Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica Magistrale

Meta-programmazione di binding OCaml per GTK3

Relatore:
Chiar.mo Prof.
Claudio Sacerdoti Coen

Presentata da:
Alberto Nicoletti

Sessione III
Anno Accademico 2018/19

Abstract

GTK è una delle principali librerie per costruire interfacce grafiche, ed essendo scritta in C sono molti i linguaggi che realizzano dei binding a questa libreria. GTK dispone di un sistema di introspezione in grado di generare un file XML che descrive l'intera API della libreria. Tramite questo meccanismo per numerosi linguaggi sono state realizzate delle librerie in grado di leggere questo file XML e generare automaticamente i binding per GTK. Nel caso di OCaml invece esiste la libreria *lablgtk*, la quale è però scritta a mano: questo permette di avere una API più vicina ai costrutti idiomatici del linguaggio, ma incompleta, sensibile ai cambi di versione e prona ad errori. L'obiettivo raggiunto in questa tesi è la realizzazione di un prototipo funzionante in grado di generare automaticamente i binding di GTK per OCaml. Il prototipo è stato ottenuto modificando *haskell-gi*, la libreria analoga per Haskell, un linguaggio simile ad OCaml che ha dovuto affrontare problematiche simili. In particolare questa libreria è composta di due moduli principali, uno per la lettura del file XML, ed uno per la generazione del codice Haskell a partire dalle informazioni ottenute dal parsing. In *ocaml-gi-gtk*, la libreria creata in questa tesi, è stato riusato completamente il modulo di parsing, ma è stato riscritto il modulo di generazione del codice per generare codice OCaml invece che Haskell.

Indice

Listings

Capitolo 1

Introduzione

La sviluppo di sistemi per la creazione di interfacce grafiche è un processo complicato che richiede ingenti sforzi, soprattutto se le applicazioni create attraverso questi sistemi devono essere indipendenti dalla piattaforma e quindi potute eseguire su più sistemi operativi. Per questo motivo la maggior parte delle applicazioni non implementano da zero i meccanismi per la creazione di interfacce grafiche, ma usano librerie che ne facilitano la creazione mettendo a disposizione widget e funzionalità. Tra le più popolari di queste librerie, particolarmente in ambiente Linux/Unix, vi è GTK.

GTK è una libreria scritta nel linguaggio C facendo uso di GObject, un sistema di tipi orientato agli oggetti per C. Una particolarità delle librerie scritte in C è che esse sono facilmente usabili anche da altri linguaggi di programmazione, i quali spesso mettono a disposizione del programmatore dei meccanismi per interfacciarsi con il codice C. Il processo di "allacciamento" tra codice di due diversi linguaggi di programmazione viene chiamato binding. GTK si presta particolarmente alla creazione di binding poiché, facendo uso di GObject, dispone di GObject Introspection, un meccanismo di introspezione in grado di generare un file GIR, un formato XML che descrive l'intera API della libreria.

Tramite l'uso di questo file GIR, sono numerosi i linguaggi che dispongono di librerie in grado di generare automaticamente i binding per GTK. OCaml, il linguaggio di programmazione principalmente trattato in questa tesi, invece non dispone di una tale libreria. È però presente lablgtk, una libreria che implementa i binding per GTK ma at-

traverso una scrittura manuale. Questa libreria mette a disposizione del programmatore una API di alto livello che ben si sposa con i costrutti idiomatici di OCaml. Purtroppo il costo per la manutenzione manuale dei binding per una libreria vasta come GTK è molto elevato ed infatti non tutta la API di GTK viene coperta. Inoltre, GTK è soggetta a dei frequenti cambi di versione, che si ripercuotono in necessarie modifiche a lablgtk.

L'obiettivo fissato da questa tesi è quindi lo sviluppo di una libreria in grado di generare automaticamente i binding OCaml per GTK. Prima di iniziare lo sviluppo sono state analizzate alcune librerie equivalenti per altri linguaggi di programmazione. Tra queste, è stata notata in particolare haskell-gi, quella per il linguaggio Haskell. Questa libreria è suddivisa in due moduli principali: uno per effettuare il parsing del file GIR ed uno per la generazione del codice Haskell a partire dalle informazioni raccolte dal primo modulo. Il parsing del file GIR è un compito tedioso e di poco interesse, per questo motivo è stato scelto di procedere con lo sviluppo effettuando un fork di haskell-gi, riusando completamente il modulo di parsing ma riscrivendo il modulo di generazione del codice in modo da emettere in output codice OCaml.

Nello riscrivere il modulo di generazione del codice, si è cercato di riusare il più possibile i meccanismi di binding di lablgtk in modo da generare codice OCaml che fosse il più idiomatico possibile ed anche per facilitare la conversione di applicazioni che attualmente fanno uso di lablgtk.

Al termine dello sviluppo è stato ottenuto un prototipo in grado di generare l'84% dell'API di GTK, a confronto del 45% coperto da lablgtk. È stato inoltre effettuato il porting di numerosi file di esempio di lablgtk, riscontrando una buona compatilibilità tra le due librerie.

Il resto della tesi sarà suddiviso nei seguenti capitoli:

- Nel capitolo 2 verranno esplorati nel dettaglio i principali linguaggi di programmazione, meccanismi e librerie usate nello sviluppo di questa tesi;
- Nel capitolo 3 verrà fatta una descrizione ad alto livello di ocaml-gi-gtk, la libreria sviluppata in questa tesi, spiegando come è stata impostata e come poterla eseguire;
- Nel capitolo 4 si scenderà nei dettagli dell'implementazione di ocaml-gi-gtk spiegando come è stata gestita la generazione di codice OCaml per ogni tipo di dato

di GTK che è stato gestito;

- Nel capitolo 5 si farà una valutazione del lavoro svolto, evidenziandone i principali limiti e descrivendo possibili lavori futuri.

Capitolo 2

Background e strumenti usati

Lo sviluppo di una libreria per la generazione automatica di binding tra due linguaggi comporta l'impiego di diverse tecnologie. Tra queste troviamo innanzitutto i linguaggi protagonisti della conversione, in questo caso C e OCaml. Laddove il primo è un linguaggio popolare e spesso insegnato nei corsi di laurea in informatica, il secondo è di nicchia e ne verrà fatta un'introduzione in questo capitolo. Inoltre verrà introdotta GTK, la libreria per cui verranno generati i binding. Infine sarà trattata lablgtk, l'attuale libreria che implementa i binding di GTK per OCaml, e le librerie di generazione automatica per altri linguaggi, in particolare Haskell.

2.1 GTK

GTK [10] [22] è uno dei più popolari toolkit open source per creare interfacce grafiche. Nasce nel 1996 durante lo sviluppo di GIMP (GNU Image Manupulation Program) [5] ma diventa successivamente una libreria general-purpose. Gli elementi più interessanti di GTK sono i widget, che sono dei componenti grafici (ad esempio bottoni, campi di testo, menu, etc.) che possono essere composti per creare l'interfaccia grafica di un'applicazione. Alcuni di questi widget vengono infatti chiamati Container e possono contenere altri widget. GTK fa in realtà uso di diverse librerie di più basso livello:

- **Pango**, gestisce il rendering dei font;
- **Cairo**, una libreria di grafica 2D;

- **GDK e GdkPixbuf** (GIMP Drawing Kit), gestiscono il caricamento di immagini in vari formati;
- **ATK** (Accessibility Toolkit), gestisce l'accessibilità dell'applicazione;

Tutte queste librerie dipendono da GLib, la libreria di più basso livello in questa gerarchia.

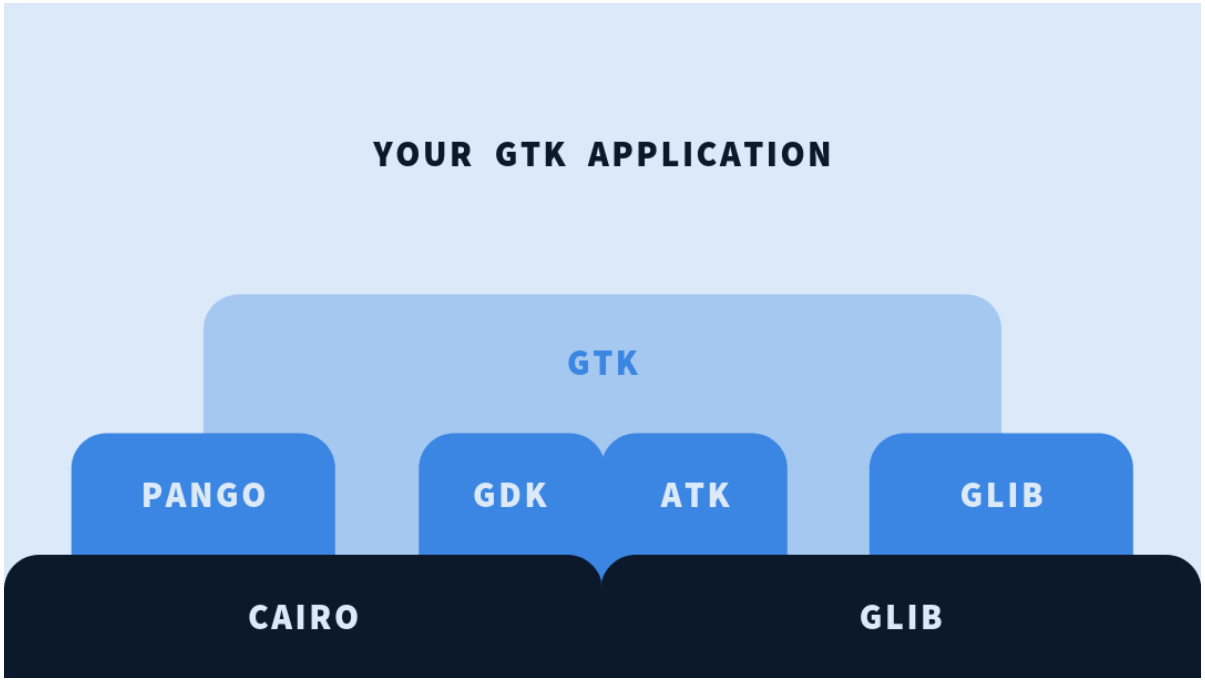

Figura 2.1: Architettura di un'applicazione GTK [11].

2.1.1 GLib

GLib definisce delle strutture dati pensate per essere usate insieme a due altre librerie che fanno parte di GLib stessa, ovvero GIO e GObject. La prima fornisce al programmatore una API di alto livello per effettuare operazioni di input/output, ad esempio operazioni su file o operazioni di rete, ma quella di più interesse per questa tesi è GObject. Questa libreria fornisce degli strumenti per scrivere codice C orientato

agli oggetti. Questo comprende meccanismi di classi, interfacce, ereditarietà ed anche un garbage collector basato su reference counting. Fornisce inoltre il supporto per una programmazione basata su eventi, fondamentale nel contesto delle interfacce grafiche. In particolare, GObject mette a disposizione delle macro per creare nuovi tipi di dato che ereditano da dei tipi di base definiti dalla libreria, ad esempio GObject, GInterface e GBoxed (rispettivamente per definire oggetti, interfacce o un tipo opaco). Le nuove classi ottenute possono essere usate come superclasse da nuove classi, ottenendo quindi una gerarchia di oggetti [8].

```
#ifndef _my_app_window_h_
#define _my_app_window_h_

G_BEGIN_DECLS

#define MY_APP_TYPE_WINDOW (my_app_window_get_type ())
G_DECLARE_FINAL_TYPE(MyAppWindow, my_app_window, MY_APP, WINDOW,
   GtkApplicationWindow)

MyAppWindow *          my_app_window_new            (void);

G_END_DECLS

#endif /* _my_app_window_h_ */
```

Listing 2.1: Esempio di creazione di un nuovo tipo GObject (.h) [21]

```
#include "my-app-window.h"

struct _MyAppWindow
{
  GtkApplicationWindow parent_instance;

  // instance variables for subclass go here
};

G_DEFINE_TYPE(MyAppWindow, my_app_window, GTK_TYPE_APPLICATION_WINDOW)

static void
```

```
13 my_app_window_init (MyAppWindow *window)
14 {
15   // initialisation goes here
16 }
17
18 static void
19 my_app_window_class_init (MyAppWindowClass *class)
20 {
21   // virtual function overrides go here
22   // property and signal definitions go here
23 }
24
25 MyAppWindow *
26 my_app_window_new (void)
27 {
28   return g_object_new (MY_APP_TYPE_WINDOW, NULL);
29 }
```

Listing 2.2: Esempio di creazione di un nuovo tipo GObject (.c) [21]

Tutte le librerie basate su GObject hanno anche la caratteristica di poter esportare la propria API in un file XML attraverso un meccanismo chiamato GObject Introspection.

2.1.2 GObject Introspection

GObject Introspection [9] è un meccanismo in grado di generare dei file GIR (un formato XML) contenenti la descrizione completa dell'API di una libreria basata su GObject. Tuttavia alcune informazioni, come per esempio l'opzionalità di un parametro, non sono ottenibili dal solo sistema di GObject e devono essere integrate tramite l'uso di annotazioni in GTK-Doc (un formato particolare per commenti in C). Le possibili annotazioni possono essere trovate sulla wiki ufficiale [4]. Questi file GIR sono particolarmente utili per creare dei binding a queste librerie poiché è molto comune tra i linguaggi di programmazione avere la possibilità di interfacciarsi con codice C, ed avendo a disposizione le informazioni relative ai tipi delle funzioni per cui vengono creati i binding, questi possono essere generati automaticamente.

2.2 OCaml

OCaml [19] è un linguaggio di programmazione general purpose principalmente funzionale, ma con supporto anche al paradigma imperativo e ad oggetti. OCaml pone particolare enfasi sulla correttezza del codice, implementando un sistema di tipi statico accompagnato da un sistema di type inference Hindley-Milner. La type inference permette al programmatore di non dover quasi mai annotare esplicitamente le variabili con i relativi tipi, ma questi vengono automaticamente dedotti dal compilatore.

Nel seguito verranno introdotte delle feature caratteristiche di OCaml che sono state usate in questa tesi e che non sono comuni nei linguaggi di programmazioni più popolari.

2.2.1 Moduli

I moduli nascono dall'esigenza di isolare in un unico punto un tipo di dato e le operazioni da usare su di esso. In OCaml ogni file definisce implicitamente un modulo che racchiude i dati e le operazioni definite all'interno del file. Gli altri file possono accedere ai tipi di dato ed alle operazioni contenute nel modulo tramite dot notation. Ad esempio per accedere alla funzione `map` del modulo `List` della libreria standard verrà usata la notazione `List.map`. Alternativamente, tramite la keyword `open ModuleName` è possibile "aprire" un file per portare nello scope corrente tutti gli identificatori definiti dentro il modulo aperto. È inoltre possibile definire moduli interni tramite le keyword `module ModuleName = struct ... end`.

2.2.2 Algebraic Data Types (ADT)

I tipi di dato algebrici sono un costrutto di programmazione introdotto dalla famiglia di linguaggi funzionale ML, di cui OCaml fa parte. Nel paradigma funzionale sono il costrutto principale usato per descrivere il dominio delle applicazioni. Gli ADT si dividono in due tipi, prodotti (tuple e record) e coprodotti (varianti).

Tuple e Record

Le tuple sono un tipo di dato formato da una sequenza ordinata e finita di valori. Questo tipo di dato è usato per raggruppare più valori all'interno della stessa variabile. Ad esempio una tupla formata da 3 elementi `(4, "hello", 3.14)` ha tipo `(int * string * float)`. Da notare che nel tipo è usata la notazione `*` per denotare un prodotto tra i tipi. Questo deriva dalla teoria degli insiemi, infatti l'insieme dei valori assumibili dalla tupla è il prodotto cartesiano degli insiemi formati dagli elementi dei singoli tipi.

I record hanno la stessa funzione delle tuple, ovvero quella di trasportare più valori in una singola variabile, ma permettono di associare un nome ad ogni componente della tupla. Al contrario delle tuple, il tipo del record deve essere dichiarato prima di poter definire una variabile di questo tipo.

```
1  type point = {
2      x : float;
3      y : float;
4  }
5
6  let is_origin p = match p with
7   | {x=0.0; y=0.0} -> true
8   | _              -> false
9  (* val is_origin : point -> bool = <fun> *)
10
11 let distance_between p1 p2 =
12     sqrt ((p2.x -. p1.x) +. (p2.y -. p1.y))
13 (* val distance_between : point -> point -> float = <fun> *)
```

Listing 2.3: Esempio di record in OCaml

Varianti

Chiamate anche coprodotti o unioni taggate, le varianti sono un costrutto di OCaml usato per definire un tipo di dato che può assumere un numero limitato di valori. Assomigliano alle enum presenti in altri linguaggi di programmazione, ma differiscono poichè i valori assumibili non sono interi, ma vengono chiamati costruttori. Questi possono contenere anche altri tipi di dato e possono avere una definizione ricorsiva.

```
type binary_tree =
  | Leaf
  | Node of int * binary_tree * binary_tree

let rec size tree = match tree with
  | Leaf          -> 0
  | Node (_,l,r) -> 1 + size l + size r
(* val size : binary_tree -> int = <fun> *)

let my_tree =
  Node (0,
    Node (1, Leaf, Leaf),
    Leaf
  )
(* val my_tree : binary_tree = ... *)

size my_tree;;
(* - : int = 2 *)
```

Listing 2.4: Esempio di variante in OCaml

2.2.3 Polimorfismo

Nel contesto dei linguaggi di programmazione il poliformismo è un meccanismo per definire delle procedure che tramite una singola definizione possano operare su più tipi di dato. OCaml fa uso di due tipi di polimorfismo: parametrico e row. Il polimorfismo ad hoc invece non è supportato da OCaml poichè renderebbe indecidibile l'algoritmo di type inference, ma è presente eccezionalmente in alcuni operatori della libreria standard come quello di uguaglianza `(=)`.

Polimorfismo parametrico

Il polimorfismo parametrico, presente anche in altri linguaggi come Java (tramite generics) e C++ (tramite template), permette di definire delle funzioni che possono operare su più tipi di dato. Al contrario di questi linguaggi, dove la variabile di tipo deve esse-

re esplicitamente introdotta, il compilatore di OCaml è in grado di determinare il tipo più generico che una funzione può assumere. Le variabili di tipo vengono rappresentate come atomi preceduti dal simbolo apostrofo ' (es: `'a` o `'b`) e vengono lette nel linguaggio naturale come lettere greche. Queste variabili vengono quantificate universalmente all'esterno della funzione ed istanziate nel tipo concreto una volta che la funzione viene applicata ai suoi argomenti.

```
let id x = x
(* val id : 'a -> 'a = <fun> *)

let compose f g x = f (g x)
(* val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun> *)

let sum_then_multiply = compose (fun x -> x * 4) (fun x -> x + 1)
(* val sum_then_multiply : int -> int = <fun> *)

type 'a binary_tree =
  | Leaf
  | Node of 'a * binary_tree * binary_tree

let rec size tree = match tree with
  | Leaf          -> 0
  | Node (_,l,r) -> 1 + size l + size r
(* val size : 'a binary_tree -> int = <fun> *)
```

Listing 2.5: Esempi di polimorfismo parametrico.

Polimorfismo row

OCaml supporta inoltre il polimorfismo row, che permette di realizzare la tecnica conosciuta come duck typing nei linguaggi dinamici in un sistema di tipi statico. Questo typing prende il nome di structural typing e permette di confrontare due variabili in base alle loro proprietà e non in base al nome del loro tipo (nominal typing). Il polimorfismo row in OCaml è realizzato attraverso una combinazione di polimorfismo parametrico e variabili row. Procediamo nella spiegazione tramite esempi:

```
let f x = print_endline (x#say_hello ()) ;;
```

```
(* val f : < say_hello : unit -> string; .. > -> unit = <fun> *)
```

f è una funzione che prende in input una variabile x, e nel corpo della funzione chiama un metodo `say_hello` su tale variabile. Il compilatore inferisce che x abbia quindi un tipo che includa il metodo `say_hello` ed utilizza zucchero sintattico tramite `..` per indicare che x può contenere anche altri campi. Il tipo completo di f sarebbe `val f : 'a. < say_hello : unit -> string; 'a > -> unit = <fun>` dove `'a` è una variabile row.

```
let donald = object
    method say_hello () = "Hi, i'm Donald!"
end
(* val donald : < say_hello : unit -> string > = <obj> *)

let goofy = object
    method say_hello () = "Hi, i'm Goofy!"
    method something_else () = ()
end
(* val goofy : < say_hello : unit -> string; something_else : unit ->
   unit > = <obj> *)

f donald;;
(* Prints: Hi, i'm Donald! *)

f goofy;;
(* Prints: Hi, i'm Goofy! *)
```

Entrambi gli oggetti `donald` e `goofy` vengono accettati da f, dove il primo unifica `'a` con il tipo vuoto ed il secondo unifica `'a` con `something_else : unit -> unit`. Il polimorfismo row introduce anche un concetto di subtyping strutturale in OCaml. Nell'esempio precedente il tipo di `goofy` è sottotipo del tipo di `donald`, in quanto il primo può essere usato in tutti i contesti nei quali può essere usato il secondo. Il cast tra tipi è possibile ma deve essere sempre esplicito tramite l'operatore `(:>)` ed è possibile effettuare solo cast da una sottoclasse verso una superclasse.

```
let f (x: < say_hello : unit -> string >) =
    print_endline (x#say_hello ())

f donald;;
```

```
(* Prints: Hi, i'm Donald! *)

f goofy;;
(* Error: This expression has type
    < say_hello : unit -> string; something_else : unit -> unit >
   but an expression was expected of type < say_hello : unit -> string
   >
   The second object type has no method something_else *)

f (goofy :> < say_hello : unit -> string >);;
(* Prints: Hi, i'm Goofy! *)
```

Listing 2.6: Esempio di upcast esplicito

2.2.4 Varianti Polimorfe

Le varianti polimorfe [3] sono un costrutto di programmazione esclusivo di OCaml e sono essenzialmente delle varianti che fanno uso di polimorfismo row. I costruttori delle varianti polimorfe devono essere preceduti dal simbolo ‘ e non devono essere necessariamente dichiarati come parte di un tipo prima di farne uso. Ad esempio:

```
let animal_say_hi = function
  | `DOG -> "woof"
  | `CAT -> "meow"
(* val animal_say_hi : [< `CAT | `DOG ] -> string = <fun> *)

let animal_say_hi_2 = function
  | `DOG -> "woof"
  | `CAT -> "meow"
  | _    -> "..."
(* val animal_say_hi_2 : [> `CAT | `DOG ] -> string = <fun> *)
```

L'input delle due funzioni viene inferito rispettivamente come `[< ‘CAT | ‘DOG ]` e `[> ‘CAT | ‘DOG ]`, dove `|` assume lo stesso significato che ha nelle varianti, ovvero quello di enumerare i possibili costruttori. `<` e `>` invece sono zucchero sintattico per una variabile row che sia rispettivamente un sottotipo o un supertipo dei costruttori elencati.

2.2.5 Garbage collector

OCaml ha una gestione automatica della memoria attraverso l'uso di un garbage collector generazionale. Il garbage collector sfrutta infatti l'ipotesi generazionale: secondo questa, la maggior parte delle variabili vengono allocate per poco tempo prima di essere deallocate, ma quelle che rimangono in memoria tendono a rimanerci per molto tempo. OCaml divide la memoria in due heap, uno chiamato *minor* in cui le variabili vengono inizialmente allocate, ed uno chiamato *major* in cui vengono spostate le variabili che sopravvivono dopo una scansione del GC.

2.3 Meccanismi di interfacciamento tra C e OCaml

OCaml dispone di meccanismi di foreign function interface con il codice C. Questi permettono di far interagire del codice OCaml con del codice C e risultano fondamentali per effettuare dei binding OCaml per una libreria C. I metodi di interfacciamento tra OCaml e C sono descritti in un capitolo del manuale di OCaml [16] e vengono qui riassunti e riportati in quanto fondamentali per questa tesi.

2.3.1 Binding a delle primitive

I binding vengono effettuati esclusivamente tra funzioni di OCaml e funzioni di C. Le funzioni C vengono chiamate primitive. I binding per le primitive aventi $n <= 5$ parametri sono implementate da funzioni C che ricevono in input n parametri di tipo `value` e restituiscono un valore di tipo `value`. Il tipo value è una rappresentazione per i valori di OCaml. Per esempio il binding ad una primitiva C che somma due interi può essere definita come:

```
1 CAMLprim value
2 sum(value x, value y)
3 {
4 ...
5 }
```

Listing 2.7: Esempio di binding OCaml

I binding alle primitive con arietà $n > 5$ devono essere invece implementati da due funzioni C:

- una avente 2 parametri (`value *argv, int argn`) usata da `ocamlc`, il compliatore bytecode;
- una avente tutti gli n parametri usata da `ocamlopt`, il compilatore nativo.

```
CAMLprim value
sum_bytecode(value *argv, int argn)
{
  ...
}

CAMLprim value
sum_native(value v1, value v2, value v3, value v4, value v5, ...)
{
  ...
}
```

Listing 2.8: Binding per funzioni aventi $n > 5$ parametri

Una volta definite in C queste funzioni, che come vedremo servono a realizzare la conversione tra i tipi OCaml e C, è possibile richiamarle da OCaml attraverso una dichiarazione *external*:

```
external sum : int -> int -> int -> int -> int -> int -> int = "
   sum_bytecode" "sum_native"
```

Listing 2.9: Esempio di dichiarazione external

In questo esempio è stata creata una funzione OCaml `sum` avente il tipo annotato. La annotazione è necessaria ed occorre prestare attenzione nell'usare i tipi corretti. La funzione `sum` realizza quindi un binding con le funzioni C `sum_bytecode` e `sum_native`. Nel caso in cui si effettui un binding per una funzione avente $n <= 5$ parametri va assegnata la sola funzione C definita.

Implementare una primitiva consiste in due compiti separati:

1. decodificare i valori OCaml in valori C, e codificare i valore di return come un valore OCaml;

2. calcolare effettivamente la funzione.

La funzione che effettua la conversione viene chiamata *stub*.

```
CAMLprim value
sum_stub(value v1, value v2)
{
  int x, y, res;
  x = Int_val(x);
  y = Int_val(y);
  res = sum(x, y);
  return Val_int(res);
}

int sum(int x, int y) {
  return x + y;
}
```

Listing 2.10: Binding completo di una funzione C

2.3.2 Il tipo value

Tutti i valori di OCaml sono rappresentati in C con il tipo `value`, definito nel file `caml/mlvalues.h`. Questo tipo può indicare:

- un integer unboxed;
- un puntatore ad un blocco dentro lo heap di OCaml;
- un puntatore ad un oggetto fuori dallo heap di OCaml (ad esempio allocato in C tramite `malloc`).

Integer

I valori integer sono a 63-bit e sono unboxed.

Blocchi nello heap

I blocchi dentro lo heap sono rappresentati tramite una struttura contenente un header, che indica la grandezza in word del blocco, ed un tag, una enumerazione che descrive il contenuto del blocco. Il tag può assumere i seguenti valori:

1. Da `0` a `No_scan_tag - 1`: un array di oggetti OCaml, dove ogni oggetto è un `value`;
2. `Closure_tag`: una closure. La prima word è un puntatore al codice, le restanti word contengono l'ambiente;
3. `String_tag`: una stringa o una sequenza di byte;
4. `Double_tag`: un double;
5. `Double_array_tag`: un array o un record di numeri double;
6. `Abstract_tag`: un tipo di dato astratto;
7. `Custom_tag`: un tipo di dato astratto con funzioni di finalizzazione, comparazione, hashing, serializzazione e deserializzazione definte dall'utente.

Blocchi fuori dallo heap

Ogni puntatore word-aligned che punta ad un indirizzo fuori dallo heap può essere castato in modo safe al (e dal) tipo `value`. Bisogna fare attenzione a non usare deallocare questi blocchi di memoria usando `free` dopo che un puntatore a questi è stato restituito ad OCaml tramite un cast a `value`. OCaml infatti non sarà a conoscenza della deallocazione e incorrerà in errori a runtime.

2.3.3 Rappresentazione di tipi di dato OCaml

I tipi di dato atomici di OCaml vengono rappresentati nel tipo `value` tramite:

- **int**, integer unboxed;
- **char**, codice ASCII rappresentato da un integer unboxed;

- **float**, blocco `Double_tag`;
- **bytes**, blocco `String_tag`;
- **string**, blocco `String_tag`;
- **int32**, blocco `Custom_tag`;
- **int64**, blocco `Custom_tag`;
- **nativeint**, blocco `Custom_tag`;

Tuple e record

Le tuple ed i record sono rappresentati da puntatori a blocchi con tag 0. I campi dei record sono nello stesso ordine della dichiarazione del tipo. I record i cui campi hanno tutti il tipo statico `float` vengono rappresentati come array di `float`, con tag `Double_array_tag`. Inoltre i record che contengono un solo campo vengono chiamati *unboxable* e possono essere rappresentati in due modi:

- **boxed**, un blocco con tag 0 o `Double_array_tag`;
- **unboxed**, usando direttamente il valore del campo.

La scelta viene fatta, in ordine decrescente di priorità, tramite:

- la presenza di un attributo `[@@boxed]` o `[@@unboxed]` nella dichiarazione di tipo;
- un'opzione del compilatore (`-unboxed-types` o `-no-unboxed-types`);
- la rappresentazione di default, che nella versione corrente di OCaml e' *boxed*.

Array

Gli array di float hanno una rappresentazione unboxed, e sono rappresentati tramite puntatori a blocchi con tag `Double_array_tag`. Gli array di interi e di altri tipi di dato sono rappresentati come tuple, ovvero come puntatori a blocchi con tag 0.

Varianti

Le varianti sono rappresentate da integer unboxed per i costruttori costanti, altrimenti da blocchi il cui tag è una numerazione dei costruttori non-costanti. L'enumerazione inizia da 0 ed è separata per i costruttori costanti e non-costanti. Un costruttore non-costante avente n parametri è rappresentato da un blocco di grandezza n.

Oggetti

Gli oggetti sono rappresentati da blocchi con tag `Object_tag`. Il primo campo del blocco fa riferimento alla classe dell'oggetto (ed ai suoi metodi). Il secondo campo contiene un ID usato per la comparazione. I campi rimanenti contengono i valori delle variabili dell'oggetto.

Varianti polimorfe

Le varianti polimorfe differiscono dalle varianti dal fatto che i costruttori non vengono numerati partendo da 0, ma vengono identificati da un hash (un intero OCaml), calcolato dalla funzione C `hash_variant` definita nel file `<caml/mlvalues.h>`. A differenza delle varianti, i parametri del costruttore non vengono espansi. Ad esempio, `'Costr(v, w)` è rappresentato da un blocco di grandezza 2, dove il primo campo contiene la rappresentazione della coppia `(v, w)`, laddove una variante sarebbe rappresentata con un blocco di grandezza 3, contenente `v` e `w` rispettivamente nel secondo e terzo campo.

2.3.4 Cooperare con il garbage collector

I blocchi nello heap non più usati vengono deallocati dal garbage collector, per questo motivo bisogna mettere un atto qualche accorgimento nelle funzioni C che operano su questi valori:

1. Una funzione che ha parametri o variabili locali di tipo `value` deve iniziare con una chiamata ad una delle macro `CAMLparam` o `CAMLlocal`, e terminare con una chiamata a `CAMLreturn`. Più nello specifico le macro disponibili sono:

- `CAMLparam0 ... CAMLparam5`, da usare all'inizio della funzione, passando il numero adeguato di parametri di tipo `value`;
- `CAMLxparam ... CAMLxparam5`, da usare se la funzione ha più di 5 parametri di tipo `value`, usare questa macro con il numero di parametri restanti;
- `CAMLlocal1 ... CAMLlocal5`, possono essere usate solo dopo una chiamata a `CAMLparam` per dichiarare una variabile locale;
- `CAMLlocalN (x, n)`, per dichiarare un array x contenente n valori di tipo `value`;
- `CAMLreturn0`, se la funzione ha tipo `void`, bisogna inserire una chiamata a questa macro al posto di `return`, anche quando esso è implicito;
- `CAMLreturn`, se il valore v da restituire ha tipo `value`, usare `CAMLreturn (v)`;
- `CAMLreturnT`, se il valore v da restituire ha tipo t, usare `CAMLreturnT (t, v)`.

2. Gli assegnamenti ai campi di blocchi strutturati devono essere fatti tramite la macro `Store_field` (per i blocchi normali), o `Store_double_field` (per gli array di float e per i record contenenti solo float). Una chiamata a `Store_field (b, n, v)` assegna il valore v nel campo numero n del blocco b.

3. Le variabili globali contenenti valori di tipo `value` devono essere registrate con il garbage collector usando la funzione `caml_register_global_root`. Una variabile globale v può essere registrata tramite `caml_register_global_root (&v)` e rimossa tramite `caml_remove_global_root(&v)`.

2.3.5 Callback

È possibile chiamare funzioni OCaml (che vengono chiamate closure) da codice C. Per farlo si usano le seguenti funzioni:

- `caml_callback(f, a)` applica la funzione f al valore a e restituisce il valore restituito da f;
- `caml_callback2(f, a, b)` (e `caml_callback3(f, a, b, c)`) applicano la funzione f ai valori a e b (a, b e c rispettivamente);

- `caml_callbackN(f, n, args)` applica la funzione f agli n parametri contenuti nell'array `args`.

2.4 lablgtk

lablgtk [15] è la libreria che viene attualmente usata per creare applicazioni GTK usando OCaml. Nasce come libreria di binding scritti a mano per la versione 1 di GTK e recentemente è stata aggiornata alla versione 3. Questa libreria è la principale fonte di ispirazione di questa tesi riguardo a come impostare i binding per GTK.

2.4.1 Binding alle classi di GTK

È particolarmente interessante come questa libreria rappresenti nel type system di OCaml le gerarchie di classi di GTK. I widget ed in generale tutti gli oggetti di GTK vengono istanziati da C e risiedono nello heap di C. Gli autori di questa libreria hanno deciso di rappresentare i valori restituiti dal costruttore di una classe tramite un tipo opaco. Nel file `src/gobject.ml` viene quindi dichiarato un tipo `type -'a obj`, dove `obj` è quindi un costruttore di tipo polimorfo. Il simbolo `-` serve per dichiarare che `'a obj` è controvariante. Questo significa che se `'a` è sottotipo di `'b`, allora `'a obj` è supertipo di `'b obj`. Questa relazione di subtyping viene sfruttata per rappresentare correttamente le gerarchie di oggetti di GTK facendo uso di varianti polimorfe per istanziare il parametro di `obj`.

```
1 type widget        = [`giu      | `widget]
2 type container     = [widget    | `container]
3 type bin           = [container | `bin]
4 type button        = [bin       | `button]
5 type toggle_button = [button    | `togglebutton]
```

Listing 2.11: Esempi di varianti polimorfe per rappresentare la gerarchia di oggetti di GTK.

Il tipo delle varianti polimorfe viene definito secondo la gerarchia di GTK. In questo esempio `widget` è sottotipo di `button` (in GTK la relazione è invece al contrario), ma

viene sfruttata la controvarianza di `obj` per invertire la relazione. Pertanto `button obj` è sottotipo di `widget obj` come in GTK.

Un'altra particolarità interessante di lablgtk3 è come gli autori hanno deciso di strutturare i binding agli oggetti. Ogni oggetto è definito in tre diversi file:

- in **gtk*.ml** sono presenti i binding di "basso livello". Qui sono definiti i binding alle proprietà, segnali e metodi. Non viene fatto uso delle funzionalità ad oggetti di OCaml;
- in **ml_gtk*.c** vengono definite le funzioni di binding vere e proprie usate dal file gtk*.ml;
- in **g*.ml** sono dichiarate le classi OCaml che fanno uso dei pre-metodi definiti nel file gtk*.ml. Queste classi hanno un parametro di tipo `obj` usato per rendere disponibile ai metodi l'oggetto a cui fanno riferimento.

```
1  module Button = struct
2    let cast w : Gtk.button obj = try_cast w "GtkButton"
3    module P = struct
4      let label : ([>`button],_) property = PrivateProps.label
5      ...
6      let xalign : ([>`button],_) property = {name="xalign"; conv=float}
7    end
8    module S = struct
9      open GtkSignal
10     let clicked = {name="clicked"; classe=`button; marshaller=
       marshal_unit}
11     ...
12     let released = {name="released"; classe=`button; marshaller=
       marshal_unit}
13   end
14   let create pl : Gtk.button obj = Object.make "GtkButton" pl
15   let make_params ~cont pl ?label ?use_stock ?use_underline ?relief =
16     let pl = (
17       may_cons P.label label (
18       may_cons P.use_stock use_stock (
19       may_cons P.use_underline use_underline (
```

```
      may_cons P.relief relief pl)))) in
    cont pl
end
```

Listing 2.12: Estratto di file di basso livello in lablgtk (gtkButtonProps.ml)

```
class virtual button_props = object
  val virtual obj : _ obj
  method label = get Button.P.label obj
  method set_label = set Button.P.label obj
  ...
  method xalign = get Button.P.xalign obj
  method set_xalign = set Button.P.xalign obj
end

class button_skel obj = object (self)
  inherit bin obj
  inherit button_props
end

class button_signals obj = object
  inherit container_signals_impl (obj : [> button] obj)
  inherit button_sigs
end

class button obj = object
  inherit button_skel (obj : Gtk.button obj)
  method connect = new button_signals obj
end

let button ?label =
  Button.make_params [] ?label ~cont:(
  pack_return (fun p -> new button (Button.create p)))
```

Listing 2.13: Estratto di file di alto livello in lablgtk (gButton.ml e ogtkButtonProps.ml)

2.4.2 Limiti di lablgtk

Il principale problema di lablgtk è che i binding sono scritti manualmente. Questo comporta innanzitutto un enorme sforzo da parte degli autori per coprire tutta la API di GTK, la quale infatti è coperta solo in parte dai binding di lablgtk. Un'ulteriore conseguenza della scrittura manuale dei binding è che possono facilmente essere commessi errori. Va inoltre considerato che GTK dipende da altre librerie basate su GObject le cui funzionalità sono attualmente coperte in minima parte lablgtk. Esistono inoltre librerie basate su GTK che implementano widget aggiuntivi (es: GtkSourceView). Anche per queste librerie i binding sono attualmente scritti a mano riportando i medesimi limiti, superabili tramite uno strumento di generazione automatica dei binding.

2.5 haskell-gi

L'obiettivo di questa tesi è l'implementazione di una libreria che sia in grado generare automaticamente dei binding di GTK per OCaml, e librerie analoghe sono già esistenti per altri linguaggi. Tra le più popolari troviamo *PyGObject* per Python [18], *gjs* per Javascript [6], *gtk-rs* [12] per Rust e *haskell-gi* [14] per Haskell. Quest'ultima è particolarmente interessante poiché Haskell ha molte caratteristiche in comune con OCaml. Entrambi sono linguaggi funzionali con un sistema di tipi simile ed entrambi condividono la presenza di un garbage collector. Per questo motivo haskell-gi è stata analizzata a fondo per individuare idee o parti di codice che potessero essere riutilizzate al fine di guidare l'implementazione di una libreria analoga per OCaml. Inoltre haskell-gi è in grado di generare dei binding Haskell non solo per GTK, ma per ogni libreria che supporti la GObject Introspection. Tra queste troviamo anche tutte le librerie dalle quali GTK dipende.

haskell-gi ha tre moduli principali:

- **base**, una libreria a sè stante, implementa le operazioni di più basso livello per effettuare le conversioni tra i tipi di C (e GLib) e quelli Haskell. Il codice generato dal modulo di CodeGen fa uso di questa libreria;

- **GIR**, si occupa di effettuare il parsing dei file GIR e di esporre al modulo di CodeGen le informazioni necessarie per la generazione del codice;
- **CodeGen**, tramite le informazioni ottenute dal modulo GIR, genera una libreria Haskell contenente i binding per la libreria C che si sta generando.

Di queste, è risultato di particolare interesse il modulo GIR. Le informazioni esposte da questo modulo sono infatti utilizzabili anche per generare codice OCaml. Gli altri due moduli sono invece troppo specifici e legati alla generazione di codice Haskell, che ha un meccanismo di interfacciamento con C completamente differente da quello di OCaml [13]. Tuttavia, il modulo CodeGen è comunque di interesse per quanto riguarda l'architettura e la strutturazione della libreria, che può essere riusata modificando le parti specifiche di generazione del codice per generare codice OCaml.

La scelta che è stata effettuata è quindi quella di effettuare un fork di haskell-gi, mantenendo intatto il modulo GIR, sostituendo il modulo base con un equivalente per OCaml, e modificando ampiamente il modulo CodeGen adattandolo per OCaml. *ocaml-gi-gtk*, la libreria sviluppata in questa tesi, è quindi scritta in Haskell, scelta atipica considerando che si deve inserire nell'ecosistema di OCaml. Questa scelta presenta infatti qualche controindicazione, come la difficoltà di attrarre futuri contributori alla libreria, che provenendo dall'ecosistema di OCaml potrebbero non conoscere Haskell ed i suoi tool. A fronte di questa barriera di entrata, troviamo però il beneficio di condividere delle parti di codice con una libreria ben consolidata e mantenuta, permettendo così una prototipazione rapida di una libreria equivalente per OCaml. Nel caso in cui ocaml-gi-gtk sia ben ricevuta dalla community sarà sempre possibile effettuarne una conversione in OCaml.

Segue ora una descrizione di alcune feature di haskell-gi che sono state riusate in questa tesi e che verranno citate nelle sezioni seguenti.

2.5.1 Type

Ad ogni variabile presente in GTK è associato il relativo tipo. Nel file GIR questa informazione è presente e denotata dal tag `<type name c:type />`. Il parser di haskell-gi gestisce questa informazione definendo i tipi BasicType, Type e API tramite degli ADT.

BasicType viene usato per rappresentare i tipi di dato base di GLib, ovvero i tipi che trovano una diretta corrispondenza con un tipo di dato base di C, ma che sono stati ridefiniti per questioni di portabilità e di facilità d'uso [7].

```
data BasicType = TBoolean          -- ^ gboolean
               | TInt              -- ^ gint
               | TUInt             -- ^ guint
               | TLong             -- ^ glong
               | TULong            -- ^ gulong
               | TInt8             -- ^ gint8
               | TUInt8            -- ^ guint8
               | TInt16            -- ^ gint16
               | TUInt16           -- ^ guint16
               | TInt32            -- ^ gint32
               | TUInt32           -- ^ guint32
               | TInt64            -- ^ gint64
               | TUInt64           -- ^ guint64
               | TFloat            -- ^ gfloat
               | TDouble           -- ^ gdouble
               | TUniChar          -- ^ gunichar
               | TGType            -- ^ GType
               | TUTF8             -- ^ gchar*, encoded as UTF-8
               | TFileName         -- ^ gchar*, encoding a filename
               | TPtr              -- ^ gpointer
               | TIntPtr           -- ^ gintptr
               | TUIntPtr          -- ^ guintptr
                 deriving (Eq, Show, Ord)
```

Listing 2.14: Tipo BasicType di haskell-gi

Il tipo `Type` invece contiene i `BasicType` e tutti gli altri tipi non di base di GLib. Un costruttore particolare di `Type` è `TInterface`. Questo è usato per tutti i restanti tipi che non hanno trovato una definizione dai costruttori precedenti. Questi sono tipi complessi come ad esempio classi, interfacce, struct ed enum. Per ottenere il tipo specifico è necessario usare la funzione `findAPIByName` definita nel file Code.hs. Questa restituisce il tipo API, che contiene dei costruttori dai quali è possibile estrarre ulteriori informazioni relative al tipo specifico.

```haskell
data Type
    = TBasicType BasicType
    | TError           -- ^ GError
    | TVariant         -- ^ GVariant
    | TParamSpec       -- ^ GParamSpec
    | TCArray Bool Int Int Type  -- ^ Zero terminated, Array Fixed
                                 -- Size, Array Length, Element Type
    | TGArray Type     -- ^ GArray
    | TPtrArray Type   -- ^ GPtrArray
    | TByteArray       -- ^ GByteArray
    | TGList Type      -- ^ GList
    | TGSList Type     -- ^ GSList
    | TGHash Type Type -- ^ GHashTable
    | TGClosure (Maybe Type) -- ^ GClosure containing the given API (if
    known)
    | TInterface Name  -- ^ A reference to some API in the GIR
      deriving (Eq, Show, Ord)
```

Listing 2.15: Tipo Type di haskell-gi

```haskell
data API
    = APIConst     Constant
    | APIFunction  Function
    | APICallback  Callback
    | APIEnum      Enumeration
    | APIFlags     Flags
    | APIInterface Interface
    | APIObject    Object
    | APIStruct    Struct
    | APIUnion     Union
    deriving Show
```

Listing 2.16: Tipo API di haskell-gi

Il tipo API ha inoltre la particolarità di rappresentare un'unità di generazione del codice in haskell-gi. Questo significa che ogni variabile di tipo API verrà considerato come un modulo a sé stante.

2.5.2 Moduli

Il tipo Name che è possibile estrarre dal costruttore di `TInterface` è molto usato all'interno della libreria. Questo, in aggiunta al nome del tipo contiene il namespace, ovvero la libreria a cui appartiene questo tipo, che può essere non solo GTK, ma anche una delle sue dipendenze.

```
data Name = Name { namespace :: Text, name :: Text }
    deriving (Eq, Ord, Show)
```

Listing 2.17: Tipo Name di haskell-gi

Ad ogni modulo preso in considerazione da haskell-gi non sempre corrisponde un file generato in output. Questo infatti è vero solo per interfacce, oggetti e struct. Le restanti API vengono raggruppate per tipo in un unico file.

```
GI
|-- Gtk
|   |-- Callbacks.hs
|   |-- Config.hs
|   |-- Constants.hs
|   |-- Enums.hs
|   |-- Flags.hs
|   |-- Functions.hs
|   |-- Interfaces
|   |   |-- *.hs
|   |-- Objects
|   |   |-- *.hs
|   |-- Structs
|   |   |-- *.hs
|-- Gtk.hs
```

Listing 2.18: Struttura dei file nelle librerie generate da haskell-gi

Capitolo 3

Struttura di ocaml-gi-gtk

ocaml-gi-gtk [17] è la libreria scritta in questa tesi. Come discusso nel capitolo precedente, questa è stata scritta partendo da haskell-gi ed è pertanto una libreria Haskell che genera i binding di GTK per OCaml. In realtà, esattamente come per haskell-gi, GTK non è l'unica libreria generabile, ma lo sono tutte quelle che usano GObject Introspection. Ciò nonostante, uno degli obiettivi di questa tesi era quello di generare dei binding che fossero il più possibile simili a quelli attualmente disponibili in lablgtk3. Questo ha un duplice beneficio: rende il più semplice possibile la conversione di applicazioni attualmente scritte usando lablgtk3 (ad esempio Matita [1]) e ocaml-gi-gtk può usare lablgtk3 come libreria riusando alcuni meccanismi di conversione tra OCaml e C già collaudati in questa libreria.

3.1 Reperibilità del codice

ocaml-gi-gtk è una libreria open source ed è disponibile su GitHub all'indirizzo `https://github.com/illbexyz/ocaml-gi-gtk`. Facendo uso delle librerie lablgtk e haskell-gi, entrambe distribuite secondo la licenza LGPL, è stata mantenuta la stessa licenza anche per ocaml-gi-gtk.

3.2 Struttura del progetto

ocaml-gi-gtk è configurato come un progetto Stack [20]. Stack è un build system per Haskell e si occupa anche di installare e gestire le dipendenze del progetto. Il progetto è strutturato come segue:

```
|-- Setup.hs
|-- src
|   |-- *.hs
|-- app
|   |-- Main.hs
|-- base-ocaml
|   |-- c
|   |   |-- *.h
|   |   |-- *.c
|   |-- gilablgtk3
|-- examples
|-- bindings
|-- overrides
|-- generate.sh
|-- ocaml-gi-gtk.cabal
|-- package.yaml
|-- stack.yaml
```

Listing 3.1: Struttura del progetto ocaml-gi-gtk

I file principali sono:

- **src**, contiene i file sorgenti Haskell della libreria;
- **app**, contiene solo Main.hs, che è configurato come il main file dell'eseguibile;
- **base-ocaml**, è il corrispettivo del modulo `base` di haskell-gi, ovvero contiene delle funzioni di conversione usate dalle librerie generate. Nello specifico contiene una cartella `c`, dove sono presenti dei file C che verranno importati da quelli generati dalla libreria per effettuare delle operazioni di conversione tra i tipi, e gilablgtk3, che è una versione di lablgtk3 "ridotta" per usare solo le funzionalità utili a questo progetto.

- **examples**, contiene delle piccole applicazioni che fanno uso delle librerie generate e ne testano alcune funzionalità. Questi esempi sono gli stessi che sono forniti in lablgtk3, convertiti per usare le API delle librerie generate, che differiscono per alcuni particolari;
- **bindings**, dentro questa cartella verranno generate le librerie;
- **generate.sh**, è uno shell script che facilita l'esecuzione di ocaml-gi-gtk;
- **overrides**, contiene gli overrides di haskell-gi. Sono dei file che sovrascrivono delle proprietà contenute nei file GIR, in quanto è possibile che queste siano incomplete o incorrette. Ne esiste uno per ogni libreria generabile;
- **package.yaml**, contiene le configurazioni del progetto Stack.

3.3 Definizione delle librerie generabili

Per generare una libreria, questa dev'essere prima configurata e resa nota ad ocaml-gi-gtk. In Main.hs sono contenute le definizioni di tutte le librerie generabili, ed ognuna contiene le seguenti informazioni:

```
1 data Library = Library {
2     name           :: Text,
3     version        :: Text,
4     overridesFile  :: Maybe FilePath
5 }
```

Listing 3.2: Tipo associato ad una libreria generabile

I valori `name` e `version` sono usati dal modulo GIR di haskell-gi per ottenere il giusto file GIR tra quelli installati nel sistema. Pertanto i valori da assegnare a questi campi sono da trovare nei file pkg.info contenuti nella cartella bindings di haskell-gi. `overridesFile` richiede invece il percorso di un file (opzionale) che può essere usato per sovrascrivere delle proprietà contenute nel file GIR. Oltre a definire la libreria, questa deve essere anche aggiunta alla funzione `parseArg`, la funzione che si occupa di effettuare il parsing dei parametri ricevuti in input dall'eseguibile.

```
parseArg :: String -> IO Library
parseArg "glib"       = return glibLib
parseArg "gobject"    = return gobjectLib
parseArg "gio"        = return gioLib
parseArg "atk"        = return atkLib
parseArg "cairo"      = return cairoLib
parseArg "pango"      = return pangoLib
parseArg "gdkpixbuf" = return gdkPixbufLib
parseArg "gdk"        = return gdkLib
parseArg "gtk"        = return gtkLib
parseArg "gtksource" = return gtkSourceLib
parseArg "-h"         = printUsage >> exitSuccess
parseArg "-v"         = printVersion >> exitSuccess
parseArg _            = printUsage >> exitSuccess
```

Listing 3.3: La funzione parseArg per il parsing degli input argument

Infine per ogni libreria generabile esiste un file `*_includes.h` dentro a `base-ocaml/c`. Questi file contengono delle istruzioni di `#include` per importare gli header delle librerie a cui fanno riferimento. Questi file verranno inclusi dai file C generati da ocaml-gi-gtk. È stato scelto di definire questi file perché per alcune librerie è necessario includere diversi header e talvolta anche definire delle costanti tramite `#define`.

3.4 Esecuzione di ocaml-gi-gtk

Attualmente ocaml-gi-gtk è eseguibile solo in ambiente UNIX. Per eseguire facilmente ocaml-gi-gtk è stato predisposto uno shell script `generate.sh`. Questo script si occupa di settare delle variabili di ambiente necessarie alla compilazione della libreria e di lanciare la generazione e compilazione delle librerie. Le variabili di ambiente da definire sono:

- `BASE_OCAML_C`, deve essere settata al percorso della directory contenente gli header C di `base-ocaml`. Di default è `_project_root_/base-ocaml/c`.
- `GI_INCLUDES`, ogni libreria generata contiene una directory `include` al proprio interno. I file header C al suo interno vengono usati dalle librerie che dipendono

da questa libreria. Questa variabile d'ambiente dovrà contenere il percorso a tali directory per ogni libreria generabile, ed ogni percorso deve essere preceduto dal simbolo `-I`.

Il motivo della presenza di queste variabili di ambiente verrà spiegato nella sezione 3.5. Lo script può ricevere in input il nome di una o più librerie da generare. È presente anche l'opzione `-f` per generare tutte le librerie disponibili. Un'altra opzione disponibile è `-i`, che serve per installare nel sistema la libreria appena generata. L'operazione di installazione è necessaria perchè le librerie generabili hanno delle dipendenze tra loro. Quindi la compilazione (ed installazione) deve partire dalla libreria di più basso livello, proseguendo fino ad arrivare a GTK, quella di più alto livello. Per questo motivo il modo raccomandato di eseguire ocaml-gi-gtk, almeno la prima volta, è quello di usare `./generate.sh -f -i`, che compila ed installa le libreria nell'ordine corretto.

```
1 Usage: generate.sh [OPTION] LIB [LIB...]
2
3 Available libs are: glib gobject gio atk cairo pango gdkpixbuf gdk gtk
4
5 Options:
6   -h, (help)     Display this help and exit
7   -f, (full)     Generate every available library. The LIB arguments
     will be ignored.
8   -i, (install)  After the library generation, it will install it using
     'dune install'
```

Listing 3.4: Messaggio usage di generate.sh

3.5 Struttura delle librerie generate

Le librerie vengono generate all'interno della directory `bindings` come dei progetti dune [2]. Dune è un build system per OCaml che permette di descrivere come compilare il progetto attraverso dei file con nome `dune`. I file `dune` vengono scritti attraverso notazione S-expression e, tramite keyword specifiche di dune, è possibile specificare: metadati relativi al progetto, dipendenze, script da eseguire prima della compilazione, come linkare OCaml con codice C, ed altre proprietà.

I file che troviamo all'interno di ogni libreria generata da ocaml-gi-gtk sono:

- **dune-project**, specifica che questa è la root del progetto dune e specifica il nome della libreria;
- **LibName**, dove il nome varia a seconda del nome della libreria generata, è una directory che contiene tutti i file OCaml che implementano i binding;
- **include**, directory che contiene tutti gli header C relativi ai file generati;
- **tools**, directory copiata da `_project_root_/base-ocaml/tools`, contiene delle istruzioni necessarie a dune per trovare le librerie di sistema relative a GTK;
- **dune**, contiene le istruzioni necessarie per usare tools.

Inoltre dentro alla directory LibName è presente un'ulteriore file `dune`. Questo file istruisce il build system su quali file devono essere compilati e come devono essere linkati tra loro i file OCaml con quelli C. Le variabili di ambiente viste nella sezione precedente 3.4 vengono qui riportate (non espanse) per indicare al linker dove trovare gli header necessari. Inoltre, se la libreria corrente ha delle dipendenze verso altre librerie nella gerarchia di GTK, queste vengono indicate come dipendenze.

```
(lang dune 2.0)
(package
 (name GIGtk))
```

Listing 3.5: Esempio di file `dune-project` (GTK)

```
(env
 (_
  (binaries
   ./tools/dune_config.exe)))
```

Listing 3.6: Esempio di file `dune` generato nella root del progetto

```
(rule
 (targets
  cflag-gtk+-3.0.sexp
  clink-gtk+-3.0.sexp)
```

```
5  (action (run dune_config -pkg gtk+-3.0 -version 3.18)))
6 (library
7  (name GIGtk)
8  (public_name GIGtk)
9  (libraries gilablgtk3 GIAtk GIGLib GIGObject GIGdk GIGdkPixbuf GIGio
     GIPango GIcairo)
10 (flags :standard -w -6-7-9-10-27-32-33-34-35-36-50-52 -no-strict-
     sequence)
11 (c_library_flags (:include clink-gtk+-3.0.sexp))
12 (foreign_stubs
13  (language c)
14  (names AboutDialog AccelGroup ... Enums)
15  (include_dirs %{project_root}/include)
16  (flags (:include cflag-gtk+-3.0.sexp) -I$BASE_OCAML_C $GI_INCLUDES -
     Wno-deprecated-declarations)))
```

Listing 3.7: Esempio di file `dune` generato nella subdirectory del progetto (GTK)

Capitolo 4

Implementazione

L'implementazione di ocaml-gi-gtk è iniziata effettuando un fork di haskell-gi ed analizzando come questo impostasse la generazione dei binding per Haskell per guidare la generazione dei binding per OCaml.

4.1 Impostazione della generazione

Le strutture costruite per impostare la generazione di codice di haskell-gi sono state mantenute e modificate per essere rese compatibili con la generazione di file OCaml. In particolare queste strutture si trovano dentro il file Code.hs, che contiene i meccanismi di generazione di più basso livello. Gli altri file di più alto livello che contengono la logica di generazione del codice usano le funzioni qui definite. Nello specifico vengono definite delle monadi che definiscono un contesto di computazione per la generazione del codice.

```haskell
-- | The base type for the code generator monad.
type BaseCodeGen excType a
  = ReaderT CodeGenConfig (StateT (CGState, ModuleInfo) (Except excType
    )) a

-- | Code generators that cannot throw errors.
type CodeGen a = forall e . BaseCodeGen e a

-- | Code generators that can throw errors.
type ExcCodeGen a = BaseCodeGen CGError a
```

Listing 4.1: Monadi per la generazione del codice

Queste monadi vengono implementate tramite uno stack di monad transformer per combinare gli effetti delle monadi Reader, State ed Except. Rispettivamente servono per definire:

- una configurazione read-only disponibile a tutte le funzioni che operano dentro la monade. Questa conterrà informazioni che non cambiano durante la generazione di un modulo, come il nome del modulo che sta venendo generato o la hashmap costruita dal parser che fa corrispondere i tipi Name ai tipi API visti nella sezione 2.5;

- uno stato della computazione disponibile a tutte le funzioni che operano dentro la monade. Questo conterrà informazioni che vengono modificate man mano che si procede nella generazione di un modulo, come ad esempio il codice che viene generato, aggiunto man mano ad una sequenza;

- la possibilità di fallire la computazione. Viene definito anche un tipo `CGError` che enumera i possibili errori. Le computazioni che possono dare luogo ad errori sono quelle dentro la monade `ExcCodeGen` e vanno gestite attraverso la funzione `handleCGExc`.

Le informazioni più interessanti sono quelle definite dentro lo stato:

```
data ModuleInfo = ModuleInfo {
      modulePath :: ModulePath -- ^ Full module name: ["Gtk", "Label"].
    , moduleCode :: Code       -- ^ Low level OCaml code.
    , gCode      :: Code       -- ^ High level OCaml code
    , tCode      :: Code       -- ^ Type declaration code
    , cCode      :: Code       -- ^ .c stubs' code
    , hCode      :: Code       -- ^ .h stubs' code
    , cDeps      :: Deps       -- ^ Set of dependencies for this module
    (c-side)
    , submodules :: M.Map Text ModuleInfo -- ^ Indexed by the relative
                                          -- module name.
    } deriving (Show)
```

Listing 4.2: Informazioni contenute dentro la stato di generazione di un modulo

Per ogni tipo di file che viene creato (*.ml, *G.ml, *T.ml, *.c, *.h) viene mantenuto nello stato il codice generato. Per modificare queste variabili vengono definite dentro a Code.hs delle funzioni `line` che appendono una nuova linea di codice alla sequenza finora generata. Per ogni tipo di file esiste una funzione differente (`line`, `gline`, `tline`, `cline`, `hline`).

Un'altra variabile interessante è cDeps. Questa è stata aggiunta per tenere traccia dell'insieme di header C dai quali il modulo corrente dipende. Tali file vengono aggiunti dalla funzione `addCDep` e verranno inclusi nell'intestazione del file *.c del modulo corrente.

4.1.1 Esecuzione del generatore

La generazione del codice ha inizio nella funzione `genBindings` di GI.hs. Qui vengono effettuate delle configurazioni e viene richiamata la funzione `genLibraryCode` per eseguire il parsing e la generazione della libreria. Per ogni modulo individuato dal parser viene eseguita la funzione `genAPI` di CodeGen.hs:

```
genAPI :: Name -> API -> CodeGen ()
genAPI _n (APIConst     _c) = return ()
genAPI _n (APIFunction  _f) = return ()
genAPI n  (APIEnum      e ) = genEnum n e
genAPI n  (APIFlags     f ) = genFlags n f
genAPI _n (APICallback  _c) = return ()
```

```
genAPI n  (APIStruct    s ) = genStruct n s
genAPI n  (APIUnion     u ) = genUnion n u
genAPI n  (APIObject    o ) = genObject n o
genAPI n  (APIInterface i ) = genInterface n i
```

Le funzioni `gen*` chiamate da questa funzione sono quelle di alto livello che contengono la logica per la generazione di ogni tipo di modulo. Quelle che contengono `return ()` non sono state gestite.

Una volta terminata la generazione di ogni modulo viene restituita una variabile di tipo `ModuleInfo`. I moduli generati sono disposti secondo una struttura ad albero e contenuti dentro questa variabile all'interno del campo `submodules`. Al primo livello dell'albero vengono creati dei collegamenti a dei nodi di tipo `ModuleInfo` e sono suddivisi a seconda del tipo di modulo. Tale suddivisione è effettuata dalla funzione `submoduleLocation` del file QualifiedNaming.hs.

```
-- | Construct the submodule path where the given API element will
-- live. This is the path relative to the root for the corresponding
-- namespace. I.e. the "GI.Gtk" part is not prepended.
submoduleLocation :: Name -> API -> ModulePath
submoduleLocation _ (APIConst     _) = "Constants"
submoduleLocation _ (APIFunction  _) = "Functions"
submoduleLocation _ (APICallback  _) = "Callbacks"
submoduleLocation n (APIEnum      _) = "Enums"
submoduleLocation n (APIFlags     _) = "Enums"
submoduleLocation n (APIInterface _) = "" /. upperName n
submoduleLocation n (APIObject    _) = "" /. upperName n
submoduleLocation n (APIStruct    _) = "" /. upperName n
submoduleLocation n (APIUnion     _) = "" /. upperName n
```

Listing 4.3: Funzione submoduleLocation di QualifiedNaming.hs

A questa suddivisione corrisponderà il path che avrà ogni modulo una volta che verranno scritti su disco i file generati. Ad esempio per ogni modulo di tipo Object ed Interface, viene creato un file separato, mentre i moduli di tipo Enum e Flag vengono generati tutti dentro gli stessi file `Enums.*`.

La scrittura su disco di ogni modulo viene effettuata dalla funzione `writeModuleInfo` di Code.hs. Questa si occupa di controllare la presenza di codice dentro i campi `*Code` di `ModuleInfo` e nel caso ve ne sia, viene scritto il file su disco.

4.2 Conversione di tipi tra OCaml e C

Come spiegato nella sezione 2.3, per effettuare i binding tra OCaml e C è necessario definire delle funzioni C chiamate stubs, che siano in grado di convertire i tipi OCaml in tipi C e viceversa. L'approccio usato per generare gli stubs è lo stesso adottato da lablgtk. Nel file `wrappers.h` sono definite delle macro `ML_X` dove x è il numero di parametri che prende in input la funzione f di cui si vuole creare uno stub. La macro ha sempre come primo parametro il nome di f e come ultimo parametro il nome di una funzione per convertire da C ad OCaml il valore restituito da f. Tra questi due parametri sono attesi anche altri x parametri, dove ognuno è il nome di una funzione per convertire da OCaml a C i valori che saranno passati ad f come parametro. La macro genera una funzione che riceve in input i valori OCaml che saranno usati da f, e si occupa di chiamare la funzione f con i parametri convertiti. Infine converte il valore restituito da f da C a OCaml e lo restituisce. In lablgtk il nome della funzione generata assume il nome di f con un prefisso `ml_`. In ocaml-gi-gtk questa convenzione è stata modificata usando il prefisso `ml_gi`, in modo da non creare conflitti tra i nomi delle funzioni generate dalle due librerie.

```
1 #define ML_2(cname, conv1, conv2, conv) \
2 CAMLprim value ml_gi##cname (value arg1, value arg2) \
3 { return conv (cname (conv1(arg1), conv2(arg2))); }
```

Listing 4.4: Esempio di macro per la creazione di stub

In ocaml-gi-gtk le funzioni necessarie a convertire i tipi tra OCaml e C vengono specificate nel file Conversions.hs. Le funzioni in questione sono `ocamlValueToC` e `cToOCamlValue`, entrambe prendono in input il tipo `Type` da convertire e restituiscono il nome della funzione C che effettua la conversione. Per i tipi di base vengono usate le funzioni di conversione della libreria di interfacciamento con C di OCaml o delle macro definite da lablgtk. Per i

tipi non di base come oggetti, interfacce ed enum vengono usate delle specifiche funzioni che vengono generate per ogni modulo nei corrispondenti file *.h e *.c.

4.3 Enum e Flags

Le Enum in C sono un tipo di dato che può assumere un numero finito di valori. Questi valori vengono elencati associando un identificatore ad un valore intero. Ad esempio in GTK l'orientamento di una pagina viene definito tramite la seguente enum:

```
typedef enum {
    GTK_PAGE_ORIENTATION_PORTRAIT = 0,
    GTK_PAGE_ORIENTATION_LANDSCAPE = 1,
    GTK_PAGE_ORIENTATION_REVERSE_PORTRAIT = 2,
    GTK_PAGE_ORIENTATION_REVERSE_LANDSCAPE = 3
} GtkPageOrientation;
```

Listing 4.5: Esempio di enum in C

Nel caso in cui i valori interi non vengano definiti esplicitamente, di default verranno assegnati degli interi crescenti a partire da 0.

Un uso che viene fatto delle enum in C è quello di definire dei flag. Questi sono delle enum dove i valori interi sono delle potenze di 2, in modo da poter essere combinati tramite OR bitwise per rappresentare una combinazione di più valori della enum.

4.3.1 Rappresentazione in lablgtk3

In lablgtk3 le enum vengono rappresentate in OCaml tramite varianti polimorfe con costruttori senza parametri. Un'altra possibile scelta sarebbe potuta essere quella di rappresentarle tramite ADT di tipo somma, ma siccome diverse enum possono condividere un costruttore con lo stesso nome, la scelta delle varianti polimorfe è particolarmente adatta.

I costruttori delle varianti polimorfe vengono rappresentati in C tramite degli interi, dove il valore di questi viene calcolato tramite una funzione di hash `hash_variant` messa a disposizione dalla libreria di interfacciamento tra codice OCaml e C, che calcola il valore intero a partire dal nome del costruttore. Il meccanismo usato per far corrispondere ad

ogni costruttore dell'enum il valore intero originario dell'enum è quello di definire una struct `lookup_info` che associa ad una chiave di tipo `value` un valore di tipo `int`. Ogni enum viene poi codificata come un array di queste struct, dove il primo elemento contiene come valore la dimensione dell'array.

```
const lookup_info ml_gigtk_table_page_orientation[] = {
  { 0, 4 },
  { MLTAG_REVERSE_LANDSCAPE, GTK_PAGE_ORIENTATION_REVERSE_LANDSCAPE },
  { MLTAG_LANDSCAPE, GTK_PAGE_ORIENTATION_LANDSCAPE },
  { MLTAG_PORTRAIT, GTK_PAGE_ORIENTATION_PORTRAIT },
  { MLTAG_REVERSE_PORTRAIT, GTK_PAGE_ORIENTATION_REVERSE_PORTRAIT },
};
```

Listing 4.6: Esempio di tabella lookup_info in C

Lablgtk3 definisce infine due funzioni, `ml_lookup_from_c(const lookup_info table[], int data)` e `ml_lookup_to_c(const lookup_info table[], value key)`, per recuperare i valori contenuti dentro una `lookup_info`. Queste funzioni scandiscono l'array in input finché non trovano la chiave/valore cercata, per poi restituire il valore/chiave corrispondente. Per abbassare la complessità di questa operazione viene assunto che le chiavi di `lookup_info` (ovvero i valori di hash) siano ordinate in ordine crescente. Riassumendo, la conversione avviene in 3 fasi:

- Vengono generate lato OCaml delle varianti polimorfe per rappresentare i valori dell'enum;
- Ad ogni costruttore delle varianti polimorfe va fatto corrispondere un valore di hash;
- Ad ogni valore di hash deve corrispondere il valore intero originario contenuto nell'enum.

La conversione dei flag avviene in modo analogo a quella delle enum, con la differenza che in OCaml i costruttori delle varianti polimorfe non vengono usati singolarmente, ma contenuti in una lista. Per questo motivo è necessario definire degli ulteriori binding in C che siano in grado di convertire una lista OCaml contenente i costruttori della variante

polimorfa nel valore reale del flag (ovvero l'OR bitwise). Per definire tali binding sono presenti due macro, `Make_Flags_val` e `Make_OptFlags_val`.

4.3.2 Generazione in ocaml-gi-gtk

In ocaml-gi-gtk è stato deciso di seguire lo stesso approccio di lablgtk3 nella generazione di enums e flags.

Il parser di haskell-gi mette a disposizione le seguenti informazioni sulle enum:

```
data Enumeration = Enumeration {
    enumMembers       :: [EnumerationMember],
    ...
}

data EnumerationMember = EnumerationMember {
    enumMemberName    :: Text,
    enumMemberValue   :: Int64,
    enumMemberCId     :: Text,
    ...
}
```

Listing 4.7: API di haskell-gi per enums

Usando queste informazioni vengono generati in bindings usando lo stesso approccio di lablgtk3. Nello specifico vengono generati tre file, Enums.h, Enums.c ed Enums.ml.

Enums.h

Per ogni enum e flag vengono generati:

- i valori di hash corrispondenti ai costruttori delle varianti polimorfe OCaml. Per calcolare il valore di hash è stata convertita in Haskell la funzione OCaml che calcola tali valori;
- la signature della tabella di lookup, definita nel file Enums.c;
- le macro per la conversione dell'enum/flag da OCaml a C e viceversa.

```
...
#define MLTAG_PORTRAIT          ((value)(305443163*2+1))
#define MLTAG_LANDSCAPE         ((value)(-133830629*2+1))
#define MLTAG_REVERSE_PORTRAIT  ((value)(765848440*2+1))
#define MLTAG_REVERSE_LANDSCAPE ((value)(-542668962*2+1))

extern const lookup_info ml_gigtk_table_page_orientation[];
#define GI_Val_GtkPageOrientation(data) ml_lookup_from_c (
   ml_gigtk_table_page_orientation, data)
#define GI_GtkPageOrientation_val(key) ml_lookup_to_c (
   ml_gigtk_table_page_orientation, key)
...
```

Listing 4.8: Estratto del contenuto di Enums.h

Enums.c

Per ogni enum e flag vengono generati:

- la tabella di lookup, facendo attenzione ad ordinare gli elementi secondo l'ordine crescente del valore di hash;
- una funzione che restituisce la tabella di lookup castata al tipo `value`;
- (solo per i flag) vengono chiamate le macro `Make_Flags_val` e `Make_OptFlags_val` per generare le funzioni di conversione per le liste di flag OCaml.

Enums.ml

Per ogni enum e flag vengono generati:

- il tipo OCaml dell'enum/flag, ovvero una variante polimorfa contenente i possibili costruttori;
- un binding `external` alla funzione per ottenere la tabella di lookup. Questa funzione viene subito chiamata;

- un `data_conv` per l'enum/flag usando la tabella di lookup appena ottenuta. I `data_conv` sono un meccanismo di lablgtk3 per rendere più agevole la conversione dei tipi tra OCaml e C nei binding delle Properietà e dei Segnali.

```
1 ...
2 type page_orientation = [ `PORTRAIT | `LANDSCAPE | `REVERSE_PORTRAIT |
    `REVERSE_LANDSCAPE ]
3
4 external get_page_orientation_table : unit -> page_orientation Gpointer
    .variant_table = "ml_gigtk_gtk_get_page_orientation_table"
5 let page_orientation_tbl = get_page_orientation_table ()
6 let page_orientation = Gobject.Data.enum page_orientation_tbl
7 ...
```

Listing 4.9: Estratto del contenuto di Enums.ml

4.4 Oggetti

GTK è una libreria basata su GObject, la quale fornisce la possibilità di strutturare il codice C secondo un modello ad oggetti tramite dei meccanismi di classi, ereditarietà ed interfacce. Per ottenere dei binding OCaml che fossero il più vicini possibile a questo modello, gli autori di lablgtk3 hanno fatto uso delle funzionalità ad oggetti di OCaml come visto nella sezione 2.4. In ocaml-gi-gtk i binding sono stati generati seguendo lo stesso schema, ma è stata cambiata la convenzione dei nomi. Per ogni oggetto vengono definiti i seguenti file:

- ***T.ml**, contiene le definizioni dei tipi OCaml associati all'oggetto;
- ***.ml**, corrisponde ai file di "basso livello" di lablgtk3;
- ***G.ml**, contiene le definizioni delle classi OCaml che rappresentano l'oggetto. Questo è il file che verrà usato dagli utenti della libreria;
- ***.c**, contiene i binding C;
- ***.h**, definisce delle macro per la conversione dell'oggetto tra C e OCaml.

Tali file vengono generati a partire dalle informazioni ottenute dal parser GIR di haskell-gi, che per ogni oggetto espone le seguenti informazioni:

```
data Object = Object {
    objParent         :: Maybe Name,
    objTypeInit       :: Text,
    objTypeName       :: Text,
    objCType          :: Maybe Text,
    objInterfaces     :: [Name],
    objDeprecated     :: Maybe DeprecationInfo,
    objDocumentation  :: Documentation,
    objMethods        :: [Method],
    objProperties     :: [Property],
    objSignals        :: [Signal]
    }
```

Listing 4.10: API di haskell-gi per gli oggetti

Nel resto della sezione seguiranno i dettagli su come queste informazioni vengono usate per la generazione di ogni componente degli oggetti.

4.4.1 Macro di conversione

Per ogni oggetto nei file *.c e *.h vengono generate delle macro per eseguire la conversione tra i tipi OCaml e C. Vengono generate tre macro:

- conversione da OCaml a C, la macro ha il nome dell'oggetto corrente, seguito dal suffisso `_val`. Prende in input il valore da convertire ed usa la macro `check_cast` di lablgtk per effettuare la conversione. Questa macro controlla che il valore in input non sia `null`, e nel caso applica una macro di conversione definita dal sistema di tipi di GTK (che è definita per ogni GObject). Questa macro controlla che il puntatore ricevuto in input sia effettivamente del tipo relativo alla classe in cui vuole essere convertita, ed applica la conversione. Nel caso in cui il tipo non corrisponda, viene emesso un warning a runtime;

- conversione da C ad OCaml, la macro ha il nome dell'oggetto corrente, preceduto dal prefisso `Val_`. Non fa altro che usare la macro `Val_GAnyObject` di lablgtk. Questa

macro controlla che il puntatore ricevuto in input sia di tipo GObject e fa un cast a `GObject*` che è compatibile con il tipo `value` di OCaml;

- conversione da C ad OCaml per un valore opzionale, ovvero che può essere `null` e che in OCaml viene codicato con il tipo `option`. Per questa conversione non basta una macro, ma deve essere generata una funzione ad hoc nel file .c. Questa funzione viene generata tramite la macro `Make_Val_option` di lablgtk. La funzione generata controlla che il valore contenuto nel puntatore che riceve come parametro non sia `null` e nel caso alloca un blocco OCaml che corrisponde al costruttore `Some` di `option`. Nel caso il valore sia `null`, viene restituita la macro `Val_unit` che è compatibile con il costruttore `None` di `option`. Nel file .h viene riportata la signature della funzione generata per renderla visibile agli altri file.

```
#define GtkButton_val(val) check_cast(GTK_BUTTON, val)
#define Val_GtkButton Val_GAnyObject
value OptVal_GtkButton (GtkButton*);
```

Listing 4.11: Esempio di macro di conversione nel file .h (GtkButton)

```
Make_Val_option(GtkButton,Val_GtkButton)
```

Listing 4.12: Esempio di macro di conversione nel file .c (GtkButton)

```
CAMLprim value ml_some (value v)
{
     CAMLparam1(v);
     value ret = alloc_small(1,0);
     Field(ret,0) = v;
     CAMLreturn(ret);
}

#define Val_option(v,f) (v ? ml_some(f(v)) : Val_unit)
#define Make_Val_option(T, Val) \
value Opt##Val(T* v) { return Val_option(v,Val); }
```

Listing 4.13: Macro `Make_Val_option` di lablgtk

4.4.2 Segnali

I segnali sono un meccanismo di GObject per permettere l'esecuzione di una callback al verificarsi di un determinato evento. Ad esempio GtkButton possiede il segnale `clicked` che permette di eseguire una callback alla pressione del bottone. Le callback non possono essere funzioni qualsiasi, ma devono avere un certo tipo definito dal segnale stesso. In altre parole, le callback devono avere dei parametri di un certo tipo e restituire un valore di un determinato tipo.

I segnali vengono generati esattamente come erano definiti in lablgtk3, in modo da poter usare i meccanismi di creazione e gestione delle closure già definiti in questa libreria.

Basso livello

Nel file *.ml viene definito un modulo interno S che per ogni segnale contiene dei record di tipo GtkSignal.t, dove GtkSignal è un modulo di lablgtk3.

```
module S = struct
    open GtkSignal
    open Gobject
    open Data
    let popdown = {name="popdown"; classe=‘gtk_scale_button; marshaller
   =marshal_unit}
    let popup = {name="popup"; classe=‘gtk_scale_button; marshaller=
   marshal_unit}
    let value_changed = {name="value_changed"; classe=‘gtk_scale_button
   ; marshaller=
        fun f -> marshal1 double "ScaleButton::value_changed" f}
end
```

Listing 4.14: Esempio di generazione dei segnali nel file *.ml

La parte più complicata da generare è la funzione di marshalling. Questa si occupa di fare la conversione tra i tipi OCaml e C per i parametri ed il valore di ritorno della callback. lablgtk3 mette a disposizione le funzioni `marshalX`, dove la X indica il numero di parametri attesi dalla callback. Per ognuna di queste funzioni, ne esiste una variante con suffisso `_ret` da usare nel caso in cui la callback abbia un valore di ritorno diverso da `void`. I parametri da passare alle funzioni `marshalX` sono di tipo `data_conv`, definito nel modulo

Gobject di lablgtk3. All'interno di questo modulo, sono anche definite delle possibili istanze di questo tipo da usare per convertire i tipi di dato di GLib e GObject. Fanno eccezione i `data_conv` per enum e flag, che sono non presenti in questo file ma vengono generati nel modulo Enums.ml come descritto nella sezione precedente. I `data_conv` per i tipi oggetto devono essere invece annotati con il tipo dell'oggetto da convertire poichè il compilatore non è in grado di inferirlo. Dalle API di haskell-gi è possibile ottenere una proprietà del segnale dl tipo `Callable`, e tramite questa è possibile ottenere il `Type` di tutti i parametri e valore di ritorno della callback associata al segnale. Nel file `Conversions.hs` è stata definita una funzione `ocamlDataConv` che converte un `Type` nel corretto `data_conv`.

```
module S = struct
    ...
    let drag_action_ask = {...; marshaller
      fun f -> marshal1_ret ~ret:int int "PlacesSidebar::
   drag_action_ask" f}

    let mount = {...; marshaller=fun f ->
      marshal1
        (gobject : GIGio.MountOperationT.t Gobject.obj data_conv)
        "PlacesSidebar::mount" f}

    let open_location = {...; marshaller=fun f ->
      marshal2
        (gobject : GIGio.FileT.t Gobject.obj data_conv)
        Enums.places_open_flags
        "PlacesSidebar::open_location" f}

    let populate_popup = {...; marshaller=fun f ->
      marshal3
        (gobject : WidgetT.t Gobject.obj data_conv)
        (gobject_option : GIGio.FileT.t Gobject.obj option data_conv)
        (gobject_option : GIGio.VolumeT.t Gobject.obj option data_conv)
    "PlacesSidebar::populate_popup" f}
    ...
end
```

Listing 4.15: Esempio di segnali da PlacesSidebar.ml

Alto livello

Nel file *G.ml viene invece definita una classe adibita al contenere i soli segnali. Questa classe porta il nome dell'oggetto in esame, seguita dal suffisso `_signals`. Per ogni segnale dell'oggetto viene generato un metodo avente il nome del segnale ed implementato attraverso un metodo `connect`. Questo metodo viene ereditato dalla classe `gobject_signals` da cui tutte le classi `_signals` ereditano.

```
class button_signals obj = object (self)
  ...
  method activate = self#connect Button.S.activate
  method clicked = self#connect Button.S.clicked
  method enter = self#connect Button.S.enter
  method leave = self#connect Button.S.leave
  method pressed = self#connect Button.S.pressed
  method released = self#connect Button.S.released
end
```

Listing 4.16: Esempio di generazione di una classe signals (GtkButton)

Il metodo `connect` applicato ai `GtkSignal.t` definiti nel file di basso livello produce una funzione che riceve in input una funzione OCaml. Facendo uso del marshaller definito nel file di basso livello viene creata una `GClosure`, un tipo di GLib per definire una callback. Tale closure viene associata al segnale tramite il metodo `g_signal_connect_closure` di GObject e la funzione OCaml ricevuta in input sarà eseguita all'attivarsi dell'evento associato al segnale.

4.4.3 Proprietà

Le proprietà sono un meccanismo di GObject per definire quelli che nella programmazione orientata agli oggetti sono conosciuti come attributi di classe. Anche in questo caso vengono sfruttati i meccanismi di lablgtk3 per effettuare i binding.

```
data PropertyFlag = PropertyReadable
                  | PropertyWritable
                  | PropertyConstruct
                  | PropertyConstructOnly
                    deriving (Show,Eq)
```

```
6
7  data Property = Property {
8      propName :: Text,
9      propType :: Type,
10     propFlags :: [PropertyFlag],
11     propReadNullable :: Maybe Bool,
12     propWriteNullable :: Maybe Bool,
13     propTransfer :: Transfer,
14     propDoc :: Documentation,
15     propDeprecated :: Maybe DeprecationInfo
16 } deriving (Show, Eq)
```

Listing 4.17: Informazioni sulle proprietà estratte dal file GIR

Basso livello

Nel file di basso livello *.ml viene creato un modulo interno P contenente dentro al quale per ogni proprietà posseduta dall'oggetto corrente viene generata una variabile di tipo `Gobject.property`. Questo è un record contenente un campo `name` a cui va assegnato il nome della proprietà ed un campo `conv` a cui va assegnato il `data_conv` del tipo della proprietà. Per ottenere quest'ultimo viene ottenuto il tipo `Type` della proprietà corrente e si ottiene il `data_conv` corrispondente tramite la funzione `ocamlDataConv` di Conversions.hs.

```
module P = struct
  open Gobject
  open Data
  let always_show_image : ([> ButtonT.t],_) property =
    {name="always-show-image"; conv=boolean}
  let image : ([> ButtonT.t],_) property =
    {name="image";
     conv=(gobject_option : WidgetT.t Gobject.obj option data_conv)}
  let image_position : ([> ButtonT.t],_) property =
    {name="image-position"; conv=Enums.position_type}
  let label : ([> ButtonT.t],_) property =
    {name="label"; conv=string}
  let xalign : ([> ButtonT.t],_) property =
    {name="xalign"; conv=float}
  ...
end
```

Listing 4.18: Esempio di proprietà generate (Button.ml)

Alto livello

Nel file di alto livello *G.ml le proprietà vengono riportate all'interno di una classe avente suffisso `_skel` insieme ai metodi. Ad ogni proprietà è associata una lista di flag (tipo `PropertyFlag` in haskell-gi) e a seconda dei flag di una proprietà varia la generazione del codice:

- **read**, indica che il valore della proprietà può essere letto, pertanto viene generato un metodo "getter". Questo prende il nome della proprietà (senza un prefisso `get_`) e viene usata la funzione `Gobject.get` di lablgtk per implementare effettivamente la funzionalità.

- **write**, indica che il valore della proprietà è modificabile, pertanto viene generato un metodo "setter". Questo prende il nome della proprietà con un prefisso `set_` e viene usata la funzione `Gobject.set` di lablgtk per implementare effettivamente la funzionalità.

- **construct e construct-only**, indicano che una proprietà può essere settata in fase di creazione dell'oggetto. Questi flag tuttavia non sono rilevanti in questa fase di generazione del codice, ma saranno trattate nella generazione dei costruttori 4.4.5.

```
class button_skel obj = object (self)
  (* Properties *)
  method always_show_image = Gobject.get Button.P.always_show_image obj
  method set_always_show_image = Gobject.set Button.P.always_show_image
    obj
  method image = Gobject.get Button.P.image obj
  method set_image = Gobject.set Button.P.image obj
  method image_position = Gobject.get Button.P.image_position obj
  method set_image_position = Gobject.set Button.P.image_position obj
  method label = Gobject.get Button.P.label obj
  method set_label = Gobject.set Button.P.label obj
  method xalign = Gobject.get Button.P.xalign obj
  method set_xalign = Gobject.set Button.P.xalign obj
  ...
```

Listing 4.19: Esempio di proprietà generate (ButtonG.ml)

Le funzioni `Gobject.get` e `Gobject.set` tramite l'uso del `data_conv` sono in grado di convertire il tipo OCaml della proprietà nel tipo C, e fanno uso delle seguenti funzioni C di GObject per ottenere e modificare il valore delle proprietà.

```
void
g_object_get_property (GObject *object,
                       const gchar *property_name,
                       GValue *value);

void
g_object_set_property (GObject *object,
                       const gchar *property_name,
                       const GValue *value);
```

Listing 4.20: Funzioni di GObject per leggere/scrivere le proprietà

4.4.4 Metodi

Seguendo l'esempio di lablgtk, per effettuare dei binding ai metodi di un oggetto è necessario definire:

- le funzioni di stub in un file *.c;
- le dichiarazioni external nel file di basso livello *.ml. In queste dichiarazioni deve essere esplicitato il tipo della funzione;
- i metodi dentro una classe OCaml, che chiamano le funzioni external definite nel file di basso livello.

```
data Method = Method {
        methodName         :: Name,
        methodSymbol       :: Text,
        methodType         :: MethodType,
        methodMovedTo      :: Maybe Text,
        methodCallable     :: Callable
    } deriving (Eq, Show)

data Callable = Callable {
        returnType             :: Maybe Type,
        returnMayBeNull        :: Bool,
        returnTransfer         :: Transfer,
        returnDocumentation    :: Documentation,
        args                   :: [Arg],
        skipReturn             :: Bool,
        callableThrows         :: Bool,
        callableDeprecated     :: Maybe DeprecationInfo,
        callableDocumentation  :: Documentation
    } deriving (Show, Eq)

data Arg = Arg {
        argCName            :: Text,
        argType             :: Type,
        direction           :: Direction,
        mayBeNull           :: Bool,
        argDoc              :: Documentation,
        argScope            :: Scope,
        argClosure          :: Int,
        argDestroy          :: Int,
        argCallerAllocates  :: Bool,
        transfer            :: Transfer
    } deriving (Show, Eq, Ord)

data Direction = DirectionIn
               | DirectionOut
               | DirectionInout
                 deriving (Show, Eq, Ord)
```

Listing 4.21: Informazioni relative ai metodi estratte dal file GIR

Stub C

Gli stub vengono generati sfruttando le macro `ML_X` descritte nella sezione 4.2. Le informazioni necessarie per la generazione di queste macro sono contenute nel campo `methodCallable` di `Method`. Da quest'ultimo vengono estratte le informazioni relative ai parametri del metodo tramite il campo `args`. Ogni parametro p possiede un campo `direction` che indica se p è un parametro di input, di output o entrambi. I parametri sia di input che di output non sono attualmente gestiti e viene fallita la generazione del metodo. Altrimenti, se il metodo contiene solo parametri di input allora viene generata una macro `ML_X`. Se sono presenti parametri di output viene invece usata una macro `ML_Xin_Yout`, dove x indica il numero di parametri di input, e y indica il numero di parametri di output.

```
1 #define ML_2in_2out(namespace, cname, conv1in, conv2in, cType1out,
     conv1out, cType2out, conv2out, convRes) \
2 CAMLprim value ml_gi##namespace##_##cname (value arg1, value arg2) \
3 { \
4     CAMLparam2(arg1, arg2); CAMLlocal1(tuple); \
5     cType1out a; cType2out b; tuple = alloc_tuple(3); \
6     value res = convRes(cname(conv1in(arg1), conv2in(arg2), &a, &b)); \
7     Store_field(tuple, 0, res); Store_field(tuple, 1, conv1out(a));
     Store_field(tuple, 2, conv2out(b)); \
8     CAMLreturn (tuple); \
9 }
```

Listing 4.22: Esempio di macro `ML_Xin_Yout`

In aggiunta ai parametri presenti anche nelle macro `ML_X`, questa macro riceve in input anche il tipo C dei parametri di output. Questi tipi vengono usati per allocare sullo stack le variabili che saranno passate per indirizzo alla funzione C. Viene infine allocata una tupla OCaml nella quale vengono inseriti i valori (convertiti nel tipo OCaml) di ritorno della funzione e dei parametri di output. Infine questa tupla viene restituita. Per ogni macro `ML_Xin_Yout` è presente anche una macro `ML_Xin_Yout_discard_ret` che viene usata per le funzioni che restituiscono `void`. In questo caso il valore di ritorno della funzione non viene inserito nella tupla. L'uso di queste macro `ML_Xin_Yout` presenta tuttavia alcuni problemi:

- è stato assunto che nelle funzioni C i parametri di input precedessero sempre i parametri di output. Tuttavia sono stati riscontrati casi in altre librerie della gerarchia di GTK dove questa assunzione è errata;
- i parametri di output con tipo non di base vengono attualmente allocati sullo stack e vengono distrutti alla fine del corpo della funzione. La soluzione prevede di allocarli sullo heap tramite `malloc`, ma non tutti i tipi non di base di GObject sono allocabili in questo modo.

Questi problemi rendono la generazione del codice C poco uniforme e quindi non adatta all'uso di macro C. Pertanto gli stub per le funzioni con parametri di output andrebbero generati scrivendo direttamente in output le funzioni C.

Infine, come spiegato nella sezione 2.3.1, se uno stub ha più di 5 parametri deve essere generato anche uno stub che riceve tali parametri attraverso un array ed il nome di entrambi gli stub va riportato nella dichiarazione external nel file di basso livello *.ml. La generazione di questi stub viene effettuata attraverso le macro `ML_bcX` dove x è il numero di parametri ricevuti dallo stub.

Basso livello

Nel file OCaml di basso livello vengono generate le dichiarazioni external che devono effettuare il binding con gli stub C. Queste dichiarazioni devono anche annotare esplicitamente il tipo della funzione. Questa operazione è complicata in quanto i tipi OCaml da annotare sono anche tipi non di base che fanno uso di più costruttori di tipo. Per gestire la generazione dei tipi è stato quindi definito un tipo intermedio (tra il tipo Type presente nelle API di haskell-gi ed il tipo testuale OCaml) tramite il seguente ADT:

```
1 data RowDirection = Less | More
2   deriving (Show, Eq)
3
4 data TypeRep =
5       ListCon TypeRep                -- list type constructor
6     | OptionCon TypeRep              -- option type constructor
7     | ObjCon TypeRep                 -- Gobject.obj type constructor
8     | RowCon RowDirection TypeRep    -- [> ] and [< ] type constructor
```

```
    | TypeVarCon Text TypeRep       -- adds a type variable ('a, 'b, ...)
    | TupleCon [TypeRep]            -- tuple type constructor ( * * )
    | PolyCon TypeRep               -- To represent a polymorphic variant constructor
    | TextCon Text                  -- For atomic types
    | NameCon Name                  -- For object/interfaces
  deriving (Show, Eq)
```

Listing 4.23: Tipo TypeRep per una rappresentazione intermedia dei tipi OCaml

Il tipo TypeRep è ricorsivo e presenta come costruttori di base `TextCon` e `NameCon`, usati rispettivamente per i tipi di base e per gli oggetti/interfacce. La conversione da Type a TypeRep avviene nelle funzioni `ocamlType` e `outParamOcamlType` di Conversions.hs. Invece tramite la funzione `typeShow` di TypeRep.hs vengono convertiti in testo i TypeRep di ogni parametro dei metodi. Vengono usate due funzioni distinte per ottenere i TypeRep poichè i tipi relativi ad oggetti ed interfacce hanno una rappresentazione diversa in base a seconda che siano dei parametri di input o dei parametri di output. In entrambi i casi il tipo atteso è quello di una variante polimorfa racchiusa nel tipo `Gobject.obj`. Tuttavia viene variato il vincolo sulla variabile row espresso tramite lo zucchero sintattico > o <:

- > viene usato per i parametri di input, rendendo così possibile passare in input alla funzione anche delle sottoclassi del tipo atteso;
- < viene invece usato per i parametri di output, non permettendo la costruzione di un oggetto più in basso nella gerarchia.

```
external enter :
    [> ButtonT.t] Gobject.obj ->
    unit
    =  "ml_gigtk_gtk_button_enter"

external get_alignment :
    [> ButtonT.t] Gobject.obj ->
    (float * float)
    =  "ml_gigtk_gtk_button_get_alignment"

```

```
11 external get_event_window :
12     [> ButtonT.t] Gobject.obj ->
13     [< GIGdk.WindowT.t] Gobject.obj
14     =  "ml_gigtk_gtk_button_get_event_window"
```

Listing 4.24: Estratto di metodi generati nel file di basso livello (GtkButton)

Alto livello

Nel file di alto livello i metodi devono essere generati dentro la classe `_skel`, la quale ha come parametro di classe `obj`, l'oggetto su cui vengono invocati i metodi. Non tutti i metodi hanno come parametro l'oggetto su cui operano, ma nel caso lo abbiano, questo è sempre in prima posizione. I metodi OCaml dentro la classe `_skel` devono quindi richiamare il binding external generato nel file di basso livello passando come primo parametro `obj` nel caso in cui la il metodo lo richieda. Ci sono tuttavia delle complicazioni:

1. se il metodo riceve altri oggetti/interfacce in input in aggiunta all'oggetto su cui opera, il metodo OCaml non deve ricevere in input un tipo `[> *] obj` come indicato nel binding external, ma un oggetto OCaml della corretta classe. Ogni classe dispone di un metodo `#as_objectname` che permette di ottenere il valore di tipo `[ ObjectName.t] obj`. Questo metodo deve essere richiamato sull'oggetto OCaml preso in input per poi passare il valore ottenuto al binding external. Tuttavia questa soluzione presenta un problema: il tipo inferito dal compilatore per l'oggetto è `< as_objectname : [> ObjectNameT.t ] Gobject.obj; .. >`, dove però il simbolo `>` della variante polimorfa è zucchero sintattico per una variabile row che non viene dichiarata all'esterno del metodo, causando un errore di compilazione. Per risolvere il problema bisogna annotare esplicitamente il tipo del metodo avendo cura di riportare le variabili di tipo anche all'esterno del metodo. Per semplificare l'annotazione del tipo degli oggetti, nel file *T.ml viene generato un tipo `class type objectname_o` contenente il solo metodo `#as_objectname`. Dovendo i metodi avere delle annotazioni di tipo differenti da quelle dei binding external per uno stesso valore TypeRep, è stata definita una funzione `methodTypeShow` in TypeRep.hs per ottenere la giusta rappresentazione testuale;

```
method pack_start : 'c.(#WidgetT.widget_o as 'c) -> bool -> bool
   -> int -> unit =
    fun c d e f -> Box.pack_start obj c#as_widget d e
```

Listing 4.25: Esempio di metodo generato avente un oggetto in input

```
class type widget_o = object
  method as_widget : t Gobject.obj
end
```

Listing 4.26: Esempio di class type generata

2. se il metodo restituisce in output un oggetto, questo non deve avere tipo `[< *] obj` come indicato nel binding external, ma deve essere un oggetto OCaml della corretta classe. Per ottenere l'oggetto OCaml bisogna eseguire il metodo external e wrappare il risultato in una `new` della corretta classe, passando in input il valore `obj` della classe `_skel`.

```
method get_action : string -> ActionG.action option =
    fun d -> Option.map (new ActionG.action) (ActionGroup.get_action
   obj d)
```

Listing 4.27: Esempio di metodo avente un oggetto in output

Nel gestire entrambe queste problematiche si presenta un'ulteriore complicazione: l'oggetto/interfaccia in input/output può essere opzionale, ovvero contenuto nel tipo `option` di OCaml. In questo caso, l'operazione di `#as_objectname` o `new` deve essere eseguita all'interno di una funzione anonima che viene data in input ad `Option.map`, una funzione della libreria standard di OCaml che riceve in input una funzione ed un valore opzionale. La funzione viene applicata al valore opzionale solo se tale valore è creato con il costruttore `Some`.

```
method join_group :
  'c.(#RadioButtonT.radio_button_o as 'c) option -> unit
  = fun c -> RadioButton.join_group obj (Option.map (fun z -> z#
   as_radio_button) c)
```

Listing 4.28: Esempio di metodo che usa Option.map

4.4.5 Costruttori

Dopo aver definito le classi relative ai segnali e proprietà/metodi viene generata una classe per unirle. Questa classe ha il nome dell'oggetto in esame, fa `inherit` della classe `_skel` ed implementa i segnali all'interno di un metodo `connect`. Questa classe viene messa in ricorsione tramite la keyword `and` con quella `_skel` poichè quest'ultima può contenere dei metodi che fanno una `new` della classe completa.

```
and button obj = object (self)
  inherit button_skel obj
  method connect = new button_signals obj
end
```

Listing 4.29: Esempio di classe finale dell'oggetto (GtkButton)

Agli utenti della libreria tuttavia non viene esposta questa classe appena definita, ma per creare un oggetto viene generata una funzione che da qui in avanti verrà chiamata "costruttore". Questo approccio è lo stesso usato in lablgtk, dove il costruttore riceve in input dei parametri opzionali. Questi parametri sono tutte le proprietà della classe aventi l'attributo `write`. In questo modo è possibile inizializzare l'oggetto con delle proprietà definite nel costruttore. Il costruttore deve però ricevere in input non solo le proprietà della classe stessa, ma anche quelle delle superclassi.

Per ottenere questo meccanismo, nel file di basso livello viene definita una funzione `make_params` che riceve in input una continuazione, una lista di parametri (provenienti dalle superclassi), e opzionalmente tutte le proprietà writable dell'oggetto corrente. Tramite la funzione `Gobject.Property.may_cons` di lablgtk viene creata una nuova lista di parametri contenente tutti i parametri correnti (le proprietà writable in aggiunta alla lista di parametri delle superclassi). Infine viene chiamata la continuazione con la nuova lista di parametri.

```
let make_params ~cont pl ?always_show_image ?image ?image_position ?
   label ?relief ?use_stock ?use_underline ?xalign ?yalign =
    let pl = (
        may_cons P.always_show_image always_show_image (
        may_cons P.image image (
        may_cons P.image_position image_position (
        may_cons P.label label (
```

```
7           may_cons P.relief relief (
8           may_cons P.use_stock use_stock (
9           may_cons P.use_underline use_underline (
10          may_cons P.xalign xalign (
11          may_cons P.yalign yalign pl)))))))))) in
12      cont pl
```

Listing 4.30: Esempio di make_params (GtkButton)

In aggiunta alle proprietà aventi l'attributo `write`, bisognerebbe inizializzare in fase di costruzione dell'oggetto anche le proprietà con l'attributo `construct-only`. Questo attributo indica che la proprietà può essere inizializzata solamente in fase di costruzione dell'oggetto. Tuttavia l'implementazione dei costruttori nel codice attualmente generato da ocaml-gi-gtk non permette di inizializzare queste proprietà. Fortunatamente, come sarà spiegato in una prossima sottosezione 4.4.5, esistono dei particolari metodi che vengono marchiati come costruttori, che verranno usati per generare dei costruttori addizionali con cui è possibile inizializzare queste proprietà.

Sempre nel file di basso livello viene definita un'ulteriore funzione `create`. Questa è la funzione che si occupa di prendere in input la lista di parametri del costruttore e di allocare in memoria una nuova istanza dell'oggetto. Questo viene fatto attraverso l'uso della funzione `GtkObject.make` di lablgtk che prende in input il nome della classe da istanziare e la lista di proprietà con cui inizializzare l'oggetto.

```
1 let create pl : ButtonT.t Gobject.obj = GtkObject.make "GtkButton" pl
```

Listing 4.31: Esempio di create (GtkButton)

Creazione di oggetti in lablgtk e floating references

La funzione `GtkObject.make` nasconde al suo interno un meccanismo di cooperazione con il garbage collector di GObject. Internamente istanzia gli oggetti attraverso il metodo `g_object_new` di GObject. Tuttavia alcuni GObject sono marchiati come GInitiallyUnowned e questo significa che possiedono una floating reference, ovvero non vengono considerati come posseduti da nessuno per motivi di convenienza per i programmatori C. Per eliminare questa floating reference ed incrementare il counter di riferimenti

nel garbage collector di GObject bisogna chiamare il metodo `g_object_ref_sink`, e questo viene fatto da `GtkObject.make` dopo la creazione dell'oggetto. l'implementazione di `g_object_ref_sink` prevede un controllo per verificare che l'oggetto su cui sta operando sia un GInitiallyUnowned e nel caso, rimuove la floating reference. Dopodiché, in tutti i casi, incrementa il reference count dell'oggetto. Va notato che nei binding generati `g_object_ref_sink` viene eseguito anche per i GObject non GInitiallyUnowned. Seppure questa operazione sia inefficiente, la perdita di performance è trascurabile soprattutto nel contesto di binding che per loro natura introducono già overhead dovuto alla conversione dei tipi.

Generazione del costruttore

La funzione `make_params` viene usata per definire il costruttore all'interno del file di alto livello. Partendo dalla superclasse alla base della gerarchia viene chiamata la sua funzione `make_params`, passando inizialmente una lista di parametri vuota. Come continuazione viene passata una funzione anonima che prende in input la lista di parametri `pl` e chiama la `make_params` della prossima classe nella gerarchia. Si procede in questo modo finchè non viene chiamata la `make_params` dell'oggetto corrente, a quel punto come sua continuazione deve essere passata una funzione che crei realmente l'oggetto. Questa funzione deve avere un valore `()` (ovvero `unit`) come ultimo parametro e nel caso in cui l'oggetto corrente erediti da Widget viene implementato ad hoc un meccanismo di lablgtk, ovvero la presenza dei parametri `?packing` e `?show`. Il parametro `packing` è di tipo `current_obj -> unit` e viene usato per aggiungere l'oggetto appena creato ad un container. Il parametro `show` invece viene settato a `true` se si vuole che il widget appena creato sia subito visibile. Questi due parametri vengono passati e gestiti dalla funzione `GObj.pack_return`. Infine viene fatta una `new` della classe relativa all'oggetto corrente a cui viene passato come valore di `obj` l'istanza dell'oggetto restituita dalla funzione `create` del modulo di basso livello.

```
1 let button = begin
2   GtkBase.Widget.size_params [] ~cont:(
3     fun pl -> Container.make_params pl ~cont:(
4       fun pl -> Bin.make_params pl ~cont:(
5         fun pl -> Button.make_params pl ~cont:(
```

```
6           fun pl ?packing ?show () -> GObj.pack_return (
7             new button (Button.create pl))
8             ~packing ~show))))
9 end
```

Listing 4.32: Esempio di costruttore (GtkButton)

Costruttori aggiuntivi

Alcune classi possiedono dei metodi che vengono marchiati come costruttori. Questi sono dei costruttori a tutti gli effetti che prendono in input dei valori particolari con cui inizializzare l'oggetto. La generazione di questi metodi segue la stessa procedura dei normali metodi nel file di basso livello. Nel file di alto livello invece non vengono generati come metodi standard ma come costruttori. Viene seguita la stessa procedura usata per i costruttori normali, ma hanno in aggiunta dei parametri posizionali a seconda di quanti parametri ha il metodo e nel corpo della funzione più interna l'oggetto non viene allocato con `GtkObject.make` ma tramite il metodo generato nel file di basso livello. Non essendo stati allocati tramite `GtkObject.make`, questi oggetti possono avere ancora una floating reference, quindi viene chiamata la funzione `GtkObject._ref_sink`. Per settare i valori iniziali delle proprietà nella lista di parametri viene chiamata la funzione `Gobject.set_params`.

```
1  let new_from_icon_name icon_name size = begin
2    GtkBase.Widget.size_params [] ~cont:(
3      fun pl -> Container.make_params pl ~cont:(
4        fun pl -> Bin.make_params pl ~cont:(
5          fun pl -> Button.make_params pl ~cont:(
6            fun pl ?packing ?show () -> GObj.pack_return (
7              let o = Button.new_from_icon_name icon_name size in
8              GtkObject._ref_sink o;
9              Gobject.set_params o pl;
10             new button o)
11             ~packing ~show))))
12 end
```

Listing 4.33: Esempio di costruttore aggiuntivo (GtkButton)

4.4.6 Ereditarietà

Le classi di GTK prevedono di poter ereditare da una superclasse, ottenendo i segnali, le proprietà ed i metodi di essa. Anche i binding devono rispettare questa struttura dove tutti gli oggetti che vengono generati ereditano almeno da GObject. Come visto precedentemente, per ogni oggetto vengono generate nel file *G.ml due classi separate, una per i segnali, ed una per proprietà e metodi. Entrambe queste classi devono pertanto ereditare dalla propria superclasse. Il meccanismo di OCaml usato per implementare l'ereditarietà di GTK è `inherit`, tramite esso vengono incluse nella sottoclasse tutte le funzionalità della superclasse. Dall'API del parser si ottiene dunque il Name della superclasse e da questa si deve ottenere il nome della classe OCaml corrispondente. Di questa operazione se ne occupa la funzione `nsOCamlClass`, che tiene conto del namespace e del modulo corrente per restituire il nome OCaml appropriato. Nella versione attuale della libreria ci si è concentrati sulla generazione per i binding di GTK senza considerare del tutto le librerie sottostanti. Per questo motivo se la superclasse proviene da un'altra libreria, questa viene ignorata e viene ereditata `GObj.gtkobj` da gilablgtk3 (`GObj.gobject_signals` rispettivamente per la classe dei segnali). Nel repository GitHub è presente una branch `complete-libs-gen` che rimuove questa limitazione generando completamente tutte le librerie, ma al momento della scrittura non è ancora stabile.

```
1  -- TODO: The default case should be the commented one but it makes
       sense only when every lib can be generated, otherwise the parent
       class is not available. Then the (Name "Gtk" _) case can be removed
2  case parents of
3    []           -> return ()
4    (parent : _) -> do
5      parentClass <- nsOCamlClass parent
6      let parentSkelClass = case parent of
7            Name "Gtk"       "Widget" -> "['a] GObj.widget_impl"
8            Name "GObject"   "Object" -> "GObj.gtkobj"
9            Name "Gtk"       _        -> parentClass <> "_skel"
10           _                         -> "GObj.gtkobj"
11           -- _                       -> parentClass <> "_skel"
12     gline $ "  inherit " <> parentSkelClass <> " obj"
```

Listing 4.34: Estratto di codice da Object.hs per la gestione dell'ereditarietà

Un'altra eccezione è presente per le classi che ereditano da GtkWidget. La classe `widget` di lablgtk3, essendo scritta a mano, ha delle shortcut per gestire gli eventi di drag&drop e quelli legati al cut&paste che non sarebbero disponibili tramite la generazione automatica. Per evitare di rompere troppo la compatibilità con le applicazioni scritte in lablgtk3 è stato scelto quindi di ereditare da `widget` di lablgtk3 invece che dalla classe automaticamente generata. Anche in questo caso è presente una branch `auto-widget` in cui viene rimossa questa eccezione. Il codice generato in questa branch è funzionante ma non è ancora stata integrato poiché va valutato l'impatto che questa scelta ha sulla difficoltà di effettuare porting di applicazioni scritte usando lablgtk.

4.4.7 Implementazione di interfacce

Gli oggetti di GTK possono implementare delle interfacce. La generazione delle interfacce verrà trattata in seguito, ma segue sostanzialmente lo stesso schema degli oggetti generando una classe per i segnali ed una classe per proprietà e metodi. Le interfacce di GObject funzionano come le regolari interfacce nella programmazione orientata agli oggetti, ovvero definiscono dei metodi (ed anche segnali e proprietà) senza una implementazione, che sarà fornita dalla classe che la implementa. Ulteriori dettagli saranno trattati nella sezione 4.5. Per implementare le interfacce una prima scelta è stata quella di fare un `inherit` della classe dell'interfaccia all'interno della classe che la implementa. Tuttavia questa soluzione portava in certi casi ad avere un clash di nomi tra i metodi della classe ed i metodi dell'interfaccia nelle classi `_skel`. Per ovviare a questo problema è stato deciso di implementare le interfacce all'interno di un metodo della classe. In particolare il nome del metodo è sempre preceduto dalla lettera `i`.

```
class button_skel obj = object (self)
  ...
  method iimplementor_iface = new GIAtk.ImplementorIfaceG.
   implementor_iface_skel obj
  method iactionable = new ActionableG.actionable_skel obj
  method iactivatable = new ActivatableG.activatable_skel obj
  method ibuildable = new BuildableG.buildable_skel obj
  ...
```

Listing 4.35: Esempio di implementazione delle interfacce (GtkButton)

Nelle classi dei segnali invece questo problema non è presente ed è stato deciso di mantenere l'`inherit` in quanto l'alternativa per rendere l'API uniforme sarebbe stata quella di inserire anche in questo caso l'implementazione dell'interfaccia all'interno di un metodo, che però dovrebbe avere un nome diverso da quello usato nelle classi `_skel`, portando così ad una eccessiva frammentazione dell'API.

4.4.8 Dipendenze cicliche

Si fa riferimento ad una dipendenza ciclica quando quando un modulo A contiene dei riferimenti ad un modulo B e viceversa. Il compilatore di OCaml non supporta mutua ricorsione fra definizioni contenute in moduli distinti e fallisce la compilazione restituendo un messaggio di errore che indica i moduli dentro al ciclo.

```
Error: Dependency cycle between the following files:
   _build/default/Gtk/.GIGtk.objs/gIGtk__TextBufferG.impl.all-deps
-> _build/default/Gtk/.GIGtk.objs/gIGtk__TextMarkG.impl.all-deps
-> _build/default/Gtk/.GIGtk.objs/gIGtk__TextBufferG.impl.all-deps
Error: Dependency cycle between the following files:
   _build/default/Gtk/.GIGtk.objs/gIGtk__TreeSelectionG.impl.all-deps
-> _build/default/Gtk/.GIGtk.objs/gIGtk__TreeViewG.impl.all-deps
-> _build/default/Gtk/.GIGtk.objs/gIGtk__TreeSelectionG.impl.all-deps
```

Listing 4.36: Esempio di messaggio di errore restituito dal compilatore in presenza di cicli di dipendenze

Nel codice generato da ocaml-gi-gtk le dipendenze cicliche possono essere trovate in due occasioni:

1. annotazioni di tipo: un modulo A fa riferimento al tipo di un modulo B e viceversa. Concretamente questo problema viene riscontrato nell'annotazione di tipo dei metodi, dove i tipi degli oggetti in input/output vengono esplicitati;

2. uso di funzioni: un modulo A usa una funzione del modulo B e viceversa. Concretamente questo problema viene riscontrato nei metodi che restituiscono un oggetto di un'altra classe. Nel corpo del metodo viene effettuata un'operazione di `new` di un'altra classe.

Il primo tipo di problema è stato gestito spostando le dichiarazioni di tipo in un modulo separato. In questo modo sia il modulo A che il modulo B fanno riferimento a dei moduli AT e BT, con AT e BT privi di mutue dipendenze e viene spezzato il ciclo. I moduli contenenti le dichiarazioni di tipo sono l moduli *T.ml. Il secondo tipo di problema è invece più complicato da gestire. È innanzitutto problematico trovare quali sono i moduli che creeranno un ciclo prima di aver generato il codice. Per farlo occorrerebbe creare un grafo dove i nodi sono le classi e gli archi sono i riferimenti ad un'altra classe e fare uso di un algoritmo per l'individuazione di cicli come l'algoritmo di Tarjan. Tuttavia anche se questa sarebbe la scelta più "corretta", l'implementazione di questo algoritmo sarebbe più complicato della soluzione attualmente adottata, ovvero quella di effettuare il parsing dell'outuput del compilatore dopo una prima generazione del codice. Per farlo viene scansionata ogni riga dell'output alla ricerca di un pattern `__*.impl`, ottenendo così nomi dei file dentro il ciclo. Per risolvere la ricorsione, le classi dei file dentro al ciclo vengono vengono spostate all'interno di un file chiamato Recursion.ml. Le classi dei segnali non formano mai dei cicli, quindi queste vengono semplicemente riportate così come sono all'inizio del file. Le altre classi (`name_skel` e `name`) invece contengono il ciclo di dipendenze e pertanto devono essere dichiarate come mutualmente ricorsive tramite l'operatore `and`, che deve essere sostituito alla keyword `class` per tutte le dichiarazioni di classi tranne la prima. Anche le occorrenze di `NameG.` devono essere eliminate in quanto ora questi moduli si trovano dentro a Recursion.ml. Infine in fondo al file vengono ripotate le dichiarazioni dei costruttori. I file *G.ml il cui contenuto è stato spostato in Recursion.ml devono ora riportare delle dichiarazioni di classi che "puntino" a quelle corrispondenti in Recursion.ml.

```
class text_buffer_signals obj = object (self) ... end
class text_mark_signals obj = object (self) ... end
class text_buffer_skel obj = object (self) ... end
and text_buffer obj = object (self) ... end
and text_mark_skel obj = object (self) ... end
and text_mark obj = object (self) ... end
let text_buffer = begin ... end
let text_mark = begin ... end
```

Listing 4.37: Esempio di Recursion.ml

```
class text_buffer_signals = Recursion.text_buffer_signals
class text_buffer_skel = Recursion.text_buffer_skel
class text_buffer = Recursion.text_buffer
let text_buffer = Recursion.text_buffer
```

Listing 4.38: Esempio di file *G.ml dopo essere stato "spostato" in Recursion.ml

4.5 Interfacce

In C le interfacce vengono dichiarate come un sottotipo di GInterface e dichiarano una struct S contenente i puntatori ai diversi metodi che devono essere implementati dalle classi che implementano l'interfaccia. Per ogni metodo vengono dichiarate inoltre delle funzioni f_m che ricevono in input un'istanza dell'oggetto che implementa l'interfaccia ed estraggono la struct S contenente i puntatori alle implementazioni dei metodi. Nel caso delle interfacce, le funzioni per le quali vanno creati i binding sono dunque le funzioni f_m, che chiameranno dinamicamente l'implementazione corretta del metodo.

```
struct _ViewerEditableInterface
{
  GTypeInterface parent_iface;

  void (*save) (ViewerEditable  *self,
                GError         **error);
};

...
```

```
10
11 void
12 viewer_editable_save (ViewerEditable  *self,
13                        GError         **error)
14 {
15   ViewerEditableInterface *iface;
16
17   g_return_if_fail (VIEWER_IS_EDITABLE (self));
18   g_return_if_fail (error == NULL || *error == NULL);
19
20   iface = VIEWER_EDITABLE_GET_IFACE (self);
21   g_return_if_fail (iface->save != NULL);
22   iface->save (self, error);
23 }
```

Listing 4.39: Esempio di interfaccia Editable che dichiara il metodo save

La generazione delle interfacce è del tutto analoga a quella degli oggetti per quanto riguarda i file *.c e *.ml di basso livello. La generazione del file di alto livello *G.ml invece differisce per alcuni aspetti:

- la classe relativa ai segnali viene generata come una classe virtuale, dichiarando `connect` come metodo virtuale;
- le interfacce non sono istanziabili e pertanto non viene generata la classe che unisce la classe dei segnali con la classe `_skel`, e non vengono generati nemmeno i costruttori.

```
1 class virtual cell_editable_signals obj = object (self)
2   method private virtual connect : 'b. ('a,'b) GtkSignal.t -> callback
     :'b -> GtkSignal.id
3   method editing_done = self#connect CellEditable.S.editing_done
4   method remove_widget = self#connect CellEditable.S.remove_widget
5 end
6
7 class cell_editable_skel obj = object (self)
8   method as_cell_editable = (obj :> CellEditableT.t Gobject.obj)
9
```

```
10   method editing_canceled = Gobject.get CellEditable.P.editing_canceled
       obj
11   method set_editing_canceled = Gobject.set CellEditable.P.
      editing_canceled obj
12
13   method editing_done : unit =
14     CellEditable.editing_done obj
15
16   method remove_widget : unit =
17     CellEditable.remove_widget obj
18 end
```

Listing 4.40: Esempio di interfaccia generata nel file di alto livello

Capitolo 5

Valutazione

Per valutare il lavoro fatto sono state effettuate due tipi di analisi, una quantitativa per capire in che percentuale è stata coperta l'API di GTK, ed una qualitativa per capire se i binding generati sono effettivamente usabili per scrivere applicazioni. Tuttavia, prima di scendere nel dettaglio delle valutazioni, occorre parlare dei limiti dell'attuale implementazione di ocaml-gi-gtk e di come essi possano essere risolti in lavori futuri.

5.1 Limitazioni e lavori futuri

Per quanto riguarda la generazione di GTK, il limite principale riguarda i tipi di dato che non sono stati generati o di cui non è stata effettuata una conversione tra C ed OCaml. Come visto nella sezione 4.1.1, la funzione `genAPI` non è implementata per alcuni tipi di API, e per questi tipi non viene generato codice. In aggiunta a questi, i tipi struct e union non vengono effettivamente generati, ma viene solo aggiunto il file contenente le dichiarazioni di tipo, che attualmente contengono un tipo opaco. La mancanza di questi tipi porta le funzioni di conversione di Conversions.hs a fallire nella generazione di metodi, proprietà e segnali, lasciando un commento che descrive l'errore incontrato. Inoltre, anche per altri tipi di dato di GLib che vengono usati da metodi, proprietà e segnali non è stata effettuata una conversione. Tra questi troviamo: GType, gptr, gintptr, guintptr, GError, GVariant, GParamSpec, GArray, GPtrArray, GByteArray, GList, GSList, GHash e GClosure. Infine, non vengono generati i metodi che hanno parametri

di input/output ed i metodi che lanciano eccezioni. I metodi che hanno come parametro di output un oggetto vengono invece attualmente generati, ma l'implementazione è errata. Fortunatamente, questo tipo di metodi non è presente in GTK, ma è presente nelle librerie di cui GTK fa uso.

Inoltre, alcuni metodi operano su tipi di dato provenienti da altre librerie (es: GDK, Pango, Cairo). La generazione di questi metodi è stata effettuata poichè sarebbe possibile, tramite una libreria scritta a mano, fornire un'implementazione di questi tipi e rendere effettivamente usabili questi metodi. Tuttavia attualmente, le librerie da cui GTK dipende non sono completamente generate, ma vengono generate solo le informazioni relative ai loro tipi. Un lavoro riguardante la generazione completa di queste librerie è già stato avviato nel branch `complete-libs-gen`, ma deve essere completato.

Esiste anche un problema legato al meccanismo di late binding, che non viene implementato correttamente né dai binding generati né da quelli di lablgtk e possibilmente anche dai binding di altri linguaggi. Infatti se consideriamo una classe B che eredita da una classe A e fa overloading di un metodo f_A (che denoteremo f_B), i metodi scritti in OCaml chiameranno a runtime correttamente f_B, ma i metodi scritti in C continueranno a invocare f_A. Una soluzione sarebbe quella di generare una classe B' anche in C, la quale fa overloading del metodo f_A richiamando un metodo OCaml che richiama il metodo C originale. Tale soluzione sarebbe tuttavia complessa ed inefficiente.

Infine, un limite non strettamente tecnico di ocaml-gi-gtk è la sua implementazione in Haskell. Questa scelta ha accelerato di molto lo sviluppo di questo prototipo, ma se questo ha l'obiettivo di diventare un tool realmente usato e supportato dalla community di OCaml, allora una conversione in OCaml è probabilmente necessaria.

5.2 Valutazione quantitativa

Per fare una valutazione quantitativa del lavoro fatto, è stato valutato quanti metodi, segnali e proprietà di GTK vengono generati da ocaml-gi-gtk, confrontandoli con il totale e con le quantità di lablgtk3.

	Totale	Generati	Non generati	Generati da lablgtk3
Metodi	2117	1568 (74%)	549 (26%)	878 (41%)
Segnali	380	372 (98%)	8 (2%)	143 (38%)
Proprietà	1046	1039 (99%)	7 (1%)	563 (54%)
Totale	3543	2979 (84%)	564 (16%)	1586 (45%)

Tabella 5.1: Valutazione quantitativa dei binding generati

Questi valori mostrano una buona copertura (84%) della libreria GTK. Questo valore risulta particolarmente positivo soprattutto se confrontato con la copertura di lablgtk3, che si attesta al 45%.

Le quantità sono state ottenute eseguendo i seguenti comandi shell, essendo situati nelle directory `ocaml-gi-gtk/bindings/Gtk/Gtk` e `lablgtk/_build/default/src`.

- metodi non generati: `grep "Could not generate method" *.ml | wc -l`
- metodi generati: `grep "external " *.ml | wc -l`
- metodi generati da lablgtk3: `grep "external " gtk*.ml | wc -l`
- segnali non generati: `grep "Could not generate signal" *.ml | wc -l`
- segnali generati: `grep " = {name=" *.ml | wc -l`
- segnali generati da lablgtk3: `grep " = {name=" gtk*.ml | wc -l`
- proprietà non generate: `grep "Could not generate signal" *.ml | wc -l`
- proprietà generate: `grep " property =" *.ml | wc -l`
- proprietà generate da lablgtk3: `grep " property =" gtk*.ml | wc -l`

5.3 Valutazione qualitativa

Per testare il corretto funzionamento dei binding generati da ocaml-gi-gtk sono stati creati degli esempi di piccole applicazioni che ne testano diversi widget e funzionalità.

Per guidare questo processo, ed anche per ottenere un confronto tra le funzionalità delle librerie generate da ocaml-gi-gtk e tra i bindings scritti a mano di lablgtk, è stato effettuato il porting dei file di esempio contenuti nel repository di lablgtk. Su 49 file di cui è stato eseguito il porting, 37 sono risultati eseguibili con successo, mentre 12 sono stati ritenuti troppo difficili da convertire nei seguenti casi:

- il file di esempio fa uso di metodi/proprietà/segnali di cui è fallita la generazione, pertanto non è possibile procedere con il porting;
- l'esempio di lablgtk faceva uso di funzionalità custom scritte a mano in lablgtk e non presenti nella libreria C. Per convertire questi esempi occorrerebbe creare una libreria scritta a mano che reimplementi queste funzionalità per il codice generato da ocaml-gi-gtk. Tuttavia tale libreria sarebbe sensibile ai cambi di versione di GTK, e va valutato se i benefici portati da questa libreria sono tali da giustificarne la manutenzione.

Conclusioni

In questo lavoro di tesi è stato messo in discussione l'attuale metodo per la scrittura di applicazioni GTK nel linguaggio OCaml, attualmente sviluppate tramite l'uso di binding scritti a mano nella libreria lablgtk. È stato discusso come tale approccio non sia sostenibile a causa della necessità di troppo lavoro manuale per la manutenzione della libreria. Un approccio più scalabile ed in linea con quanto fatto per altri linguaggi di programmazione è lo sviluppo di una libreria in grado di generare automaticamente binding per GTK, ottenendo come conseguenza anche la possibilità di generare binding per le librerie su cui GTK è basata, per le librerie che definiscono widget aggiuntivi ed in generale per tutte le librerie basate sul sistema di tipi di GLib.

Il contributo di questa tesi è stato proprio lo sviluppo di una tale libreria in grado di coniugare le principali idee di lablgtk con la generazione automatica di binding, ottenendo così una libreria per GTK avente una API il più idiomatica possibile per i programmatori OCaml, facilitando anche la conversione di applicazioni lablgtk già esistenti.

Sono stati ottenuti risultati più che soddisfacenti, riuscendo a generare l'84% dei binding totali di GTK (facendo riferimento a metodi, segnali e proprietà), ottenendo una copertura della libreria maggiore del 39% rispetto a quella di lablgtk3. È stato possibile inoltre convertire il 76% delle applicazioni di esempio di lablgtk3, dimostrando il corretto funzionamento dei binding generati.

Tuttavia, come reso evidente da questi numeri, la copertura di GTK non è ancora totale e ocaml-gi-gtk, la libreria sviluppata in questa tesi, necessita di ulteriore lavoro per la generazione di strutture dati di GLib che al momento non sono state implementate. Infine, è giusto notare che per accelerare i tempi di sviluppo di un prototipo, l'implementazione attuale di ocaml-gi-gtk è nel linguaggio Haskell, una scelta che potrebbe portare

la community a non supportare un'ulteriore sviluppo di questa libreria, rendendo una conversione nel linguaggio OCaml di fatto necessaria.

Bibliografia

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The matita interactive theorem prover. In *International Conference on Automated Deduction*, pages 64–69. Springer, 2011.

[2] Dune - a composable build system for ocaml, 2020. [Online; accessed March, 2020].

[3] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, volume 13, page 7. Baltimore, 1998.

[4] Gobject introspection's annotations, 2020. [Online; accessed March, 2020].

[5] Gimp - gnu image manipulation program. `https://www.gimp.org/`.

[6] gjs - gnome javascript bindings, 2020. [Online; accessed March, 2020].

[7] Glib basic types. `https://developer.gnome.org/glib/stable/glib-Basic-Types.html`.

[8] Gtk3 - object hierarchy, 2020. [Online; accessed March, 2020].

[9] Gobject introspection. `https://gi.readthedocs.io/en/latest/`.

[10] Gtk. `https://www.gtk.org/`.

[11] Gtk architecture, 2020. [Online; accessed March, 2020].

[12] gtk-rs. `https://github.com/gtk-rs/gtk`.

[13] Glasgow Haskell Compiler User's Guide - foreign function interface. `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ffi-chap.html`.

[14] haskell-gi. `https://github.com/haskell-gi/haskell-gi`.

[15] Lablgtk, 2020. [Online; accessed March, 2020].

[16] Ocaml manual - foreign function interface. `https://caml.inria.fr/pub/docs/manual-ocaml/intfc.html`.

[17] ocaml-gi-gtk. `https://github.com/illbexyz/ocaml-gi-gtk/`.

[18] Pygobject, 2020. [Online; accessed March, 2020].

[19] Didier Rémy and Jérôme Vouillon. Objective ml: An effective object-oriented extension to ml. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[20] Stack. `https://docs.haskellstack.org/en/stable/README/`.

[21] Gobject - subclass gobject, 2020. [Online; accessed March, 2020].

[22] Sébastien Wilmet. The glib/gtk+ development platform. 2019.

Ringraziamenti

- Desidero ringraziare il Prof. Claudio Sacerdoti Coen per avermi proposto questa tesi e per averne seguito lo sviluppo con interesse ed estrema disponibilità, guidandomi attraverso gli oscuri errori del compilatore di OCaml e l'intricato sistema a oggetti di GLib.
- Ringrazio la mia famiglia, mia madre e Giovanni per avermi permesso di completare questo lungo percorso di studi.
- Ringrazio i miei amici ed in particolare le fantastiche persone che ho conosciuto durante il periodo universitario.
- Un ringraziamento speciale va infine a Devid Farinelli, per essere stato un compagno di mille avventure informatiche e non da oltre un decennio.