ₂}
  -> Σ | (e₁ , e₂) ⇊ wk ∘^s wk ∝ Σ ▷^h ↦ e₁ ▷^h ↦ e₂ /⊢ wk
                   | (vs vz ,⇓ vz) ✓ zero

eval-uncurry
  :  forall {Γ σ₁ σ₂ n τ Σ} {e : Γ ▷ σ₁ ▷ σ₂ ⊢ Thunk n τ}
  -> Σ | uncurry e ⇊ id^s ∝ Σ | uncurry⇓ e ✓ zero

-- Variables.

eval-vz
  :  forall {Γ Δ τ ms n ρ Σ₁ Σ₂}
            {e₁ : Γ ⊢ Thunks ms τ} {v : Δ ⊢⇓ Thunks ms τ}
  -> let v' = drop-return⋆⇓ ms v
         e₂ = valToTm v'
     in
     Σ₁                | e₁      ⇊ ρ    ∝ Σ₂          | v          ✓ n
  -> Σ₁ ▷^h ↦^n^s ms e₁ | var vz ⇊ ρ ↑ ∝ Σ₂ ▷^h ↦ e₂ | v' /⊢⇓ wk ✓ n

eval-vs
  :  forall {Γ Δ σ τ n ρ Σ₁ Σ₂}
            {x : Γ ∋ τ} {b : Γ ↦ σ} {v : Δ ⊢⇓ _}
  -> Σ₁       | var x      ⇊ ρ    ∝ Σ₂          | v          ✓ n
  -> Σ₁ ▷^h b | var (vs x) ⇊ ρ ↑ ∝ Σ₂ ▷^h b /↦ ρ | v /⊢⇓ wk ✓ n
```

```
eval-gwait∋
  :  forall {Γ Δ τ m₁ m₂ n Σ₁ Σ₂ C} {ρ : Γ ⇒ Δ}
            {x : Γ ∋ C [ Thunk m₁ τ ]} {v}
  -> Σ₁ | var x              ⇊ ρ ⋉ Σ₂ | v            ✓ n
  -> Σ₁ | var (gwait∋ C m₂ x) ⇊ ρ ⋉ Σ₂ | gwait⇓ C m₂ v ✓ n

-- Primitives.

eval-return
  :  forall {Γ Δ τ n Σ₁ Σ₂} {ρ : Γ ⇒ Δ} {e : Γ ⊢ τ} {v}
  -> Σ₁ | e        ⇊ ρ ⋉ Σ₂ | v          ✓ n
  -> Σ₁ | return e ⇊ ρ ⋉ Σ₂ | return⋆⇓ v ✓ n

eval-✓
  :  forall {Γ Δ τ m n Σ₁ Σ₂} {ρ : Γ ⇒ Δ} {e : Γ ⊢ Thunk m τ} {v}
  -> Σ₁ | e   ⇊ ρ ⋉ Σ₂ | return⋆⇓ v ✓ n
  -> Σ₁ | ✓ e ⇊ ρ ⋉ Σ₂ | return⋆⇓ v ✓ n

eval-pay
  :  forall {Γ Δ τ m₁ m₂ n Σ₁ Σ₂} {ρ : Γ ⇒ Δ}
            {e : Γ ⊢ Thunk m₁ τ} {v}
  -> Σ₁ | e        ⇊ ρ ⋉ Σ₂ | return⋆⇓ v            ✓ n
  -> Σ₁ | pay m₂ e ⇊ ρ ⋉ Σ₂ | return⋆⇓ (return⋆⇓ v) ✓ n

eval-gpay
  :  forall {Γ Δ τ m₁ m₂ n Σ₁ Σ₂ C} {ρ₁ : Γ ⇒ Δ}
            {e : Γ ⊢ C [ Thunk m₁ τ ]} {v}
  -> let r  = gpayResult C m₂ Σ₂ v
         ρ₂ = ρ₁ ∘ˢ GPay.ρ r
     in
     Σ₁ | e           ⇊ ρ₁ ⋉ Σ₂         | v                    ✓ n
  -> Σ₁ | gpay C m₂ e ⇊ ρ₂ ⋉ GPay.Σ₂ r | return⋆⇓ (GPay.v₂ r) ✓ n

eval->>=
  :  forall {Γ Δ X σ τ m₁ m₂ n₁ n₂ Σ₁ Σ₂ Σ₃} {ρ₁ : Γ ⇒ Δ} {ρ₂ : _ ⇒ X}
            {e₁ : Γ ⊢ Thunk m₁ σ} {e₂ : Γ ⊢ σ → Thunk m₂ τ} {v v₂}
  -> let ρ₃ = wk⇒ ρ₁ ∘ˢ ρ₂ in
     Σ₁                 | e₂        ⇊ ρ₁ ⋉ Σ₂ | v₂          ✓ n₁
  -> Σ₂ ▷ʰ ↦ⁿ e₁ /⊢ ρ₁ | v₂       •⇊ ρ₂ ⋉ Σ₃ | return⋆⇓ v ✓ n₂
  -> Σ₁                 | e₁ >>= e₂ ⇊ ρ₃ ⋉ Σ₃ | return⋆⇓ v ✓ (n₁ + n₂)

eval-gwait
  :  forall {Γ Δ τ m₁ m₂ n Σ₁ Σ₂ C} {ρ : Γ ⇒ Δ}
            {e : Γ ⊢ C [ Thunk m₁ τ ]} {v}
  -> Σ₁ | e           ⇊ ρ ⋉ Σ₂ | v            ✓ n
  -> Σ₁ | gwait C m₂ e ⇊ ρ ⋉ Σ₂ | gwait⇓ C m₂ v ✓ n
```

```
  -- Fixpoints.

  eval-fix
    :  forall {Γ Δ σ m n Σ₁ Σ₂} {ρ : _ ⇒ Δ}
              {e : Γ ▷ Thunk (suc m) σ ⊢ Thunk m σ} {v}
    -> Σ₁ ▷ʰ ↦ fix e | e     ⇓⇓ ρ       ⋉ Σ₂ | return⋆⇓ v ✓ n
    -> Σ₁            | fix e ⇓⇓ wk ∘ˢ ρ ⋉ Σ₂ | return⋆⇓ v ✓ suc n

  -- Applications.

  eval-·
    :  forall {Γ Δ X σ τ n₁ n₂ Σ₁ Σ₂ Σ₃}
              {ρ₁ : Γ ⇒ Δ} {ρ₂ : _ ⇒ X}
              {e₁ : Γ ⊢ σ → τ} {e₂ : Γ ⊢ σ} {v v₁}
    -> Σ₁                | e₁      ⇓⇓ ρ₁           ⋉ Σ₂ | v₁ ✓ n₁
    -> Σ₂ ▷ʰ ↦ e₂ /⊢ ρ₁ | v₁     •⇓⇓ ρ₂           ⋉ Σ₃ | v  ✓ n₂
    -> Σ₁                | e₁ · e₂ ⇓⇓ wk⇒ ρ₁ ∘ˢ ρ₂ ⋉ Σ₃ | v  ✓ (n₁ + n₂)

data _|_•⇓⇓_⋉_|_✓_
  :  forall {Γ Δ : Ctxt thunked} {σ τ}
  -> Heap (Γ ▷ σ) -> Γ ⊢⇓ σ → τ
  -> Γ ▷ σ ⇒ Δ -> Heap Δ -> Δ ⊢⇓ τ
  -> ℕ
  -> Set where

  eval-·-s
    :  forall {Γ} {Σ : Heap (Γ ▷ Nat)}
    -> Σ | s⇓ •⇓⇓ idˢ ⋉ Σ | s₁⇓ vz ✓ zero

  eval-·-λ
    :  forall {Γ Δ m n σ τ Σ₁ Σ₂} {ρ : _ ⇒ Δ}
              {e₁ : Γ ▷ σ ⊢ Thunk m τ} {v}
    -> Σ₁ | e₁     ⇓⇓ ρ ⋉ Σ₂ | return⋆⇓ v ✓ n
    -> Σ₁ | λ⇓ e₁ •⇓⇓ ρ ⋉ Σ₂ | return⋆⇓ v ✓ suc n

  eval-·-λ²
    :  forall {Γ m σ₁ σ₂ τ Σ}
              {e₁ : Γ ▷ σ₁ ▷ σ₂ ⊢ Thunk m τ}
    -> Σ | λ²⇓ e₁ •⇓⇓ idˢ ⋉ Σ | λ⇓ e₁ ✓ zero

  eval-·-natrec-zero
    :  forall {Γ₁ Γ₂ Γ₃ m n₁ n₂ τ ρ₁ Σ₁ Σ₂ Σ₃}
              {ρ₂ : Γ₂ ⇒ Γ₃} {x₁¹ : Γ₁ ∋ Thunk m τ} {e₁²} {v}
    -> let x₁¹' = var x₁¹ /⊢ wk ∘ˢ ρ₁
           ρ₃   = ρ₁ ∘ˢ ρ₂
       in
       Σ₁ | var vz          ⇓⇓ ρ₁ ⋉ Σ₂ | z⇓         ✓ n₁
    -> Σ₂ | x₁¹'            ⇓⇓ ρ₂ ⋉ Σ₃ | return⋆⇓ v ✓ n₂
    -> Σ₁ | natrec⇓ x₁¹ e₁² •⇓⇓ ρ₃ ⋉ Σ₃ | return⋆⇓ v ✓ suc (n₁ + n₂)
```

```
eval-·-natrec-suc
  :  forall {Γ₁ Γ₂ Γ₃ m n₁ n₂ τ ρ₁ ρ₂ Σ₁ Σ₂ Σ₃}
            {x₁¹ : Γ₁ ∋ Thunk m τ} {e₁²}
            {x₂ : Γ₂ ∋ Nat} {v : Γ₃ ⊢⇓ τ}
  -> let Σ₂'  = Σ₂ ▷ʰ ↦ (natrec (var x₁¹) e₁² /⊢ (wk ∘ˢ ρ₁)) · var x₂
         e₁²', = e₁² /⊢ (wk ∘ˢ ρ₁) ↑ ↑ /⊢ sub x₂ ↑
         ρ₃    = wk⇒ ρ₁ ∘ˢ ρ₂
     in
     Σ₁  | var vz           ⇊ ρ₁ ⋉ Σ₂ | s₁⇓ x₂      ✓ n₁
  -> Σ₂' | e₁²'             ⇊ ρ₂ ⋉ Σ₃ | return⋆⇓ v ✓ n₂
  -> Σ₁  | natrec⇓ x₁¹ e₁² •⇊ ρ₃ ⋉ Σ₃ | return⋆⇓ v ✓ suc (n₁ + n₂)

eval-·-uncurry
  :  forall {Γ₁ Γ₂ Γ₃ m n₁ n₂ σ₁ σ₂ τ ρ₁ ρ₂ Σ₁ Σ₂ Σ₃}
            {e₁ : Γ₁ ▷ σ₁ ▷ σ₂ ⊢ Thunk m τ}
            {x₂ : Γ₂ ∋ σ₁} {y₂ : Γ₂ ∋ σ₂} {v : Γ₃ ⊢⇓ τ}
  -> let e₁' = e₁ /⊢ (wk ∘ˢ ρ₁) ↑ ↑ /⊢ sub x₂ ↑ /⊢ sub y₂ in
     Σ₁ | var vz       ⇊ ρ₁       ⋉ Σ₂ | (x₂ ,⇓ y₂) ✓ n₁
  -> Σ₂ | e₁'          ⇊ ρ₂       ⋉ Σ₃ | return⋆⇓ v ✓ n₂
  -> Σ₁ | uncurry⇓ e₁ •⇊ ρ₁ ∘ˢ ρ₂ ⋉ Σ₃ | return⋆⇓ v ✓ suc (n₁ + n₂)
```

6 Correctness

This section states the correctness results. The proofs of these results—mostly uninteresting proofs by induction—are not included in this document.

6.1 MainResults

```
------------------------------------------------------------------------
-- Statements of the main results
------------------------------------------------------------------------

-- Some of these results are proved elsewhere (see the module Proofs).

module MainResults where

------------------------------------------------------------------------
-- The type system is well-defined, as are the simple and thunked
-- semantics

-- The following modules define the type system and the semantics,
-- which by construction are well-defined (well-typed etc.).

open import Kind
open import TypeSystem
open import Erasure
open import Thunked
open import Substitution
open import Value
import Heap
import Heap.Erasure
open import OperationalSemantics.Binding
open import OperationalSemantics.Simple
open import PayCtxt
open import TypeTriple
open import OperationalSemantics.GPay
open import OperationalSemantics.Thunked

------------------------------------------------------------------------
-- The simple semantics is functional

-- To state functionality we first need to define equality between the
-- results of derivations.

open VarSubst
open Heap→
open import Data.Nat using (ℕ)
open import Logic using (_≡_; _≅_)
```

```
data _=⇓=_
  {Γ : _} {τ : _} {Σ₁ : Heap Γ} {t : Γ ⊢ τ}
  {Δ¹ : _} {ρ¹ : Γ ⇒ Δ¹} {Σ₂¹ : Heap Δ¹} {v¹ : Δ¹ ⊢⇓ τ} {n¹ : ℕ}
  {Δ² : _} {ρ² : Γ ⇒ Δ²} {Σ₂² : Heap Δ²} {v² : Δ² ⊢⇓ τ} {n² : ℕ}
  (d¹ : Σ₁ ∣ t ⇓ ρ¹ ∝ Σ₂¹ ∣ v¹ ✓ n¹)
  (d² : Σ₁ ∣ t ⇓ ρ² ∝ Σ₂² ∣ v² ✓ n²)
  : Set where
  results-= : Δ¹ ≅ Δ² -> ρ¹ ≅ ρ² -> Σ₂¹ ≅ Σ₂² -> v¹ ≅ v² -> d¹ =⇓= d²

-- Now functionality can be stated.

Functionality : Set
Functionality =  forall {Γ : Ctxt simple} {τ Σ₁} {t : Γ ⊢ τ}
                        {Δ¹} {ρ¹ : Γ ⇒ Δ¹} {Σ₂¹ v¹ n¹}
                        {Δ²} {ρ² : Γ ⇒ Δ²} {Σ₂² v² n²}
             -> (d¹ : Σ₁ ∣ t ⇓ ρ¹ ∝ Σ₂¹ ∣ v¹ ✓ n¹)
             -> (d² : Σ₁ ∣ t ⇓ ρ² ∝ Σ₂² ∣ v² ✓ n²)
             -> d¹ =⇓= d²

-------------------------------------------------------------------------
-- The thunked semantics is sound and complete with respect to the
-- simple one

open Heap→Erasure
open import Relation.Binary.PropositionalEquality using (≡-subst)

Soundness : Set
Soundness =  forall {Γ Δ τ Σ₁ ρ Σ₂ n} {e : Γ ⊢ τ} {v : Δ ⊢⇓ τ}
          -> Σ₁ ∣ e ⇊ ρ ∝ Σ₂ ∣ v ✓ n
          -> ⌜ Σ₁ ⌝ʰ ∣ ⌜ e ⌝ ⇓ ⌜ ρ ⌝↦ ∝ ⌜ Σ₂ ⌝ʰ ∣ ⌜ v ⌝⇓ ✓ n

-- The type ∃Thunked Σ₁ e ρ Σ₂ v n, if inhabited, says that Σ₁ ∣ e
-- evaluates in n steps (in the thunked semantics) to something which
-- erases to Σ₂ ∣ v. (The statement is actually a bit stronger, since
-- the resulting substitution also has to be related to ρ.)

data ∃Thunked {Γ : Ctxt thunked} {Δ : Ctxt simple} {τ : Ty thunked}
              (Σ₁ : Heap Γ) (e : Γ ⊢ τ)
              (ρ : ⌜ Γ ⌝▷ ⇒ Δ) (Σ₂ : Heap Δ) (v : Δ ⊢⇓ ⌜ τ ⌝⋆) (n : ℕ)
              : Set where
  ∃thunked :  (Δ' : Ctxt thunked)
           -> (Δ-eq : Δ ≡ ⌜ Δ' ⌝▷)
           -> (ρ' : Γ ⇒ Δ')
           -> ≡-subst (_⇒_ _) Δ-eq ρ ≡ ⌜ ρ' ⌝↦
           -> (Σ₂' : Heap Δ')
           -> ≡-subst Heap Δ-eq Σ₂ ≡ ⌜ Σ₂' ⌝ʰ
           -> (v' : Δ' ⊢⇓ τ)
           -> ≡-subst (\Δ -> Δ ⊢⇓ ⌜ τ ⌝⋆) Δ-eq v ≡ ⌜ v' ⌝⇓
           -> Σ₁ ∣ e ⇊ ρ' ∝ Σ₂' ∣ v' ✓ n
           -> ∃Thunked Σ₁ e ρ Σ₂ v n
```

```
Completeness : Set
Completeness =  forall Γ {Δ τ} Σ₁ {ρ Σ₂ n} (e : Γ ⊢ τ) {v : Δ ⊢⇓ ⌜ τ ⌝⋆}
             -> ⌜ Σ₁ ⌝ʰ | ⌜ e ⌝ ⇓ ρ ∝ Σ₂ | v ✓ n
             -> ∃Thunked Σ₁ e ρ Σ₂ v n

------------------------------------------------------------------------
-- The time bounds asserted by the type system are correct

open import Data.Nat using (zero; _+_; _≤_)
open import Data.List using (sum)

-- A term of type Thunk n₁ (Thunk n₂ (Thunk n₃ Nat)) takes at most
-- n₁ + n₂ + n₃ steps amortised time to evaluate to WHNF. The time
-- function calculates this number, given a type.

time : Ty thunked -> ℕ
time (Thunk n τ) = n + time τ
time _           = zero

-- Heap bindings carry credit, corresponding to evaluation steps which
-- have already been paid off, but not yet performed.

credit↦ : forall {Γ : Ctxt thunked} {τ} -> Γ ↦ τ -> ℕ
credit↦ (↦ⁿˢ ns e) = sum ns

-- A heap inherits the credit of its bindings.

credit : forall {Γ : Ctxt thunked} -> Heap Γ -> ℕ
credit ∅ʰ       = zero
credit (Σ ▷ʰ b) = credit↦ b + credit Σ

-- Correctness statement.

Correctness : Set
Correctness =  forall {Γ Δ τ ρ Σ₁ Σ₂ n} {e : Γ ⊢ τ} {v : Δ ⊢⇓ τ}
            -> Σ₁ | e ⇊ ρ ∝ Σ₂ | v ✓ n
            -> n + credit Σ₂ ≤ time τ + credit Σ₁

-- The statement above can be simplified. The simplified variant does
-- not make use of the thunked semantics, so it is easier to
-- understand. However, it is also less modular: no credit is attached
-- to Σ₂, so instances of this result cannot (easily) be chained
-- together.

Correctness' : Set
Correctness' =  forall {Γ Δ τ ρ} (Σ₁ : Heap Γ) {Σ₂ n}
                       (e : Γ ⊢ τ) {v : Δ ⊢⇓ ⌜ τ ⌝⋆}
             -> ⌜ Σ₁ ⌝ʰ | ⌜ e ⌝ ⇓ ρ ∝ Σ₂ | v ✓ n
             -> n ≤ time τ + credit Σ₁
```

References

Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. Accepted for publication in POPL '08: Conference record of the 35th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2008.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

The Agda Team. The Agda Wiki. Available at http://www.cs.chalmers.se/~ulfn/Agda/, 2007.