

Keep Singing the Gospel

Tiago Soares

January 29, 2026

Introduction

Gospel

Gospel is a tool agnostic specification language for OCaml.

Provides a lightweight Separation Logic syntax

```
val add : int → t → t
(*@ s2 = add n s1
    ensures s2 = s1 ∪ { : n : } *)
```

The Gospel Ecosystem

The tools that interface with Gospel specifications are:

- Ortac: Runtime Assertion Checking
- Cameleer: Automated Deductive Verification using Why3
- **Peter**: Interactive verification using Iris or CFML.

Some work by Gros and Pereira has also been done in translating Gospel to Viper.

New features

Notable new features:

- Intermediate Separation Logic representation.
- Stricter separation between OCaml and Gospel values.
- Multiple relations between OCaml and logical values.

Outline

1. Two OCaml modules and their Gospel specifications
2. Translation of Gospel into Separation Logic
3. Multiple relations between OCaml and Gospel values.
4. Demo of a Iris proof for a Gospel specification.

Bit Set Example

Bit sets

```
type bset
```

```
val add : int → bset → bset
```

```
val mem : int → bset → bool
```

```
val empty : bset
```

Specification - Model Annotation

```
(*@ open Set *)
```

```
type bset
```

```
(*@ model : integer set *)
```

A `model` annotation declares a relation between an OCaml type and a logical (Gospel) type.

In Gospel, we refer to relations as *lenses*.

Specification - add

```
val add : int → bset → bset
(*@ s2 = add n s1
    ensures s2 = s1 ∪ { : n : } *)
```

s1 and s2 have type integer set in the post condition.

The type of n is integer, the model for `int`.

Variables in logical assertions have logical types.

Specification - mem & empty

```
val mem : int → bset → bool
```

```
(*@ b = mem n s
```

```
    ensures b ↔ n ∈ s *)
```

```
val empty : bset
```

```
(*@ ensures empty = ∅ *)
```

Priority Queues

Priority Queues

```
type priority = int
```

```
type elt = char
```

```
type queue
```

```
val create : unit → queue
```

```
val add : queue → elt → priority → unit
```

```
val extract : queue → elt option
```

This example is monomorphic to so that the semantics of the specifications are easier to present.

Type specification queue

```
(*@ open Bag *)
```

```
type queue
```

```
(*@ mutable model : (character * integer) bag *)
```

The type bag is a built-in Gospel type for multisets.

The **mutable** keyword is used to state that the logical representation of a priority queue may change over time.

Specification - add function

```
val add : queue → elt → priority → unit
(*@ add q x i
    modifies q
    ensures q = Bag.add (x, i) (old q) *)
```

The **modifies** clause states that this function may modify the contents of `q`.

The post condition uses **old** to refer to the logical representation of `q` before the function is called.

Translation into Separation Logic

How Do We Use These Specifications?

Gospel's semantics are defined with a translation into GospelSL.

GospelSL is designed to be translated into multiple Separation Logics.

Bit Sets - Model Annotations

```
type bset  
(*@ model : integer set *)
```

Type bset

GospelSL

Persistent Lens Bset :

bset $\rightarrow$ integer set $\rightarrow$ slProp

Priority Queues - Model Annotations

```
type queue  
(*@ mutable model : (character * integer) bag *)
```

Type queue

GospelSL

Lens Queue :

queue $\rightarrow$ (character * integer) bag $\rightarrow$ slProp

Priority Queues - add Specification

```
val add : queue → elt → priority → unit
(*@ add q x i
    modifies q
    ensures q = Bag.add (x, i) (old q) *)
```

GospelSL

```
Value add : queue → elt → priority → unit
```

```
Triple add_spec :
```

```
∀ val_q q val_x x val_i i
```

```
{ Queue val_q q ★ Char val_x x ★ Int val_i i }
```

```
add val_q val_x val_i
```

```
{ ∃ q', Queue val_q q' ★  $\neg$  q' = Bag.add (x, i) q  $\neg$  }15
```

Priority Queues - add Specification

val add : queue → elt → priority → unit

(*@ add q x i

modifies q

ensures q = Bag.add (x, i) (old q) *)

GospelSL

Value add : queue → elt → priority → unit

Triple add_spec :

$\forall$ val_q q val_x x val_i i

{ Queue val_q q * Char val_x x * Int val_i i }

add val_q val_x val_i

{ $\exists$ q', Queue val_q q' * $\neg$ q' = Bag.add (x, i) q $\neg$ }₁₅

Priority Queues - add Specification

```
val add : queue → elt → priority → unit
(*@ add q x i
    modifies q
    ensures q = Bag.add (x, i) (old q) *)
```

GospelSL

```
Value add : queue → elt → priority → unit
```

```
Triple add_spec :
```

```
∀ val_q q val_x x val_i i
```

```
{ Queue val_q q ★ Char val_x x ★ Int val_i i }
```

```
add val_q val_x val_i
```

```
{ ∃ q', Queue val_q q' ★  $\neg$  q' = Bag.add (x, i) q  $\neg$  }15
```

Priority Queues - add Specification

```
val add : queue → elt → priority → unit
(*@ add q x i
    modifies q
    ensures q = Bag.add (x, i) (old q) *)
```

GospelSL

```
Value add : queue → elt → priority → unit
```

```
Triple add_spec :
```

```
∀ val_q q val_x x val_i i
```

```
{ Queue val_q q ★ Char val_x x ★ Int val_i i }
```

```
add val_q val_x val_i
```

```
{ ∃ q', Queue val_q q' ★  $\neg$  q' = Bag.add (x, i) q  $\neg$  }15
```

Priority Queues - add Specification

```
val add : queue → elt → priority → unit
(*@ add q x i
    modifies q
    ensures q = Bag.add (x, i) (old q) *)
```

GospelSL

```
Value add : queue → elt → priority → unit
```

```
Triple add_spec :
```

```
∀ val_q q val_x x val_i i
```

```
{ Queue val_q q ★ Char val_x x ★ Int val_i i }
```

```
add val_q val_x val_i
```

```
{ ∃ q', Queue val_q q' ★  $\neg$  q' = Bag.add (x, i) q  $\neg$  }_{15}
```

Priority Queues - add Specification

```
val add : queue → elt → priority → unit
(*@ add q x i
    modifies q
    ensures q = Bag.add (x, i) (old q) *)
```

GospelSL

```
Value add : queue → elt → priority → unit
Triple add_spec :
  ∀ val_q q val_x x val_i i
  { Queue val_q q ★ Char val_x x ★ Int val_i i }
  add val_q val_x val_i
  { ∃ q', Queue val_q q' ★  $\neg$  q' = Bag.add (x, i) q  $\neg$  }15
```

Priority Queues - add Specification

```
val add : queue → elt → priority → unit
(*@ add q x i
    modifies q
    ensures q = Bag.add (x, i) (old q) *)
```

GospelSL

```
Value add : queue → elt → priority → unit
Triple add_spec :
  ∀ val_q q val_x x val_i i
  { Queue val_q q * Char val_x x * Int val_i i }
  add val_q val_x val_i
  { ∃ q', Queue val_q q' * q' = Bag.add (x, i) q }15
```

Multiple Lens

Motivation

So far, we only allow one way to relate OCaml values to their models.

Limitation: users only have one way to describe ownership of OCaml values.

Example - Physical Equality

```
val ph_eq : queue → queue → bool
(*@ b = ph_eq q1 q2
   preserves q1
   preserves q2
   ensures b  $\leftrightarrow$  q1 = q2 *)
```

Problems with this specification:

- The comparison $q1 = q2$ compares the models, not the addresses.
- The specification requires the two queues to be disjoint.

Translation into Separation Logic

Value `ph_eq` : `t` $\rightarrow$ `t` $\rightarrow$ `bool`

GospelSL

Triple `ph_eq_spec` :

```
   $\forall$  val_q1 q1 val_q2 q2  
  { Queue val_q1 q1  $\star$  Queue val_q2 q2 }  
    ph_eq val_q1 q2  
  { ... }
```

Precondition requires separation of `q1` and `q2`.

```
(*@ persistent lens Qloc : queue  $\triangleright$  loc *)
```

`Qloc` is a lens that relates values of type `queue` to values of type `loc`.

`loc` is a built-in type for memory addresses.

Physical Equality w/ Lenses

```
val ph_eq : queue → queue → bool
  (*@ b = ph_eq q1 q2
    preserves q1 @ Qloc
    preserves q2 @ Qloc
    ensures b ↔ q1 = q2 *)
```

```
Triple ph_eq_spec :
  ∀ val_q1 q1 val_q2 q2
  { Qloc val_q1 q1 ★ Qloc val_q2 q2 }
  ph_eq val_q1 q2
  { ... }
```

GospelSL

Models and Lenses

`model` declarations are just sugar for `lens` declarations

```
type queue  
(*@ mutable model : (character * integer) bag *)
```

Desugars into

```
type queue  
  
(*@ lens Queue : queue ▷ (character * integer) bag *)
```

Implicit lenses

The specification for add:

```
val add : queue → elt → priority → unit
(*@ add q x i
    modifies q
    ensures q = add (x, i) (old q) *)
```

Is desugared into:

```
val add : queue → elt → priority → unit
(*@ add q x i
    modifies q @ Queue
    ensures q = add (x, i) (old q) *)
```

Demo

Future Work

Higher order lenses.

More extensive set of examples.

Adding additional Separation Logic backends.

Thank you

Specifications for Polymorphic Functions

Polymorphic Priority Queue

```
type 'a queue
(*@ mutable model : ('a * integer) bag *)

val add : 'a queue → 'a → priority → unit
(*@ add q x i
    consumes x @ 'A
    modifies q @ 'A Queue
    ensures q = Bag.add (x, i) (old q) *)
```

Separation Logic Translation

Lens Queue : $\forall 'a 'A.$

GospelSL

$'a \text{ queue} \rightarrow ('a \rightarrow 'A \rightarrow \text{slProp}) \rightarrow$
 $('A * \text{integer}) \text{ bag} \rightarrow \text{slProp}$

Value add : $'a \text{ queue} \rightarrow 'a \rightarrow \text{priority} \rightarrow \text{unit}$

Triple add_spec :

$\forall A \text{ val_q } q \text{ val_x } x \text{ val_i } i$

$\{ \text{Queue val_q } q * A \text{ val_x } x * \text{Int val_i } i \}$

$\text{add val_q val_x val_i}$

$\{ \exists q', \text{Queue val_q } q' * \neg q' = \text{Bag.add } (x, i) \text{ } q \neg \}$