Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica Magistrale

Conversione in OCaml della libreria ocaml-gi-gtk

Relatore:
Chiar.mo Prof.
Claudio Sacerdoti Coen

Presentata da:
Alberto Drusiani

Sessione straordinaria
2019/2020

Abstract

L'utilizzo in larga scala della libreria grafica GTK implementata in C ha forzato gli sviluppatori della stessa a rendere semplice la creazione di binding ad essa con linguaggi di alto livello. Esistono due tipi di binding: manuale e automatico. Il binding automatico permette di svincolarsi dalle modifiche che vengono effettuate a GTK, rendendo il processo di binding più robusto. OCaml, linguaggio di programmazione emergente e multi-paradigma, non possiede una libreria di binding automatici per GTK scritta in OCaml, ma una libreria scritta in Haskell chiamata *ocaml-gi-gtk* e sviluppata all'interno dell'Università di Bologna.
Questo progetto ha lo scopo di convertire in OCaml questa libreria, in modo che possa essere addottata dalla community del linguaggio. È spesso infatti presente una sorta di tendenza all'autarchia nel mondo dei linguaggi di programmazione, che rende riluttanti gli utilizzatori di un linguaggio nei confronti di uno simile e "concorrente".

Indice

1 Introduzione **1**
1.1 Language binding 1

2 Background **3**
2.1 GTK 3
2.1.1 GObject-Introspection 5
2.2 OCaml 6
2.2.1 Features caratterizzanti 6
2.3 Haskell 11
2.3.1 Monadi 11
2.3.2 haskell-gi 12
2.4 ocaml-gi-gtk 12
2.4.1 Binding GTK per OCaml 13

3 Impostazione del progetto **17**
3.1 Struttura di ocaml-gi-gtk 17
3.1.1 Implementazione delle monadi 18
3.1.2 Routine di alto livello 20
3.1.3 Routine di basso livello 23
3.2 Tentativo di parsing 28

4 Implementazione **33**
4.1 Scelte implementative 33
4.1.1 Libreria di parsing XML 33

4.1.2 Sintassi e costrutti OCaml 35
4.1.3 Conversione delle monadi 37
4.2 Parsing 37
4.3 Generazione di codice 43

5 Risultati **49**

Conclusione **51**

Listings

2.1 Utilizzo del comando `open` in OCaml . . . 6
2.2 Creazione di un funtore . . . 7
2.3 Definizione di varianti in OCaml . . . 8
2.4 Definizione di varianti polimorfe in OCaml . . . 9
2.5 Utilizzo di variante polimorfa . . . 9
3.1 Esempio di comandi di override . . . 18
3.2 Monade per la generazione di codice . . . 19
3.3 Monade per la scrittura su file . . . 19
3.4 Funzione `genAPI` Haskell . . . 21
3.5 Tipo di dato `API` . . . 22
3.6 Tipo di dato `GIRInfo` . . . 22
3.7 Funzione `parseNSElement` Haskell . . . 23
3.8 Tipo di dato `Code` e `CodeToken` . . . 24
3.9 Tipo di dato `ModuleInfo` . . . 24
3.10 Funzioni per modificare lo stato `ModuleInfo` . . . 25
3.11 Tipo di dato `CGState` . . . 26
3.12 Creazione variabile fresca . . . 26
3.13 Conversioni tipi di bassi livello . . . 27
3.14 Parsing di costanti in Haskell . . . 29
3.15 Parsing di costanti in OCaml . . . 30
4.1 Elementi XML in Haskell [1] . . . 34
4.2 Tipo di dato XML in xml-light . . . 35
4.3 Differenza tra stili di naming . . . 36

4.4 Estratto da GLib.gir 38
4.5 Rappresentazione in xml-light del frammento .gir 38
4.6 Tipo `name` in OCaml 39
4.7 Gestione dei nomi qualificati 39
4.8 Monade Haskell per il parsing 40
4.9 Codice monadico nel parsing XML 41
4.10 Codice OCaml equivalente nel parsing XML 41
4.11 Dichiarazione dei moduli/tipi per mappe e insiemi 42
4.12 Funzioni base per la valutazione del codice monadico dei generatori 43
4.13 Funzione equivalente `evalCodeGen` in OCaml 44
4.14 Lancio di computazioni lazy su stati nuovi in Haskell 45
4.15 Lancio di computazioni lazy su stati nuovi in OCaml 45
4.16 Gestione monadica di errori non bloccanti 46
4.17 Gestione tramite chiusure di errori non bloccanti 46

Capitolo 1

Introduzione

Al giorno d'oggi vengono largamente utilizzati, sia in ambito accademico che commerciale, linguaggi di programmazione di alto livello. Questo rende possibile sviluppare applicativi più velocemente ed evita di occuparsi di tutti gli aspetti di basso livello, come l'interfacciarsi con il sistema operativo e la gestione esplicita della memoria, meccanismi invece necessari in linguaggi come C. Nello sviluppo di un applicativo è spesso richiesto di interfacciarsi con librerie scritte in altri linguaggi (tendenzialmente di più basso livello) per evitare di riscrivere codice o per accedere a funzionalità di basso livello che non sono supportate dal linguaggio. È quindi necessario un meccanismo che permetta in qualche modo di "collegare" un linguaggio di alto livello e un linguaggio di basso livello, meccanismo che viene chiamato "language binding".

1.1 Language binding

In questo contesto, con il termine binding si intende una tecnica che permette di costruire del "codice collante" tra due linguaggi/librerie in modo tale da poter sfruttare all'interno del linguaggio di alto livello tutte le funzionalità di basso livello senza perdere il proprio valore idiomatico. In questa tesi verrà trattato il binding che riguarda la libreria GTK, ampiamente supportata ed utilizzata per creare interfacce grafiche. La libreria GTK è stata scritta in C, quindi è necessario implementare dei binding per poterla sfruttare a partire da altri linguaggi di programmazione.

Capitolo 2

Background

2.1 GTK

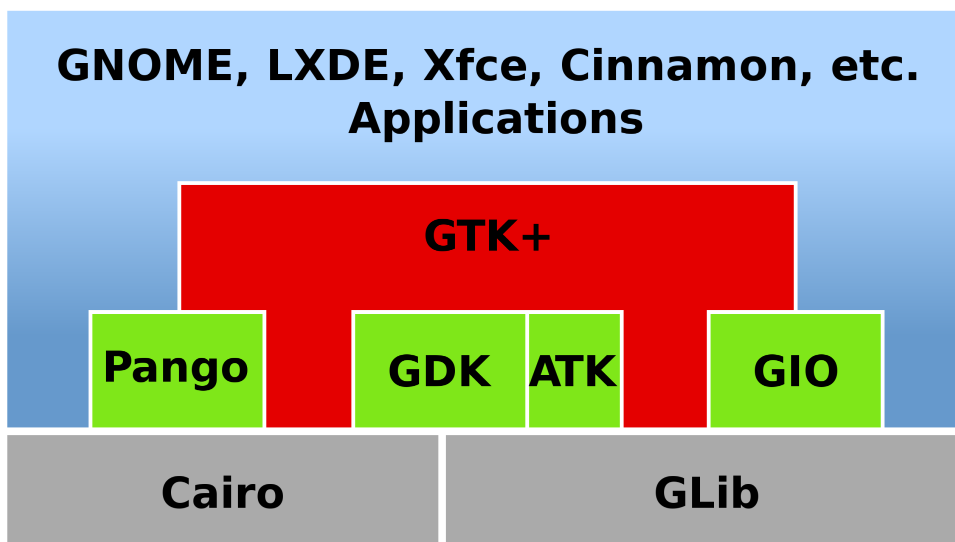

Figura 2.1: Architettura semplificata di GTK

GIMP Toolkit, noto come GTK, è una libreria open-source sotto licenza LGPL, inizialmente sviluppata all'interno di GIMP [2] e attualmente una delle librerie più utilizzate per lo sviluppo di interfacce grafiche, tendenzialmente in ambiente Linux. GTK è strutturata in maniera gerarchica [3] come mostrato in figura 2.1 ed è formata principalmente dalle seguenti librerie:

- **GLib**: implementa strutture dati come liste, hashtable, alberi e uilities varie
- **GObject**: implementa oggetti scritti in C ed è stata progettata per permettere facilmente l'implementazione di binding in altri linguaggi
- **GIO**: implementa ed espone API di alto livello per la gestione di input/output
- **ATK**: aggiunge la gestione dell'accessibilità, in modo da permettere agli sviluppatori di rendere i propri programmi utilizzabili da chiunque
- **cairo**: libreria grafica di basso livello, espone primitive per la creazione di grafica vettoriale a due dimensioni
- **Pango**: libreria per il rendering di testo multi-linguaggio
- **GDK**: fornisce l'astrazione per le librerie sottostanti, in modo da esporre a GTK una serie di API di più alto livello
- **GTK**: definisce gli oggetti per creare e interagire con le interfacce

L'intera libreria GTK è basata su reference counting, dove ogni oggetto dev'essere allocato e deallocato in maniera scrupolosa per evitare memory leaks o crash. Ogni volta che viene creato un oggetto viene incrementato il suo reference count di 1 e il numero di reference che ha un oggetto indica il numero di owners dell'oggetto stesso. Esistono due tipi di oggetti che sfruttano il meccanismo di reference counting in maniera differente [4]:

- Discendenti diretti di `GObject`: al momento della creazione dell'oggetto viene settato il reference count a 1.
- Discendenti di `InitiallyUnowned`: al momento della creazione dell'oggetto, viene settato il reference count a 1 ma con un valore speciale, chiamato *floating reference count*.

I primi vengono gestiti in maniera classica, incrementandone il reference count ogniqualvolta viene aggiunto un owner e decrementandolo ogni volta che viene distrutto un owner. I secondi vengono creati con un floating reference count settato ad uno, indicando che l'oggetto esiste ma non ha ancora un owner. Una volta assegnato al primo owner,

non verrà incrementato a 2 il floating reference count ma verrà convertito in un reference count normale, rimanendo ad 1, per poi comportarsi in modalità standard fino alla deallocazione. Questa politica è stata adottata dagli implementatori di GTK per permettere la scrittura di codice con oggetti "anonimi" che vengono creati direttamente all'interno di altri oggetti.

2.1.1 GObject-Introspection

Come detto sopra, il largo uso di GTK ha portato alla creazione di binding per molti linguaggi di alto livello; per questo il processo di creazione dei binding è stato reso semplice dagli implementatori tramite un meccanismo chiamato GObject-Introspection. GObject-Introspection (GIR)[5] consiste nel creare automaticamente dei file `.gir` (uno speciale file XML) a partire da una libreria, dove vengono riportati tutti i dati della libreria stessa. In questo modo gli sviluppatori di binding hanno uno strumento estremamente leggibile e standardizzato per poter scrivere dei binding corretti e per poterlo fare in modo automatico. Esistono infatti due metodologie per scrivere dei binding: la prima consiste nel scrivere il codice collante a mano, guardando la libreria di partenza e scrivendo tutto il codice necessario per poter creare dei binding corretti per il linguaggio di alto livello; la seconda consiste nel creare una libreria che generi i binding automaticamente a partire dai file `.gir` creati dal meccanismo di GObject-Introspection. La prima soluzione risulta soggetta alle modifiche della libreria di basso livello, per cui se viene cambiato qualcosa, di conseguenza i binding devono essere modificati. Nonostante questo, la scrittura manuale tende a mantenere gli idiomi del linguaggio di alto livello. La seconda soluzione non necessita della manutenzione della prima in quanto, nel caso in cui venisse modificata la libreria di basso livello, automaticamente i file .gir verrebbero aggiornati e basterà ricompilare i binding per averne la versione aggiornata. D'altra parte, la generazione automatica di binding richiede più sforzo per ottenere un'integrazione naturale nell'ecosistema e nelle buone pratiche del linguaggio di programmazione. Ad esempio: il processo di binding di una libreria stateful in un linguaggio funzionale forza uno stile di programmazione imperativa che si vorrebbe evitare.

2.2 OCaml

OCaml è un linguaggio di programmazione multi-paradigma, sviluppato sulla base di ML [6]. Nasce con l'idea di mantenere la type-safety tipica dei linguaggi funzionali moderni senza rinunciare al paradigma imperativo e orientato agli oggetti. Viene rilasciato per la prima volta nel 1996 come linguaggio accademico e ad oggi viene utilizzato in diverse ambiti e soluzioni commerciali che necessitano di efficienza e stabilità.

2.2.1 Features caratterizzanti

OCaml spicca tra gli altri linguaggi per alcune feature interessanti che lo contraddistinguono, come il sistema di moduli estremamente espressivo, le varianti polimorfe, un garbage collector efficiente e un'alta portabilità data dalla compilazione in bytecode. Nelle prossime sotto-sezioni vengono approfonditi questi aspetti in modo da dare una panoramica generale.

Moduli

Il sistema di moduli di OCaml viene utilizzato per due scopi principali:

- Organizzazione del codice
- Implementazione di codice generico

L'organizzazione del codice viene gestita e semplificata attraverso i moduli, in quanto ogni file `prova.ml` viene automaticamente associato ad un modulo `Prova` che espone tutto ciò che è descritto all'interno del file `prova.ml`.

```
1 (* prova.ml *)
2 let somma a b =
3   a + b
4
5 (* main.ml senza open *)
6 let s = Prova.somma 3 4 in
7 print_endline s;;
8
9 (* main.ml con open *)
```

```
10 open Prova
11 let s = somma 3 4 in
12 print_endline s;;
```

Listing 2.1: Utilizzo del comando `open` in OCaml

Ad esempio, per poter utilizzare la funzione `somma` all'interno del file `prova.ml` basterà richiamare la funzione specificando il namespace del modulo e il nome della funzione in dot notation: `Prova.somma`. È anche possibile importare il nome non qualificato utilizzando la direttiva `open` in modo da chiamare `somma` senza specificare il namespace. È buona norma utilizzare la direttiva `open` solamente per i moduli che contengono funzioni il cui nome non coincide con altre, in modo da poter invocare la funzione senza dover scrivere ogni volta il nome qualificato. Solitamente vengono scritti anche dei file `.mli` di interfaccia, per descrivere dettagliatamente quali funzioni siano esposte al modulo e il loro tipo, in modo da vincolare il compilatore e semplificare la documentazione e la comprensione dei tipi.
L'aspetto più interessante dei moduli è invece la possibilità di strutturare il codice in maniera generica, principalmente mediante l'uso di *funtori*. I funtori non sono altro che funzioni che prendono in input e danno in output moduli e possono essere utilizzati per strutturare il codice in modo che sia il più generico e riutilizzabile possibile. Come per le funzioni, un modulo può avere un'interfaccia di cui si può scrivere un'implementazione attraverso il comando `module type`.

```
1 (* definizione di una signature per un modulo *)
2 module type Una_Stringa = sig val s : string end
3
4 (* dichiarazione di un funtore *)
5 module Esclama (S: Una_Stringa) : Una_Stringa = struct
6   let s = S.s ^ "!"
7 end
8
9 (* utilizzo di un funtore *)
10 module Ciao = struct let s = "Ciao" end;;
11 module Ciaooo = Esclama(Ciao);; (* Ciaoo diventa un modulo dove s = "
     Ciao!"*)
```

Listing 2.2: Creazione di un funtore

Un funtore si implementa dichiarando il tipo di modulo in input e in output e potrà essere usato su qualsiasi modulo la cui interfaccia soddisfi i tipi del funtore. Prendendo l'esempio sopra, si potrà dare in input al funtore qualsiasi modulo che contenga almeno un campo `x` di tipo `string`.
Questo meccanismo può essere utilizzato in OCaml per rendere il codice estremamente flessibile e astratto ed è uno dei pochi linguaggi che possiede un sistema di moduli così espressivo.

Varianti polimorfe

Come molti linguaggi di porgrammazione funzionali, OCaml fa un uso esteso delle varianti. Le varianti sono un tipo di dato algebrico (ADT) che esprime il concetto di unione disgiunta o co-prodotto. Gli ADT sono un insieme di tipi di dato divisibile in due macro-categorie: i prodotti, come le tuple e i record, e i coprodotti, le varianti. Mentre le tuple e i record esprimono il concetto di una moltitudine di dati presenti nello stesso momento (come i record in C), le varianti indicano un tipo di dato che può contenere al massimo uno dei valori con cui è stata dichiarata.

```
1  (* definizione di una variante *)
2  type sport =
3    | Calcio
4    | Pallavolo
5    | Basket
6  (* type sport = Calcio | Pallavolo | Basket *)
7
8  (* pattern matching sul tipo sport *)
9  let dimmi_lo_sport c =
10   match c with
11   | Calcio -> print_endline ("Pratico il calcio")
12   | Pallavolo -> print_endline ("Pratico la pallavolo")
13   | Basket -> print_endline ("Pratico il basket")
14 (*val dimmi_lo_sport : sport -> unit = <fun>*)
15
16 dimmi_lo_sport Pallavolo;;
17 (*Pratico la pallavolo - : unit = ()*)
```

Listing 2.3: Definizione di varianti in OCaml

Nell'esempio viene creato un tipo `sport` che può, ad esempio, codificare lo sport praticato da una persona. Il meccanismo utilizzato per sfruttare le varianti è il pattern-matching che permette di destrutturare il dato e leggerne il valore.
La peculiarità di OCaml è la presenza di un costrutto chiamato **varianti polimorfe** che, come suggerisce il nome, permette di aggiungere polimorfismo alle varianti e rende più flessibili le varianti "normali".

```
(* definizione di tre varianti polimorfe *)
let cinque = ‘Numero 5;;
(* val cinque : [> ‘Numero of int ] = ‘Numero 5 *)
let hey = ‘Stringa "hey";;
(* val hey : [> ‘Stringa of string ] = ‘Stringa "hey" *)
let pallavolo = ‘Sport Pallavolo
(* val pallavolo : [> ‘Sport of sport ] = ‘Sport Pallavolo *)

(* il compilatore riesce a costruire la lista di tipi diversi,
   inferendo un nuovo tipo che comprende tutti i tag *)
let lista_mista = [cinque; hey; pallavolo];;
(*val lista_mista :
  [> ‘Numero of int | ‘Sport of sport | ‘Stringa of string ] list =
  [‘Numero 5; ‘Stringa "hey"; ‘Sport Pallavolo]*)
```

Listing 2.4: Definizione di varianti polimorfe in OCaml

Come viene mostrato nell'esempio, anteponendo il `‘` al tag, si indica che quel tipo è una variante polimorfa, permettendo al compilatore di creare una struttura espandibile, come `lista_mista`. La sintassi `[> Numero of int ]` indica che quel tipo di variante può contenere **almeno** `Numero` ma anche altri tags. Il compilatore può anche inferire un bound massimo, come mostrato nell'esempio.

```
(* variante polimorfa con un massimo di tags accettati *)
let tipo_di_sport x =
  match x with
  | ‘String s -> s
  | ‘Sport s -> dimmi_lo_sport s
(* val tipo_di_sport : [< ‘String of string | ‘Sport of sport ] -> bool
   = <fun> *)
```

Listing 2.5: Utilizzo di variante polimorfa

Questo perché la funzione `tipo_di_sport` non sa come gestire tipi diversi da quelli all'interno del pattern-matching, per cui inferisce che `x` possa contenere al massimo due tag. In altre parole `<` e `>` sono indicatori che definiscono un upper bound ed un lower bound per il numero e il tipo di tag ammessi.
In generale le varianti polimorfe permettono di creare codice altamente espressivo e flessibile, risultando una feature unica nel suo genere.

Garbage collector

OCaml gestisce la memoria in maniera implicita tramite un meccanismo di garbage-collection generazionale [7]. Il garbage collector è necessario poiché non è possibile (appositamente) una gestione esplicita della memoria, al contrario di linguaggi come C.
Il meccanismo di garbage-collection di OCaml è interessante in quanto si basa sull'ipotesi generazionale, ipotesi empirica che mostra come in molti programmi venga o allocata della memoria e deallocata dopo poco tempo, oppure rimanga allocata fino alla fine dell'esecuzione. Sfruttando questo a proprio vantaggio, il garbage collector di OCaml è stato progettato con due diversi heap, uno chiamato **minor heap** e uno **major heap**. I due heap vengono utilizzati in questo modo: nel minor heap vengono memorizzati tutti quegli oggetti poco durevoli, mentre nel major heap rimangono quelli che durano più tempo. Il funzionamento è il seguente: ad ogni passata nel minor heap, tutto ciò che sopravvive (cioè che non è stato deallocato), viene passato al major heap. Le passate nel minor heap sono molto più frequenti mentre quelle nel major heap sono più rare, in quanto tende a riempirsi meno; per questo motivo il garbage-collector di OCaml risulta particolarmente efficiente e non affetto dalle problematiche dei garbage collector basati su reference-counting.
Durante l'implementazione di binding è necessario tenere in conto di connettere il garbage collector OCaml con la gestione tramite reference counting di GTK [8], in modo tale che un oggetto venga deallocato dal garbage collector solamente quando non è più utilizzato in nessun modo.

2.3 Haskell

Haskell è un linguaggio di programmazione funzionale puro, quindi senza side effects. Nasce negli Stati Uniti all'inizio degli anni '90 prendendo spunto da Miranda, ed è attualmente il linguaggio funzionale puro più utilizzato a livello commerciale ed accademico[9]. Haskell fa largo uso di monadi, un costrutto di programmazione ad alto livello non così presente in OCaml. Durante la conversione effettuata in questa tesi, è stato necessario tradurre il codice monadico in codice OCaml (senza monadi), pertanto nella prossima sezione viene introdotto il concetto di monade.

2.3.1 Monadi

Una monade è un'astrazione di alto livello che viene descritta in matematica come "un monoide nella categoria degli endofuntori", cioè un funtore che mappa una categoria con se stessa. A parte gli aspetti matematici, in informatica e soprattutto in linguaggi come Haskell vengono utilizzate per esprimere computazioni componibili e permettere allo sviluppatore di astrarre la logica del programma evitando codice boilerplate [10].
Una monade è definibile come una tripla $< T, \eta, \mu >$:

- T: rappresenta il tipo della monade
- η: funzione del tipo `a -> T a`
- μ. funzione che permette di comporre monadi dello stesso tipo

In Haskell, la funzione η è chiamata `return` e "inietta" un valore all'interno della monade, mentre μ è chiamata `bind`. In una monade di tipo `m`, la `bind` è tipata come `m a -> (a -> m b) -> m b` e ha lo scopo di combinare due valori monadici. È stata infatti creata una sintassi infissa della `bind` per ricordare un concetto di sequenzialità, ad esempio: `ma >>= \x -> mb` Nei prossimi capitoli verrà illustrato quali monadi sono state incontrate nel codice Haskell e in che modo la loro semantica è stata convertita in OCaml.

2.3.2 haskell-gi

Per quanto riguarda la generazione automatica di binding per Haskell, esiste una libreria chiamata haskell-gi [11], che implementa un modulo di parsing dei file .gir e un modulo di generazione di binding tra il codice C e il codice Haskell. Il lavoro svolto in questa tesi ha preso come riferimento haskell-gi per la conversione del modulo di parsing e di rimbalzo anche per il modulo di generazione di codice. haskell.gi è utilizzabile per generare binding per qualsiasi libreria che implementi un meccanimo di GObject-Introspection, in particolare GTK e tutte le librerie sottostanti.

2.4 ocaml-gi-gtk

Nonostante gli utilizzi industriali, OCaml è considerabile un linguaggio di programmazione emergente, non avendo ancora a disposizione alcune librerie che sono già state implementate per linguaggi più diffusi. Una di queste è una libreria per la generazione automatica di binding GTK tra C e OCaml tramite GObject-Introspection.
I binding tra GTK e OCaml vengono attualmente implementati da lablgtk[12], una libreria scritta a mano che copre circa il 45% delle API di GTK [13]. Nonostante la correttezza della libreria, l'implementazione di binding richiede lavoro manuale ed è soggetta, come descritto in precedenza, a manutenzione nel caso venga modificata GTK.
Dal 2018 esiste ocaml-gi-gtk, un progetto sperimentale partito dalla tesi magistrale di Alberto Nicolini con il prof. Sacerdoti Coen che punta a creare una libreria per la generazione automatica di binding GTK per OCaml. Questa libreria sfrutta il meccanismo di GObject-Introspection e si ispira ad haskell-gi e lablgtk, andando a riutilizzare il modulo di parsing di haskell-gi tout-court e modificando il modulo di generazione di codice di haskell-gi in modo che generi codice OCaml, ereditando tecniche utilizzate all'interno di lablgtk. Il risultato di questo progetto è promettente, in quanto è stato possibile coprire l'80% delle API di GTK in modo automatico[13]. Un limite di ocaml-gi-gtk risiede nel fatto è che è scritta in Haskell e potrebbe non venire addottata dalla community OCaml.
L'obiettivo di questa tesi è la conversione di ocaml-gi-gtk in OCaml, in modo da poter rendere rendere disponibile una libreria potenzialmente utilizzabile ed evidenziare le differenze principali che emergono nella conversione tra i due linguaggi.

2.4.1 Binding GTK per OCaml

Lo studio approfondito del metodo di creazione di binding GTK per OCaml è stato fatto durante lo sviluppo di ocaml-gi-gtk. I concetti principali sono:

- Binding delle primitive
- Conversione dei tipi tra OCaml e C
- Gestione del garbage collector

Per un dettaglio più granulare si rimanda alla tesi di Nicolini, di seguito vengono descritte superficialmente le tre tematiche principali.

Binding delle primitive

Nella creazione dei binding si tengono in conto esclusivamente le funzioni di C e le funzioni di OCaml, dove le prime vengono chiamate primitive.
Esistono due tipologie principali binding per le primitive:

- Binding per primitive aventi arietà $<= 5$
- Binding per primitive aventi arietà > 5

Le prime verranno implementate tramite funzioni C che prendono in input n parametri di tipo `value`, dove il tipo `value` viene utilizzato per rappresentare i tipi di OCaml all'interno delle funzioni C.
Le seconde necessitano di due funzioni C per poter implementare un binding corretto, dove la prima funzione prende in input due parametri (un puntatore e un intero che indica il numero di parametri) e la seconda prende in input tutti e gli n i parametri. Queste funzioni sono necessarie per garantire il corretto processo di conversione dei tipi di dati tra i due linguaggi e possono essere chiamate all'interno di un file OCaml attraverso la direttiva `external`.
Infine, per completare l'implementazione di una primitiva sono necessari due passaggi:

- Fare in modo che il passaggio dei tipi da un linguaggio all'altro sia coerente, il che consiste nel convertire i valori dati in input alla funzione OCaml in valori C da dare

in input alla funzione C sottostante e ritrasfomare i valori C restituiti dalla funzione C in valori corretti per OCaml.

- Computare il risultato della funzione

Un aspetto fondamentale da tenere a mente durante l'implementazione delle primitive è la gestione corretta della memoria: è infatti necessario fare in modo che l'interazione tra la gestione della memoria di GTK tramite reference counting e il garbage collector di OCaml sia coerente, "istruendolo" su quali siano le nuove radici (nel caso di puntatori in strutture dati C a valori OCaml) e su quando rilasciarle.

Conversione dei tipi tra OCaml e C

Il tipo `value` che viene utilizzato nelle primitive C per rappresentare un dato OCaml dev'essere codificato in modo tale da poter contenere le informazioni necessarie ai tipi di alto livello utilizzati in OCaml. Le rappresentazioni sono le seguenti:

- **Tipi di dato atomici**: vengono rappresentati da integer unboxed (int e char) o da blocchi taggati per tutti gli altri tipi di base.

- **Tuple e record**: vengono sfruttati dei puntatori a blocchi con tag $= 0$ e vengono effettuate delle ottimizzazioni nel caso i record siano collassabili ad array o a dei dati unboxed.

- **Array**: gli array di interi vengono rappresentati come tuple, cioè sfruttando dei puntatori a blocchi con tag $= 0$, mentre gli array di float vengono rappresentati in modo unboxed.

- **Varianti**: i costruttori costanti vengono rappresentati da integer unboxed mentre per i costruttori non costanti vengono utilizzati dei blocchi in cui il tag e una numerazione dei costruttori non costanti.

- **Oggetti**: vengono rappresentati da blocchi in cui sono memorizzate le seguenti informazioni: classe e metodi, ID utilizzato per il confronto e i valori delle variabili dell'oggetto.

- **Varianti polimorfe**: queste sono di interesse poiché non viene utilizzata una numerazione che parte da 0 come per le varianti normali, ma viene utilizzato un hash, calcolato all'interno del file `<caml/mlvalues.h>`.

Gestione del garbage collector

Per com'è implementato il GC, ogni volta che viene allocata una variabile OCaml c'è la possibilità che venga effettuata una collection. Per evitare che il garbage collector deallochi variabili OCaml ancora utilizzate nel codice C, sono fornite delle macro per indicare al GC di "fermarsi" e non iniziare una collection.
Quando il garbage collector OCaml effettua una collection (minor o major), è necessario che le funzioni C che operavano sugli elementi deallocati vengano gestite nel modo corretto[8]:

- Ogni funzione che ha parametri o variabili locali di tipo `value` è necessario che inizi con una chiamata a delle macro specifiche e terminare con una chiamata e `CAMLreturn`.

- Ogni assegnamento al campo di un blocco deve essere effettuato tramite delle marco apposite

- Ogni volta che è presente una variabile globale di tipo `value`, questa dev'essere necessariamente registrata globalmente, sempre utilizzando delle macro specifiche.

Capitolo 3

Impostazione del progetto

Questa tesi ha l'obiettivo di convertire ocaml-gi-tk, attualmente scritta in Haskell, nel linguaggio OCaml, in modo da creare la prima libreria di generazione automatica di binding GTK tra C e OCaml scritta a sua volta in OCaml. L'implementazione parte dallo studio della struttura di ocaml-gi-gtk, in modo da comprenderne i punti focali e i possibili problemi per quanto riguarda l'obiettivo di conversione. Verranno inoltre discusse le differenze con haskell-gi, in quanto la libreria risulta un fork di haskell-gi e molto del codice è stato riutilizzato tout-court.

3.1 Struttura di ocaml-gi-gtk

La repository è caricata su GitHub al seguente indirizzo `https://github.com/illbexyz/ocaml-gi-gtk` [14] ed è ancora in fase di sviluppo.
Le directory principali della repository sono le seguenti:

- `app`: contiene il file Main.hs, entry point della repository
- `src`: contiene tutti i file sorgente per la generazione di codice

La conversione della directory `src` è stata eseguita per quasi tutti i file: per alcuni non era necessario o non è stata effettuata per motivi di tempo. Di seguito vengono riportati i dettagli, dividendo i file per sezioni in modo da rendere più chiara la gerarchia

delle chiamate tramite un approccio top-down. Alla fine del capitolo verrà trattata la gestione del parsing dei file.gir.

Altri file

È presente tutta una serie di file per la gestione specifica di ogni struttura (Costanti, Oggetti, Metodi, Interfacce, Segnali, Proprietà, Strutture, Funzioni, Enum) dove viene implementata la scrittura di codice a basso livello e una directory contenente dei file di override. I file di `override` vengono utilizzati nella generazione dei binding in quanto contengono delle speciali annotazioni che permettono di sovrascrivere alcuni valori letti dai file `.gir`, aggiunendo informazioni utili per l'implementazione corretta del binding [15]. Come mostrato di seguito, le annotazioni contengono dei comandi che permettono di cambiare un certo valore del file `.gir`: la prima riga indica che nel namespace `Atk` si vuole modificare il campo nullable del `return-value` della funzione `get_description` appartenente alla classe `Action`, settandolo come `true`. Riassumendo: il file `.gir` può contenere dei valori che possono essere cambiati in base al tipo di linguaggio per cui si vuole generare il binding. In questo esempio si settano a `nullable` dei valori di ritorno di alcune funzioni poiché OCaml li può gestire utilizzando un tipo `option`.

```
1 set-attr Atk/Action/get_description/@return-value nullable 1
2 set-attr Atk/Action/get_keybinding/@return-value nullable 1
3 set-attr Atk/Action/get_localized_name/@return-value nullable 1
4 set-attr Atk/Action/get_name/@return-value nullable 1
5 set-attr Atk/Action/get_description/@return-value nullable 1
6 set-attr Atk/Action/get_keybinding/@return-value nullable 1
7 set-attr Atk/Action/get_localized_name/@return-value nullable 1
```

Listing 3.1: Esempio di comandi di override

3.1.1 Implementazione delle monadi

Prima di entrare nel dettaglio del codice sorgente, viene di seguito illustrata la struttura delle monadi principali che vengono utilizzate all'interno di ocaml-gi-gtk. Le seguenti monadi sono ereditate completamente da haskell-gi, che ne fa lo stesso uso al suo interno.

Monade per la generazione del codice

All'interno del file `Code.hs` viene dichiarata la monade principale per la gestione della generazione di codice, come mostrato.

```
type BaseCodeGen excType a
  = ReaderT CodeGenConfig (StateT (CGState, ModuleInfo) (Except excType)
    ) a
```

Listing 3.2: Monade per la generazione di codice

La monade è una combinazione di:

- Uno stato in sola lettura `CodeGenConfig`, utilizzato per leggere le informazioni sulle API disponibili (quelle già generate), la configurazione dell'ambiente del modulo e una mappa utilizzata per la gestione della documentazione del codice generato.
- Uno stato in lettura e scrittura `StateT`, composto dalla tupla `(CGState, ModuleInfo)`. Il primo elemento della tupla viene utilizzato principalmente per la generazione di variabili fresche, mentre il secondo elemento è il core dove viene salvato tutto il codice che dev'essere generato.
- La possibilità di generare e gestire errori, utilizzato per la gestione delle eccezioni non bloccanti (impossibilità a generare un metodo, conflitti di tipi, etc).

Monade per la scrittura del codice

Per quanto riguarda la scrittura del codice su file, viene dichiarata la monade `GenOutput` che assolve al compito di avere uno stato globale dove tenere memorizzati i file C necessari alla generazione corretta delle primitive.

```
type GenOutput = StateT CFiles IO
```

Listing 3.3: Monade per la scrittura su file

Come mostrato sopra, viene dichiarato uno state `StateT` (come nella monade per le generazione del codice) e viene anche data la possibilità di eseguire operazioni di input output tramite il tipo `IO`. Questo perché il fine ultimo della monade è scrivere su file e Haskell forza ad utilizzare la monade `IO` per quanto riguarda la gestione e l'utilizzo di side-effects di questo tipo [16].

Paradigma di utilizzo delle monadi

All'interno di ocaml-gi-gtk e in generale all'interno del codice Haskell, le monadi compongono computazioni, rendendo il codice lazy per quanto riguarda il sollevamento di eccezioni o altri side-effects. Sono presenti quindi delle funzioni di libreria che "valutano" la monade, come ad esempio la `runExcept`: questa prende in input una monade, gli stati iniziali e esegue tutte le computazioni che sono state composte all'interno della monade. Ad esempio: se il codice sollevasse un'eccezione in un certo punto, l'eccezione non verrebbe sollevata immediatamente, ma solo al momento della `runExcept`. Questo è stato pensato per permettere alla libreria di scrivere ad esempio submoduli utilizzando stati di partenza vuoti e per meglio gestire le eccezioni. Nel prossimo capitolo verrà illustrato com'è stato ricreato questo comportamento in OCaml.

3.1.2 Routine di alto livello

In questa sezione vengono inseriti tutti i file di alto livello che contengono le routine principali per la generazione di codice.

Main.hs

All'interno del `Main.hs` vengono dichiarate manualmente tutte le librerie GTK che sono state testate, il quale elenco si trova nel capitolo precedente. Oltre alla gestione degli input da riga di comando per l'avvio dello script, viene chiamata la funzione `genBindings` appartenente al modulo `GI`, la quale prende in input un tipo di dato `library` e restituisce `unit`, scrivendo il codice generato su file. Questo file è completamente creato per ocaml-gi-gtk, non appartiene ad haskell-gi.

GI.hs

All'interno di `GI.hs` viene dichiarato il tipo di dato `library` come record contenente nome della libreria, versione e file di override opzionale. La funzione `genBindings` esegue alcune subroutine per la gestione delle directory e del file-system per la generazione del codice, carica l'eventuale file di override e chiama la `genLibraryCode`, la quale prende in input una libreria e degli override e restituisce un modulo completamente parsato e

una lista di dipendenze per quel modulo. La `genLibraryCode` è importante perché connette i due macro-moduli, quello di parsing e quello di generazione di codice. Infatti al suo interno viene chiamata la `loadGIRInfo` (definita in `API.hs` che restituisce le informazioni riguardo ai file `.gir` parsate e processate e le dà in pasto alla `genCode` (definita in `Code.hs`). Le funzioni `loadGIRInfo`, `genCode` e `genModule` sono gli entry point principali che chiamano la maggior parte delle subroutine di basso livello. Questo file non appartiene ad haskell-gi.

CodeGen.hs

All'interno di questo file viene definita la `genModule`, che prende in input le api parsate e modifica lo stato della monande `BaseCodeGen` con tutte le informazioni generate per il modulo corrente. La funzione `genModule` effettua una serie di pulizie sui dati e chiama la funzione `submodule` passandole la funzione `genAPI`: la prima è definita in `Code.hs` ed una routine per la gestione della generazione dei submoduli, la seconda è l'entry point per la generazione delle API singole. Qui viene fatto pattern matching sul tipo dell'API (restituito dal parser come mostrato in seguito) e chiamata la routine corrispondente. Come esposto nella tesi di Nicolini, attualmente ocaml-gi-gtk non ha ancora implementato tutte quelle API in cui compare `return ()`.

```haskell
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
```

Listing 3.4: Funzione `genAPI` Haskell

La generazione delle `Struct` e delle `Union` viene fatta internamente al file `CodeGen.hs` in quanto più semplice, mentre quelle più complesse sono la generazione di `Enum`, `Flags`, `Object` e `Interface`. In haskell-gi la parte di generazione API viene fatta interamente

in questo file (a parte per gli Enum) mentre in ocaml-gi-gtk sono stati creati dei file ad hoc per ogni tipo di API.

API.hs

Questo file connette la parte di generazione di codice con quella di parsing dei file `.gir`. Al suo interno vengono dichiarate diverse routine per il parsing a partire da un documento fino ad arrivare alla singola API contenuta all'interno di esso. Viene dichiarato un tipo API come variante che "impacchetta" i vari tipi dichiarati durante la fase di parsing per ogni tipo di struttura.

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

Listing 3.5: Tipo di dato `API`

La funzione che viene esposta è `loadGIRinfo` che prende in input il nome del modulo da parsare e restituisce un formato `GIRInfo` in cui vengono memorizzate tutte le informazioni processate del modulo e la lista delle relative dipendenze.

```
data GIRInfo = GIRInfo {
      girPCPackages      :: [Text],
      girNSName          :: Text,
      girNSVersion       :: Text,
      girAPIs            :: [(Name, API)],
      girCTypes          :: M.Map Text Name
    }
```

Listing 3.6: Tipo di dato `GIRInfo`

La terza funzione interessante e che mostra come il file `API.hs` si vada a collegare con il modulo di parsing è la `parseNSElement`, la quale esegue un pattern-matching sul

nome dell'elemento parsato dai file `.gir` e restituisce l'API e il parser corrispondente che si occupano della creazione delle strutture a partire dai dati grezzi.

```haskell
parseNSElement :: M.Map Alias Type -> GIRNamespace -> Element ->
    GIRNamespace
parseNSElement aliases ns@GIRNamespace{..} element
    | lookupAttr "introspectable" element == Just "0" = ns
    | otherwise =
        case nameLocalName (elementName element) of
          "alias" -> ns      -- Processed separately
          "constant" -> parse APIConst parseConstant
          "enumeration" -> parse APIEnum parseEnum
          "bitfield" -> parse APIFlags parseFlags
          "function" -> parse APIFunction parseFunction
          "callback" -> parse APICallback parseCallback
          "record" -> parse APIStruct parseStruct
          "union" -> parse APIUnion parseUnion
          "class" -> parse APIObject parseObject
          "interface" -> parse APIInterface parseInterface
          "boxed" -> ns -- Unsupported
          "docsection" -> ns -- Ignored for now, see https://github.com/
    haskell-gi/haskell-gi/issues/318
          n -> error . T.unpack $ "Unknown GIR element \""
```

Listing 3.7: Funzione `parseNSElement` Haskell

La funzione che va a leggere effettivamente i file `.gir` è la `readGirepository` che fa parte del modulo di parsing e viene chiamata all'interno di questo file per poter dare in input il file crudo alle routine di parsing XML. Questo file viene riutilizzato da ocaml-gi-gtk praticamente tout-court.

3.1.3 Routine di basso livello

In questa sezione vengono descritti i file e le funzioni che implementano la generazione di codice di più basso livello, andando a scrivere effettivamente il codice generato all'interno dello stato della monade.

Code.hs

Questo file contiene le dichiarazioni delle monadi descritte in precedenza e tutte le strutture dati utilizzate per la generazione di codice. Viene definito un tipo di dato `Code` come una sequenza di `CodeToken`, un tipo variante. Questo tipo di dato verrà utilizzato per immagazzinare effettivamente le stringhe di codice da scrivere su file.

```haskell
-- | Dichiarazione del tipo di dato Code
newtype Code = Code (Seq.Seq CodeToken)

-- | Dichiarazione del tipo di dato CodeToken
data CodeToken
    = Line Text
    | Indent Code
    | Group Code
    | IncreaseIndent
```

Listing 3.8: Tipo di dato `Code` e `CodeToken`

Come mostrato, i due tipi di dato sono mutualmente ricorsivi, fatto che necessiterà di un adeguamento per la conversione in OCaml. Le `Seq` in Haskell sono delle strutture dati simili alle liste che permettono di avere perfomance migliori in operazioni come l'accesso in coda, la concatenazione e la lookup di un elemento [17]. Il tipo `CodeToken` può memorizzare delle linee codice, delle regioni indentate di codice, degli insiemi raggruppati di linee e la presenza di un'indentazione.

La seconda struttura dati interessante è quella che viene utilizzata all'interno dello stato della monade `BaseCodeGen`, cioè `ModuleInfo`.

```haskell
data ModuleInfo = ModuleInfo {
      modulePath :: ModulePath -- ^ Full module name: ["Gtk", "Label"]
    , moduleCode :: Code       -- ^ Generated code for the module
    , bootCode   :: Code
    , gCode      :: Code       -- ^ OOP OCaml code
    , cCode      :: Code       -- ^ .c stubs' code
    , hCode      :: Code       -- ^ .h stubs' code
    , tCode      :: Code       -- ^ code for the type declaration module
    , submodules :: M.Map Text ModuleInfo
    , moduleDeps :: Deps -- ^ Set of dependencies for this module
```

```haskell
    , cDeps       :: Deps -- ^ Set of C dependencies for this module
    , moduleExports :: Seq.Seq Export -- ^ Exports for the module.
    , qualifiedImports :: Set.Set ModulePath -- ^ Qualified imports.
    , sectionDocs    :: M.Map HaddockSection Text -- ^ Documentation
    }
```

Listing 3.9: Tipo di dato `ModuleInfo`

Il tipo di dato `ModuleInfo` è un record che contiene tutte le informazioni necessarie per la generazione del codice e i campi vengono modificati in modo da aggiungere linee di codice ogni volta che viene generata una struttura. Nel dettaglio i campi del record interessanti sono:

- **modulePath**: lista di stringhe per memorizzare il percorso del modulo
- **moduleCode**: codice OCaml generato di basso livello
- **gCode**: codice OCaml generato di alto livello
- **tCode**: codice OCaml generato per le dichiarazioni di tipo
- **cCode**: codice C generato nei file .c
- **hCode**: codice C generato nei file .h
- **submodules**: la mappa di tutti i sotto moduli di un modulo, indicizzati dal nome del sottomodulo
- **moduleDeps**: l'insieme di dipendenze OCaml di un modulo
- **cDeps**: l'insieme di dipendenze C di un modulo

Tutti questi campi vengono modificati dalle rispettive funzioni mostrate, che alterano lo stato della monade andando ad aggiungere o modificare informazioni.

```haskell
tellCode :: CodeToken -> CodeGen ()
tellCode c = modify'
  (\(cgs, s) -> (cgs, s { moduleCode = moduleCode s <> codeSingleton c
    }))

```

```haskell
tellGCode :: CodeToken -> CodeGen ()
tellGCode c =
  modify' (\(cgs, s) -> (cgs, s { gCode = gCode s <> codeSingleton c }))

tellHCode :: CodeToken -> CodeGen ()
tellHCode c =
  modify' (\(cgs, s) -> (cgs, s { hCode = hCode s <> codeSingleton c }))

tellTCode :: CodeToken -> CodeGen ()
tellTCode c =
  modify' (\(cgs, s) -> (cgs, s { tCode = tCode s <> codeSingleton c }))
```

Listing 3.10: Funzioni per modificare lo stato `ModuleInfo`

La funzione `modify'` è una funzione monadica che modifica lo stato della monade nella maniera specificata [18].
Il primo elemento della tupla che forma lo stato della monade è un tipo di dato record che trasporta informazioni che riguarda la generazione di nomi per variabili di tipo fresche e che viene acceduto e modificato in maniera sensibilmente meno frequente rispetto al `ModuleInfo`. Ad esempio nelle funzioni mostrate sopra, questo dato è rappresentato da `cgs` e viene restituito tout-court.

```haskell
data CGState = CGState {
  cgsNextAvailableTyvar :: NamedTyvar -- ^ Prossima variabile
   disponibile
  }

data NamedTyvar = SingleCharTyvar Char -- ^ variabile senza indice
                | IndexedTyvar Text Integer -- ^ variabile con indice
```

Listing 3.11: Tipo di dato `CGState`

L'unica funzione che permette di modificare questo stato è la `getFreshTypeVariable`, che semplicemente legge lo stato, crea una stringa fresca in modo sequenziale e restituisce lo stato allo step successivo, in modo tale che ad ogni chiamata venga restituita una nuova stringa non ancora utilizzata e non in scope.

```haskell
getFreshTypeVariable :: CodeGen e Text
getFreshTypeVariable = do
```

```
3   (cgs@(CGState{cgsNextAvailableTyvar = available}), s) <- get
4   let (tyvar, next) =
5         case available of
6           SingleCharTyvar char -> case char of
7             'z' -> ("z", IndexedTyvar "a" 0)
8             'm' -> ("n", SingleCharTyvar 'o')
9             c -> (T.singleton c, SingleCharTyvar (toEnum $ fromEnum c +
    1))
10          IndexedTyvar root index -> (root <> tshow index,
11                                      IndexedTyvar root (index+1))
12  put (cgs {cgsNextAvailableTyvar = next}, s)
13  return tyvar
```

Listing 3.12: Creazione variabile fresca

Riassumendo, partendo dalla stringa `a`, una volta esaurito l'alfabeto, si riparte da capo aggiungendo un numero che cresce in maniera sequenziale. In questo modo non si potranno mai avere collisioni di nomi.

Il file `Code.hs` viene quindi importato in tutti i file specifici per la generazione delle strutture dati in quanto espone la monade e permette di modificarne lo stato.
All'interno del file troviamo anche la routine per la scrittura su file, la `writeModuleInfo`, che prende in input la directory in cui scrivere, il `ModuleInfo` completo e restituisce la monade per la scrittura del codice mostrata in precedenza, la `GenOutput`. Questa viene utilizzata dalla `WriteModuleTree` per generare ogni singolo modulo. Si ricorda infatti che ogni modulo contiene più submoduli, creando una struttura arborescente.

Conversions.hs e TypeRep.hs

Questi due file sono stati completamente ri-arrangiati in quanto esprimono concetti radicalmente connessi al linguaggio che dev'essere generato in output. In particolare vengono definiti dei tipi di dato e delle routine utili a convertire effettivamente i tipi di dato in C a quelli in OCaml e viceversa e per esprimere con la stringa corrispondente i tipi di dati espressi tramite il codice della libreria. Di seguito viene mostrato un estratto di codice.

```
1 -- ^ converte da tipo interno a rappresentazione testuale in OCaml da
    scrivere su file
```

```
ocamlBasicType TFloat     = TextCon "float"

-- ^ converte da tipo interno a rappresentazione testuale in C
cType TFloat     -> return "gfloat"

-- ^ converte da tipo interno a rappresentazione testuale in OCaml
ocamlDataConv TFloat     -> return "float"

-- ^ converte da tipo interno a rappresentazione testuale dei valori di
    input delle primitive
ocamlValueToC TFloat     -> return "Float_val"

-- ^ converte da tipo interno a rappresentazione testuale dei valori in
    output delle primitive
cToOCamlValue TFloat     -> return "caml_copy_double"
```

Listing 3.13: Conversioni tipi di bassi livello

Il file `TypeRep.hs` definisce un tipo di dato variante per rappresentare strutture dati e pattern di OCaml.

```
data TypeRep = ListCon TypeRep                  -- liste
             | OptionCon TypeRep                -- valori option
             | ObjCon TypeRep                   -- Gobject.obj
             | RowCon RowDirection TypeRep      -- [> ] and [< ]
             | TypeVarCon Text TypeRep          -- aggiunge variabili
             | TupleCon [TypeRep]               -- tuple
             | PolyCon TypeRep                  -- varianti polimorfe
             | TextCon Text                     -- tipi atomici
             | NameCon Name                     -- oggetti/interfacce
```

3.2 Tentativo di parsing

Come si può notare dalla sezione precedente, non viene menzionato il modulo di parsing all'interno di ocaml-gi-gtk. Questo perché è stato completamente riutilizzato quello di haskell-gi senza applicarne nessuna modifica, scelta sensata in quanto non risulta connesso in nessun modo alla generazione di codice soprastante.

Durante lo sviluppo di questo elaborato è stato quindi necessario tradurre anche il modulo di parsing direttamente da haskell-gi, fattore che ha inciso sui tempi di sviluppo. Il primo tentativo effettuato è stato quello di sfruttare la repository `OCaml-GObject-Introspection` [19] che espone delle API per raccogliere informazioni sui file `.typelib`, un formato binario quasi equivalente ai file `.gir`. I file `.typelib` vengono anch'essi generati dal meccanismo di GObject-Introspection e possono essere letti per recuperare le medesime informazioni.
L'obiettivo era quello di utilizzare `OCaml-GObject-Introspection` con lo scopo di evitare il parsing xml diretto e velocizzare lo sviluppo, adattando i valori restituiti con le strutture dati di parsing utilizzate dal modulo di haskell-gi. La repository espone un entry-point per il caricamento di una libreria (in questo caso GTK) e una serie di API per raccogliere dati sulla libreria caricata.
Nelle due figure seguenti viene mostrata la differenza tra l'implementazione del tipo di dato `Constant` in `haskell-gi` e il primo tentativo in questa tesi, in modo da mostrare differenze e criticità di `OCaml-GObject-Introspection`.

```haskell
data Constant = Constant {
      constantType          :: Type,
      constantValue         :: Text,
      constantCType         :: Text,
      constantDocumentation :: Documentation,
      constantDeprecated    :: Maybe DeprecationInfo
    } deriving (Show)

parseConstant :: Parser (Name, Constant)
parseConstant = do
  name <- parseName
  deprecated <- parseDeprecation
  value <- getAttr "value"
  t <- parseType
  ctype <- parseCType
  doc <- parseDocumentation
  return (name, Constant { constantType = t
                         , constantValue = value
                         , constantCType = ctype
```

```
                              , constantDocumentation = doc
                              , constantDeprecated = deprecated
                              })
```

Listing 3.14: Parsing di costanti in Haskell

La libreria haskell-gi implementa tutte le strutture dove inserire i dati parsati come record ed utilizza la monade `parser` per la gestione del parsing, la quale verrà spiegata nel prossimo capitolo. Come si può notare dalla figura successiva, è stato creato un record quasi identico per rimanere il più aderenti possibile all'implementazione originale. A parte la differenza tra le sintassi dei due linguaggi, salta all'occhio la presenza di due campi commentati: il grosso limite riscontrato infatti in `OCaml-GObject-Introspection` è quello di non avere la quantità di API necessarie per coprire tutte le informazioni esposte dai file `.gir`, come per esempio il campo `constantCType` o il campo `constantDocumentation`. Per altre informazioni mette a disposizione solo dati parziali, come per `constantIsDeprecated`, il quale è stato dichiarato `bool` perché il tipo di ritorno non dava altre informazioni. È stato anche notato come `OCaml-GObject-Introspection` restituisca a volte tipi già processati, come `option`, astraendo la presenza o meno di alcune informazioni, fatto che dal parsing XML andrebbe controllato a mano.

```
module GI = GObject_introspection

type constant = {
    constantType: type_ml option;
    constantValue: GI.Types.argument_t Ctypes.union Ctypes.ptr;
   (*constantCType: string; (* campo non disponibile nell'API *)
    constantDocumentation: string;*)(* campo non disponibile nell'API *)
    constantIsDeprecated: bool; (* differisce da haskell-gi, non indica
   il testo *)
    }

let parseConstant constant_info =
    let name = GI.Constant_info.to_baseinfo constant_info |> getName in
    (name,
    { constantValue = GI.Constant_info.get_value constant_info;
      constantType = GI.Constant_info.get_type constant_info |>
   cast_to_type_ml;
```

```
16        constantIsDeprecated = GI.Constant_info.to_baseinfo constant_info
      |> GI.Base_info.is_deprecated;
17     })
```

Listing 3.15: Parsing di costanti in OCaml

Si è proceduto implementando tutti i file di parsing e il file `API.hs` del modulo di generazione di codice e al momento del testing sulle librerie GTK ci si è accorti che veniva sollevato un `coredump`. Un errore di questo tipo non può essere sollevato da OCaml, in quanto il sistema di tipi e il compilatore non ammettono errori a runtime. Il problema nasceva quindi da un'eventuale implementazione scorretta di `OCaml-GObject-Introspection`, la quale si interfaccia con codice C per la lettura dei file `.typelib`. È stato riscontrato l'errore nella gestione del garbage collector, che per un qualche motivo non risultava coerente con le chiamate effettuate e generava accessi a porzioni di memoria deallocate.
Si è provato a contattare il creatore di `OCaml-GObject-Introspection` e a creare l'issue relativa su GitHub ma non si è ricevuta risposta. Per questo motivo si è deciso di creare un modulo di parsing ad-hoc a partire dalla lettura dei file XML, in modo da risolvere anche il problema dei dati mancanti.
È attualmente comunque aperto il branch `With_C_API` di questo elaborato che utilizza `OCaml-GObejct-Instrospection` come metodo di parsing.

Capitolo 4

Implementazione

Dopo aver analizzato la struttura di ocaml-gi-gtk e di haskell-gi si è passati alla fase di conversione vera e propria. La repository è pubblica ed è disponibile all'indirizzo `https://github.com/AlbertoDrusiani/ocaml-gi`

4.1 Scelte implementative

Sono state fatte alcune scelte di implementazione per ottimizzare il processo di conversione tra i due linguaggi, alcune delle quali sono rivedibili a posteriori, discusse nel prossimo capitolo.
Le scelte principali sono:

- Scelta della libreria di parsing XML
- Sintassi e costrutti OCaml
- Conversione delle monadi

4.1.1 Libreria di parsing XML

Essendo necessario un parsing di basso livello dopo il tentativo fallimentare avvenuto utilizzando `OCaml-GObject-Introspection`, è stata scelta una specifica libreria di parsing XML. Per OCaml esistono diverse librerie (sempre meno rispetto a linguaggi più in voga)

di parsing XML, con caratteristiche diverse e utilizzi differenti. Per cercare di mantenere l'aderenza con haskell-gi si è consultata la libreria di parsing utilizzata all'interno della repository, `Text.XML` [1].
Essa definisce degli elementi `Node` ed `Element` come mostrato di seguito.

```haskell
data Node = NodeElement :: Element
          | NodeInstruction ::Instruction
          | NodeContent :: Text
          | NodeComment :: Text

data Element = Element {
    elementName :: Name
    elementAttributes :: Map Name Text
    elementNodes :: [Node]
}

data Name = Name {
    nameLocalName :: Text
    nameNamespace :: Maybe Text
    namePrefix :: Maybe Text
}
```

Listing 4.1: Elementi XML in Haskell [1]

Il tipo `Node` rappresenta qualsiasi entità all'interno del file XML (elementi, testo all'interno degli elementi, attributi, commenti), mentre `Element` rappresenta solamente un elemento XML taggato. Per il lavoro fatto in questa tesi l'aspetto più importante è quello espresso da `Element`: ogni elemento ha un `Name`, una mappa di attributi e una lista di figli. Il tipo di dato di `Name` è estremamente importante perché incapsula il nome locale dell'elemento e il namespace di quell'elemento all'interno del file XML, necessario in quanto possono essere presenti elementi con stesso nome ma namespace diverso.
Tra le librerie di parsing XML in OCaml sono state valutate:

- **xml-light**: espone API di basso livello per il parsing di documenti XML e DTD
- **xmlm**: implementa uno streaming codec per la codifica e la decodifica di file XML
- **Markup.ml**: implementa un parser per HTML5 e uno per xml

La scelta è ricaduta su `xml-light` [20] in quanto risulta quella più facilmente adattabile alle specifiche di haskell-gi. Essendo una libreria del 2003, `xml-light` è ben nota nel panorama OCaml e non soffre di particolari problemi. Il tipo di dato di base che implementa risulta estremamente minimale, come mostrato di seguito.

```
type xml =
| Element of (string * (string * string) list * xml list)
| PCData of string
```

Listing 4.2: Tipo di dato XML in xml-light

Ogni nodo `xml` può quindi essere o un `Element` o un `PCData`, cioè il testo all'interno di un nodo. Ogni `Element` è una tupla composta da tre elementi come si può notare nella dichiarazione del tipo. In ordine codificano:

- **Nome**: indica il nome locale (quindi non qualificato) dell'elemento, eventualmente con un prefisso di namespace (esempio: `c:value`).
- **Attributi**: lista di coppie di stringhe, rappresenta quello che nel codice Haskell è la mappa degli attributi, dove la prima stringa della coppia è il nome dell'attributo e la seconda stringa indica il valore.
- **Elementi figli**: tipata come una lista di `xml`, rappresenta tutti i nodi figli dell'elemento corrente.

Si può quindi notare un'analogia con la libreria `Text.XML` di Haskell, con la differenza che al posto di una mappa di attributi è presente una lista di coppie (nome, valore) e al posto di un record di tipo `Name` è presente un campo stringa non qualificato. Nella prossima sezione verrà discusso come è stato aggirato quest'ultimo aspetto.

4.1.2 Sintassi e costrutti OCaml

È stato scelto di tenere i nomi delle funzioni e delle variabili Haskell identici all'interno del codice OCaml. Solitamente i due linguaggi seguono stili diversi, dove Haskell predilige un approccio camelCase mentre OCaml uno snake_case [21]. La scelta è stata effettuata per semplificare la conversione, in modo da trovare più velocemente possibile le funzioni e le variabili ed avere una sorta di mapping 1:1 tra i due linguaggi. Anche in un'ottica in

cui ci si continui ad ispirare ad ocaml-gi-gtk per completare i binding, può essere comodo avere il codice con i nomi identici. Quando la libreria sarà completa e pronta per essere rilasciata, si potrà pensare di tradurre tutti i nomi in snake_case in modo da essere il più aderenti possibili alle best practice sintattiche di OCaml.

```
(* underscore per separare le parole *)
let snake_case = 3;;

(* utilizzo di maiuscole per separare le parole *)
let lowerCamelCase = 3;;

(* utilizzo di maiuscole dalla prima parola *)
let UpperCamelCase = 3;;
```

Listing 4.3: Differenza tra stili di naming

L'unica eccezione che è stata fatta riguarda i nomi dei tipi. In Haskell i nomi dei tipi sono tutti UpperCamelCase, inziando quindi con una lettera maiuscola. La sintassi OCaml forza ad utilizzare una lettera minuscola per dichiarare un tipo in quanto i nomi che iniziano con le maiuscole sono riservati per i costruttori e per i moduli. I tipi di dato sono stati quindi dichiarati in snake_case.
Si è cercato di seguire il più possibile l'approccio funzionale, senza utilizzo di cicli o riferimenti a variabili. Nonostante OCaml permetta questo tipo di approccio, è risultato interessante utilizzare solamente costrutti funzionali come `map` e `fold`.

È stato scelto di non utilizzare mai la direttiva `open` per quanto riguarda le librerie standard di OCaml per evitare di inquinare il namespace e rischiare collisioni di nomi. La struttura del progetto è stata ereditata da ocaml-gi-gtk, aggiungendo all'interno della directory `src` due directory separate, `CodeGen` e `GIR`, rispettivamente contenenti i file per la generazione di codice e i file per il parsing. Il nome di queste directory è stato ereditato da haskell-gi.

4.1.3 Conversione delle monadi

Come già anticipato, OCaml non fa largo utilizzo di codice monadico, nonostante sia possibile una sua implementazione. Le monadi utilizate nel codice Haskell da convertire sono essenzialmente tre:

- **State**: trasporta informazioni su uno stato in scrittura e lettura
- **Reader**: trasporta informazioni su uno stato solo in lettura
- **Error**: trasporta informazioni riguardo errori sollevati

La monade `Error` è stata convertita sfruttando la gestione esplicita delle eccezioni, mentre la `State` e la `Reader` sono state convertite passando e restituendo da funzione a funzione lo stato eventualmente modificato.
Considerando che lo stato utilizzato nelle monadi consisteva spesso di una tupla, si è deciso di utilizzare una tupla anche nella conversione in OCaml, passandola e restituendola come parametro. Un'altra scelta possibile sarebbe stata quella di utilizzare un record con all'interno i campi necessari, ma si rischiava di avere codice più verboso per via dell'accesso ai campi, anche se avrebbe semplificato il debugging del tipaggio di alcune funzioni.

4.2 Parsing

Il modulo di parsing è stato completamento riscritto ex-novo, sfruttando `xml-light` e convertendo il modulo di parsing all'interno di haskell-gi. All'interno del file `XMLUtils.ml`, che in haskell-gi implementa funzioni di supporto al parsing, sono state scritte tutte le funzioni necessarie per rendere aderente `xml-light` a `Text.XML`. Il "problema" del nome non qualificato descritto nel capitolo precedente è stato bypassato implementando una serie di routine ad-hoc per estrarlo dall'elemento XML.
I file `.gir` inseriscono infatti un eventuale prefisso all'interno del nome e/o attributo dell'elemento, che va a sporcare il nome locale. Inoltre, come detto in precedenza, `xml-light` non estrae il namespace dagli elementi, quindi per qualificare un nome è stato necessario indicare manualmente il namespace. Il seguente estratto dal file `GLib.gir` mostra come vengono utilizzati i prefissi.

```xml
<package name="glib-2.0"/>
  <c:include name="glib.h"/> <!-- prefisso nel nome -->
  <namespace name="GLib"
             version="2.0"
             shared-library="libgobject-2.0.so.0,libglib-2.0.so.0"
             c:identifier-prefixes="G" <!-- prefisso nel nome di un attributo-->
             c:symbol-prefixes="g,glib">
```

Listing 4.4: Estratto da GLib.gir

Nell'immagine seguente si può notare come questo frammento XML venga parsato da `xml-light`, stampato nel top level di OCaml.

```ocaml
[Xml.Element ("package", [("name", "glib-2.0")], []);
     Xml.Element ("c:include", [("name", "glib.h")], []);
     Xml.Element
      ("namespace",
       [("name", "GLib"); ("version", "2.0");
        ("shared-library", "libgobject-2.0.so.0,libglib-2.0.so.0");
        ("c:identifier-prefixes", "G"); ("c:symbol-prefixes", "g,glib")
   ],
        .....
```

Listing 4.5: Rappresentazione in xml-light del frammento .gir

Il prefisso viene quindi inserito come suffisso del nome, per cui è necessario scorporarli, mentre il namespace non viene proprio considerato, per cui, come detto sopra, dev'essere aggiunto manualmente. I namespace dei file `.gir` sono tre e vengono definiti all'inizio, come richiesto dalla sintassi standard XML. In namespace sono:

- **Core**: `http://www.gtk.org/introspection/core/1.0`, settato come namespace di default
- **C**: `http://www.gtk.org/introspection/c/1.0`
- **GLib**: `http://www.gtk.org/introspection/glib/1.0`

Innanzitutto è stato definito un record `name` aderente a quello decritto in haskell-gi.

```
(* dichiarazione tipo name aderente a quello Haskell *)
type name = {
    nameLocalName: string;
    nameNamespace: string option;
    namePrefix: string option;
}
```

Listing 4.6: Tipo `name` in OCaml

Per la gestione dell'inserimento del namespace all'interno del record `name` e della rimozione del prefisso per estrarre il nome locale, sono state definite le utilities seguenti.

```
(*estrae il nome locale dalla stringa nome di un elemento/attributo*)
(* string -> string *)
let localName str =
    let l = String.split_on_char ':' str in
    List.hd (List.rev l)

(*estrae il nome qualificato a partire da un elemento*)
(* xml -> name *)
let element_to_name el =
    {nameLocalName = localName (Xml.tag el);
     nameNamespace = Some (get_prefix (Xml.tag el) |>
   prefixToGIRNamespace);
     namePrefix = get_prefix (Xml.tag el);
    }

(*estrae il prefisso da un nome di un elemento o attributo*)
(* string -> string option *)
let get_prefix str =
    let l = String.split_on_char ':' str in
    match l with
    | [xs; _] -> Some xs
    | _ -> None

(* string -> GIRXMLNamespace *)
let prefixToGIRXMLNamespace p =
    match p with
    | Some "c" -> CGIRNS
```

```
    | Some "glib" -> GLibGIRNS
    | _ -> CoreGIRNS

(* GIRXMLNamespace -> string *)
let girXMLNamespaceToPrefix ns =
    match ns with
    | CGIRNS -> "c"
    | GLibGIRNS -> "glib"
    | CoreGIRNS -> ""

(* string option -> string *)
let prefixToGIRNamespace p =
    prefixToGIRXMLNamespace p |> girNamespace
```

Listing 4.7: Gestione dei nomi qualificati

Utilizzando queste funzioni è possibile estrarre in qualsiasi momento un nome qualificato da un elemento XML, costruendo una struttura dati identica a quella utilizzata da haskell-gi.
In haskell-gi viene implementata la monade `Parser` per l'esecuzione del parsing, introducendo uno stato in sola lettura che funge da contesto.

```
data ParseContext = ParseContext {
      ctxNamespace     :: Text,

      treePosition     :: [Text],
      currentElement   :: Element,
      knownAliases     :: M.Map Alias Type
    } deriving Show

type ParseError = Text

type Parser a = ReaderT ParseContext (Except ParseError) a
```

Listing 4.8: Monade Haskell per il parsing

All'interno dello stato è presente il namespace, la posizione nell'albero di parsing, l'elemento corrente e una mappa degli alias conosciuti. Come mostra la dichiarazione della monade, non sono ammessi errori di parsing, ogni errore possibile solleva un'eccezione che

ferma l'esecuzione.
In OCaml la semantica della monade è stata convertita passando namespace, elemento corrente e mappa di alias a tutte le funzione necessarie, e restituendoli in output quando necessario. In questo modo, ogni funzione che necessita di quelle informazione le ha disponibili passate come parametri. La posizione nell'albero di parsing non è stata presa in considerazione in quanto viene utilizzata in haskell-gi solamente per motivi di debugging.
Di seguito viene mostrata la differenza tra il codice monadico di Haskell e il codice senza monadi di OCaml.

```
getAttr :: XML.Name -> Parser Text
getAttr attr = do
  ctx <- ask
  case lookupAttr attr (currentElement ctx) of
    Just val -> return val
    Nothing -> parseError $ "Expected attribute \"" <>
                (T.pack . show) attr <> "\" not present."
```

Listing 4.9: Codice monadico nel parsing XML

```
(* string -> xml -> string *)
let getAttr attr element =
    match lookupAttr attr element with
    | Some v -> v
    | None -> assert false
```

Listing 4.10: Codice OCaml equivalente nel parsing XML

È evidente come il codice Haskell prenda in input solo il nome dell'attributo, richieda alla monade il contesto attuale attraverso il comando `ctx <- ask` e recuperi il valore dell'elemento corrente tramite l'accesso al record `currentElement ctx`, restituendo la monade `Parser Text`.
Il codice OCaml necessita il passaggio esplicito dell'elemento corrente alla funzione e utilizza direttamente quel valore senza doverlo richiedere al contesto. Questo implica che la prima funzione invocata dovrà passare il contesto e verrà passato a macchia d'olio a tutte le funzioni che lo necessitano, ma questo non inficia le performance.
All'interno del file `BasicTypes.ml` sono stati definiti i tipi di base nello stesso modo di haskell-gi, con l'aggiunta di moduli custom per quanto riguarda mappe e insiemi. Il

linguaggio OCaml non implementa infatti `Map` e `Set` come tipo out-of-the-box, ma fornisce dei funtori per implementarle in maniera personalizzata da parte dell'utente. È necessario prima di tutto definire un modulo che implementa l'operazione `compare` sul tipo `t` necessario, e poi utilizzare i funtori `Set.Make` o `Map.Make`, che prendono in input il modulo appena dichiarato e danno in output un modulo che implementa mappe e insieme su quel tipo. La funzione `compare` è necessaria fornirla poiché il linguaggio rende libero l'implementatore di costruirla in maniera personalizzata, fornendonla di default solo per i tipi primitivi.

```
module Name = struct
  type t = name
  let compare {namespace=ns1; name=nm1} {namespace=ns2; name=nm2} =
    match Stdlib.compare ns1 ns2 with
    | 0 -> Stdlib.compare nm1 nm2
    | c -> c
end

module StringString = struct
  type t = string*string
  let compare (str1, str2) (str3, str4) =
    match Stdlib.compare str1 str3 with
    | 0 -> Stdlib.compare str2 str4
    | c -> c
end

module NameSet = Set.Make(Name)
module NameMap = Map.Make(Name)

module StringStringSet = Set.Make(StringString)
module StringStringMap = Map.Make(StringString)
```

Listing 4.11: Dichiarazione dei moduli/tipi per mappe e insiemi

L'utilizzo della `compare` è quello di indicare al compilatore come ordinare le chiavi/elementi per quanto riguarda tipi complessi. Ad esempio, per implementare una mappa che prende come chiave una tupla di stringhe, è necessario creare prima un modulo che implementi la `compare` sul tipo di dato `string*string`. Un'implementazione standard

può essere quella mostrata, cioè tramite un ordinamento lessicografico prima sulla prima stringa e, nel caso siano uguali, sulla seconda stringa.
Una volta creato il modulo `StringString` sarà possibile sfruttare il funtore `Map.Make` per ottenere un tipo mappa con chiave coppia di stringhe. In Haskell non è necessario tutto questo poiché le mappe e gli insiemi vengono già forniti come tipi, ad esempio scrivendo `Data.Map.Map (String, String) Int`, si ottiene una mappa con chiave una tupla di stringhe e valore un intero.

4.3 Generazione di codice

Dopo aver convertito il modulo parsing, si è passati al modulo di generazione di codice, che risulta la parte più corposa e più complessa. Per quanto riguarda questo modulo si è passati a convertire direttamente ocaml-gi-gtk. Come mostrato nel capitolo precedente, vengono dichiarate due monadi, una per la gestione dello stato per la generazione, una per la scrittura su file. La conversione più interessante è per l'appunto quella della monade `CodeGen`, la quale è stata gestita in maniera analoga a quella di parsing. Il codice monadico è stato convertito passando i tre stati alle funzioni necessarie. Si ricorda: i tre stati sono uno in sola lettura per leggere le configurazione attuale dell'ambiente, uno per la gestione delle variabili fresche e una per la memorizzazione delle informazioni riguardanti il modulo da scrivere. Il prossimo estratto di codice mostra la differenza tra la struttura del codice Haskell e la struttura del codice OCaml nella funzione `evalCodeGen`.

```haskell
runCodeGen
  :: BaseCodeGen e a
  -> CodeGenConfig
  -> (CGState, ModuleInfo)
  -> Either e (a, ModuleInfo)
runCodeGen cg cfg state_ = dropCGState
  <$> runExcept (runStateT (runReaderT cg cfg) state_)
 where
  dropCGState :: (a, (CGState, ModuleInfo)) -> (a, ModuleInfo)
  dropCGState (x, (_, m)) = (x, m)


```

```haskell
unwrapCodeGen
  :: CodeGen a -> CodeGenConfig -> (CGState, ModuleInfo) -> (a,
   ModuleInfo)
unwrapCodeGen cg cfg info = case runCodeGen cg cfg info of
  Left _             -> error "unwrapCodeGen:: The impossible happened!"
  Right (r, newInfo) -> (r, newInfo)


evalCodeGen
  :: Config -> M.Map Name API -> ModulePath -> CodeGen a -> (a,
   ModuleInfo)
evalCodeGen cfg apis mPath cg =
  let initialInfo = emptyModule mPath
      cfg'        = CodeGenConfig { hConfig    = cfg
                                  , loadedAPIs = apis
                                  , c2hMap     = cToHaskellMap (M.toList
    apis)
                                  }
  in  unwrapCodeGen cg cfg' (emptyCGState, initialInfo)
```

Listing 4.12: Funzioni base per la valutazione del codice monadico dei generatori

Haskell ha un approccio `lazy` che permette, tramite le monadi, di passare in giro descrizioni di computazioni che poi vengono eseguite a comando su input decisi al momento, come appunto mostrato sopra. Queste tre funzioni mostrano il funzionamento del codice monadico. La `runCodeGen` esegue le computazioni all'interno dello monade e restituisce solo i due stati interessanti, quelli scrivibili. La `evalCodeGen` è una funzione chiave: setta gli stati iniziali e lancia la computazione `cg` sugli stati appena creati. Nella conversione in OCaml è stata implementata la `evalCodeGen` in maniera simile ma senza avere a disposizione le monadi. Di conseguenza quello che è rappresentato da `cg` nel codice sopra, nel codice sotto è rappresentato dal parametro `action`, che non è altro che una funzione che prende in input degli stati e restituisce un risultato. Il comportamento monadico `lazy` è stato quindi simulato tramite una chiusura, uno dei modi standard per ottenere codice `lazy` in linguaggi `eager`.

```ocaml
(* config ->
api NameMap.t ->
```

```
module_path ->
(code_gen_config * cg_state * module_info -> module_info) ->
module_info) *)
let evalCodeGen cfg apis mPath action =
  let initialInfo = emptyModule mPath in
  let cfg' = {
    hConfig = cfg;
    loadedAPIs = apis;
    (*c2hMap =*)
  }
  in action (cfg', emptyCGState, initialInfo)
```

Listing 4.13: Funzione equivalente `evalCodeGen` in OCaml

Ad esempio, ogni volta che è necessario generare ricorsivamente il codice per un sottomodulo, in ocaml-gi-gtk viene chiamata la `recurseWithState`.

```
recurseWithState
  :: (CGState -> CGState) -> BaseCodeGen e a -> BaseCodeGen e (a, Code)
recurseWithState cgsSet cg = do
  cfg             <- ask
  (cgs, oldInfo) <- get
  -- Start the subgenerator with no code and no submodules.
  let info = cleanInfo oldInfo
  case runCodeGen cg cfg (cgsSet cgs, info) of
    Left e -> throwError e
    Right (r, new) ->
      put (cgs, mergeInfoState oldInfo new) >> return (r, moduleCode new
  )
```

Listing 4.14: Lancio di computazioni lazy su stati nuovi in Haskell

Questa resetta gli stati e li dà in input alla computazione. Quello che succede intuitivamente è che viene generato il codice a partire da uno stato "pulito" e poi viene fuso al modulo padre. In OCaml è stata implementata come segue.

```
let recurseWithState f oldInfo =
  let info = cleanInfo oldInfo in
  let cgstate, newInfo = f info in
```

```
  let code = newInfo.moduleCode in
  let minfo = mergeInfoState oldInfo newInfo in
  cgstate, minfo, code
```

Listing 4.15: Lancio di computazioni lazy su stati nuovi in OCaml

Il parametro `oldInfo` rappresenta lo stato che viene passato in giro e la `f` rappresenta la computazione da lanciare sui nuovi stati. Come mostrato, viene creato un nuovo modulo vuoto, applicata la `f` su esso, salvato il codice, restituito lo stato unendo quello vecchio con quello nuovo e restituito lo stato aggiornato insieme al codice, mantenendo la semantica del codice Haskell.
Analogamente, la gestione degli errori non bloccanti è eseguita in Haskell dalla funzione `handleCGExc`.

```
handleCGExc :: (CGError -> CodeGen a) -> ExcCodeGen a -> CodeGen a
handleCGExc fallback action = do
  cfg             <- ask
  (cgs, oldInfo) <- get
  let info = cleanInfo oldInfo
  case runCodeGen action cfg (cgs, info) of
    Left  e             -> fallback e
    Right (r, newInfo) -> do
      put (cgs, mergeInfo oldInfo newInfo)
      return r
```

Listing 4.16: Gestione monadica di errori non bloccanti

Questa prende in input una computazione che non può fallire e una che può fallire e lancia la seconda su uno stato pulito. Nel caso la seconda sollevi un'eccezione, viene lanciata la prima sullo stato vecchio. Questo permette di non inquinare lo stato nel caso in cui una computazione sollevi un'eccezione e abbia già scritto nello stato alcune informazioni. Questo comportamento non sarebbe ottenibile per esempio con una variabile globale, in quanto verrebbe comunque manipolata prima del sollevamento dell'eccezione.

```
let handleCGExc (cgstate, oldInfo) fallback action =
  let info = cleanInfo oldInfo in
  try
    let (cgstate, newInfo) = action cgstate info in
    cgstate, mergeInfo oldInfo newInfo
```

```
6    with CGError e -> fallback cgstate oldInfo e
```

Listing 4.17: Gestione tramite chiusure di errori non bloccanti

In OCaml è stato fatto lo stesso, utilizzando la gestione delle eccezioni al posto della monade `Either` e ovviamente passando lo stato vecchio esplicitamente alla funzione.

Capitolo 5

Risultati

Dopo una fase di debugging, la conversione del codice da Haskell a OCaml è riuscita, a meno dei seguenti aspetti:

- **Gestione degli overrides**: l'aspetto è importante ma non fondamentale per quanto riguarda la compilazione del codice. Questa parte non è stata implementata per motivi di tempo: era necessario tradurre anche funzioni di basso livello che si interfacciano con il codice C.

- **Fixup delle interfacce**: in ocaml-gi-gtk sono presenti delle funzioni che vanno a correggere alcuni errori dati dal parsing dei file `.gir`. Le funzioni hanno come prefisso `fixup` e sono implementate per structs, unions, fields, methods e interfaces. Sono state tutte implementate a parte quella per le interfacce, che necessita di convertire anch'essa codice che si allaccia alle librerie C. Questo implica una presenza errata di alcuni dati che nella generazione di codice si traduce nella mancanza di alcune classi.

- **Risoluzione dei cicli di dipendenze**: in ocaml-gi-gtk è stata inserita una routine che permette di bypassare il problema delle dipendenze cicliche tra i moduli OCaml del codice generato, andando a sostituire alcune definizione con delle definizioni equivalenti ma ricorsive. Questa problema si presenta solo alla compilazione di `Gdk` e `Gtk`, per cui si è tenuto da implementare per eventuali lavori futuri.

A parte gli aspetti citati sopra, il codice risulta corretto ed identico a quello generato da ocaml-gi-gtk. L'analisi dell'uguaglianza tra file è stata fatta utilizzando il comando `diff` di Unix. Le librerie che compilano correttamente sono:

- GLib
- GObject
- Atk
- Pango
- cairo
- GdkPixBuf

Mentre `Gio`, `Gdk` e `Gtk` non compilano poiché risultano mancanti quelle classi che necessitano del fixup delle interfacce. Una volta implementata quella parte di codice potrà compilare l'intera libreria GTK.

Conclusione

Nel panorama OCaml non esiste attualmente una libreria per la generazione automatica di binding GTK tramite GObject-Introspection. Esiste una libreria per Haskell a cui si è ispirato Alberto Nicolini insieme al prof. Sacerdoti Coen per produrre ocaml-gi-gtk, una libreria scritta in haskell per la generazione automatica di binding GTK per OCaml sfruttando GObject-Introspection. Questa tesi ha avuto l'obiettivo di convertire la repository di Nicolini in codice OCaml, in modo che sia più appetibile una potenziale adozione da parte della community di sviluppatori OCaml. Attraverso questo elaborato si è riusciti a convertire in OCaml il codice necessario per rendere compilabili le librerie GTK di più basso livello, e si è creato un parser OCaml di file `.gir` efficiente e che può essere utilizzato anche in modalità stand-alone su altri progetti. Gli sviluppi futuri possono includere l'aggiunta del fixup delle interfacce, la gestione degli overrides e si potranno convertire facilmente le eventuali modifiche e gli eventuali sviluppi apportati ad ocaml-gi-gtk, in quanto il codice è stato progettato per apparire molto simile.

Bibliografia

[1] *Haskell Text.XML Documentation.* URL: https://hackage.haskell.org/package/xml-conduit-1.9.1.1/docs/Text-XML.html.

[2] Stig Hackvän. *GTK history.* URL: https://web.archive.org/web/19990417052141/http://www.linuxworld.com/linuxworld/lw-1999-01/lw-01-gimp.html.

[3] *GTK structure.* URL: https://www.gtk.org/docs/architecture/.

[4] Stewart Weiss. *GTK reference counting.* URL: http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci493.70/lecture_notes/GTK_memory_mngmt.pdf.

[5] *GObject$_I$ntrospection.* URL: https://gi.readthedocs.io/en/latest/.

[6] *OCaml history.* URL: https://ocaml.org/learn/history.html.

[7] *OCaml garbage collector.* URL: https://dev.realworldocaml.org/garbage-collector.html.

[8] *Interfacing C with OCaml, Living in harmony with the garbage collector.* URL: https://ocaml.org/manual/intfc.html#s%5C%3Ac-gc-harmony.

[9] *Haskell Documentation.* URL: https://www.haskell.org/documentation/.

[10] *A Gentle Introduction to Haskell, About Monads.* URL: https://www.haskell.org/tutorial/monads.html.

[11] *haskell-gi.* URL: https://github.com/haskell-gi/haskell-gi.

[12] *lablgtk.* URL: https://github.com/garrigue/lablgtk.

[13] Claudio Sacerdoti Coen Alberto Nicolini. *Meta-programmazione di binding OCaml per GTK3*. URL: https://amslaurea.unibo.it/20508/1/Meta_programmazione_di_binding_OCaml_per_GTK3.pdf.

[14] Claudio Sacerdoti Coen Alberto Nicolini. *Repository: ocaml-gi-gtk*. URL: https://github.com/illbexyz/ocaml-gi-gtk.

[15] *GObject-Introspection annotations*. URL: https://gi.readthedocs.io/en/latest/annotations/giannotations.html.

[16] $Haskell_I O_f or_I mperative_P rogrammers$. URL: https://wiki.haskell.org/Haskell_IO_for_Imperative_Programmers.

[17] *Haskell Data.Sequences Documentation*. URL: https://hackage.haskell.org/package/containers-0.6.4.1/docs/Data-Sequence.html.

[18] *Haskell Control.Monad.Trans.RWS.CPS Documentation*. URL: https://hackage.haskell.org/package/transformers-0.5.6.2/docs/Control-Monad-Trans-RWS-CPS.html#v:modify.

[19] Cédric Le Moigne. *OCaml-GObject-Introspection*. URL: https://github.com/cedlemo/OCaml-GObject-Introspection.

[20] Nicolas Cannasse. *xml-light*. URL: http://tech.motion-twin.com/xmllight.

[21] *CS 3110 OCaml Style Guide*. URL: https://www.cs.cornell.edu/courses/cs3110/2018sp/handouts/style.html.

Ringraziamenti

Ringrazio infinitamente il prof. Sacerdoti Coen per la disponibilità e professionalità: capita raramente di trovare persone così competenti in materia, sia a livello teorico che operativo.

Ringrazio Eleonora per la pazienza di questi ultimi due anni e mezzo, sei stata fondamentale.

Ringrazio la mia famiglia per essermi sempre stata accanto.

Ringrazio i miei soci e il team Squiseat per la comprensione e la totale trasparenza.