Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

COMPILATORE PER LINGUAGGIO DI PROGRAMMAZIONE FUNZIONALE SPERIMENTALE

Relatore:
Chiar.mo Prof.
CLAUDIO SACERDOTI COEN

Presentata da:
LUCA BORGHI

Sessione
III sessione 2021-2022

Indice

1 Introduzione

Il paradigma funzionale nella programmazione non è certamente nuovo in ambito accademico: le sue origini risiedono a parecchie decine di anni fa, grazie agli studi di Alonzo Church sui formalismi del *lambda-calcolo* negli anni '30. Sebbene vi siano numerosi studi a livello accademico, il successo non è stato tale nel mondo industriale. Uno dei primi linguaggi funzionali fu Lisp, sviluppato a fine anni '50, il quale è stato, in un certo senso, pioniere di alcuni concetti e costrutti come la gestione automatica della memoria, funzioni di ordine superiore, la ricorsione, etc.. Successivamente, negli anni '70, citando i più famosi, nacquero Scheme (dialetto dello stesso Lisp) ed ML. Negli '80, è la volta di Erlang e Standard ML, nei '90 di OCaml, Haskell e Racket. Nonostante molti di questi linguaggi vengano considerati come general-purpose, non avranno inizialmente molto successo come lo hanno avuto, invece, le controparti dei linguaggi imperativi e dei linguaggi orientati agli oggetti. Un caso emblematico è quello di Erlang, il quale nacque nell'ambito della telefonia come linguaggio proprietario dell'azienda Ericsson. Successivamente, è stato adottato per molteplici tipi di applicativi, diventando un linguaggio di più ampio successo, soprattutto in ambiti in cui la gestione della concorrenza è molto importante. Negli ultimi anni, infatti, c'è stata una "riscoperta" del paradigma funzionale in quanto quest'ultimo permette di scrivere codice in stile dichiarativo e componibile, fornendo una maggiore manutenibilità e garantendo proprietà importanti dei programmi. Oltre al già citato Erlang, utilizzato ad esempio nello sviluppo dell'applicativo Whatsapp [30], altri linguaggi funzionali utilizzati nell'industria sono Haskell, con cui è stata implementata la famosa blockchain Cardano [31], OCaml, con cui è stato scritto il primo compilatore Rust ed è largamente utilizzato da Facebook in molti dei suoi software come Hack [32] e Flow [33], etc.. Inoltre, numerosi linguaggi di programmazione non propriamente funzionali ispirano alcuni dei loro costrutti al paradigma funzionale. Esempi ne sono il costrutto di pattern matching per Python e Rust oppure il costrutto di lambda-astrazione per C# e JavaScript oppure le funzioni come "cittadini di prima classe" per Lua e Go, etc..

I linguaggi di programmazione funzionali hanno spesso caratteristiche molto diverse tra loro: linguaggi come Erlang e Clojure hanno una tipizzazione dinamica, a differenza di altri linguaggi come Haskell e OCaml che effettuano un'analisi statica; Haskell e Elm vengono considerati come linguaggi puri, mentre OCaml, F# e molti altri linguaggi sono considerati come impuri; Lisp, Scheme, Closure e Racket sono linguaggi omoiconici che manipolano *S-expressions*, mentre altri linguaggi permettono la meta-programmazione utilizzando meccanismi differenti dalle *S-expressions*, un esempio è Template Haskell [34].

Il progetto di tesi, consiste nello sviluppo di un compilatore per un nuovo linguaggio di programmazione funzionale chiamato Fex. L'obiettivo iniziale è la costruzione di un linguaggio che racchiuda le caratteristiche più interessanti degli odierni linguaggi di programmazione funzionali. In particolare Fex deve avere un type-system statico che possa garantire proprietà importanti a tempo di compilazione. Inoltre, caratteristiche che romperebbero la "purità" del linguaggio devono essere gestite sempre tramite il sistema di tipi: un esempio ne sono gli effetti (cfr. Effetti). Tuttavia, gli obiettivi iniziali non sono stati completamente raggiunti, in particolare Fex risulta momentaneamente un linguaggio minimale le cui *features* sono riconducibili, per la maggior parte, a quelle di Haskell. Il codice sorgente è disponibile al link `https://github.com/bogo8liuk/Fex-lang` ([36]).

Lavoro iniziale Prima dello sviluppo del compilatore Fex, c'è stata una preparazione in vista del prodotto finale: vi sono stati alcuni brevi studi di fattibilità che riguardavano il nuovo linguaggio, in particolare sono stati scelti il target di compilazione e il linguaggio con il quale scrivere il software per il compilatore. Inizialmente, come target di compilazione è stata scelta la rappresentazione intermedia di Llvm [35] (Llvm IR) e OCaml come linguaggio per il compilatore. Tuttavia, benché il codice Llvm IR offra portabilità su molte architetture e, al contempo, buone prestazioni, questa prima scelta sul target di compilazione è stata scartata. Il motivo risiede nel fatto che Fex rappresenta un linguaggio ad altissimo livello e l'utilizzo di Llvm IR non renderebbe disponibile un garbage collector e, più in generale, un run-

time system. Ciò implicherebbe un oneroso lavoro per costruire l'astrazione esatta. Successivamente, è stata valutata un'altra opzione: utilizzare una rappresentazione intermedia di un altro linguaggio ad alto livello come target di compilazione. La scelta è ricaduta su Core, una rappresentazione intermedia di Haskell. Questa scelta ha due grandi vantaggi: avere a disposizione un run-time system (quello di Haskell); Core e Fex hanno costrutti molto simili. Come conseguenza è cambiato anche il linguaggio con cui sviluppare il compilatore: la scelta è ricaduta su Haskell, in quanto GHC (the Glasgow Haskell Compiler) espone delle API scritte in Haskell per costruire e manipolare programmi Core.

1.1 Note

Prima di presentare l'articolo, è necessario fare alcune precisazioni utili alla lettura:

- la sintassi degli identificatori di Fex è Haskell-like, quindi, tutto ciò che viene considerato come *variabile* nel linguaggio ha lettera iniziale minuscola, mentre tutto ciò che viene considerato come *valore* ha lettera iniziale maiuscola;
- i termini *constraint* e *predicato* sono considerati sinonimi (a meno di contesti particolari). Essi vengono intercambiati abbastanza liberamente;
- i termini *schema di tipo* e *poli-tipo* sono considerati sinonimi. Nell'articolo viene preferita per la maggior parte delle volte la prima forma;
- talvolta, è possibile che venga presentato codice sorgente (Haskell) del compilatore Fex, tuttavia, verranno isolati e mostrati solamente i punti più interessanti, rimuovendo i commenti e il codice non direttamente inerente al contesto. Altre volte, gli algoritmi e i concetti utilizzati nel compilatore vengono mostrati con uno pseudo-codice oppure con uno pseudo-linguaggio della logica. Questo per evitare di presentare dettagli implementativi poco utili ai fini delle spiegazioni;
- per quanto riguarda la struttura della tesi, nella prima parte, vengono presentate le principali caratteristiche di Fex senza entrare in eventuali dettagli implementativi, inoltre, verranno menzionati i formalismi principali a cui Fex si ispira. Dopodiché, verrà presentato il compilatore, il quale si divide in tre fasi principali:

  1. nella prima fase, vi è la descrizione della costruzione dell'albero di sintassi astratta e di tutti i controlli e le manipolazioni sull'albero stesso necessari per la seconda fase;
  2. nella seconda fase, appaiono le varie nozioni di tipo di Fex, in particolare vi è la generazione delle symbol-tables che conterranno i token "tipati";
  3. nella terza e ultima fase, vi è la generazione del codice Core che successivamente viene tradotto dal backend di Haskell in vari target di compilazione.

  Nell'ultima parte, vengono mostrati alcuni possibili sviluppi futuri di Fex, presentando le eventuali nuove caratteristiche e le possibili implementazioni nel compilatore Fex;
- nel corso dell'articolo, verranno mostrati degli schemi che raffigurano la struttura del compilatore. In generale, le freccie continue rappresentano una fase del compilatore, mentre le freccie tratteggiate rappresentano una dipendenza.

1.2 Haskell e System-F come modelli

Il compilatore è stato sviluppato utilizzando il linguaggio *Haskell*, anch'esso linguaggio di programmazione funzionale. Inoltre, il linguaggio target di Fex è *Core*, un linguaggio che estende *System-F* (cfr. System-F). Core, inoltre, viene utilizzato da GHC (compilatore Haskell) come rappresentazione intermedia di Haskell. All'interno di questo contesto, GHC espone delle API che permettono al cliente di utilizzare le funzionalità del compilatore [3]; è quindi possibile creare e manipolare programmi Core mediante le API, le quali sono scritte in Haskell. Per quest'ultimo motivo, Haskell è stato scelto come linguaggio di implementazione del compilatore di Fex, tuttavia, nel contesto del progetto, Haskell ha anche un altro importante ruolo: alcune delle sue funzionalità sono state, direttamente o indirettamente, fonte di ispirazione per il design di Fex. Nel prossimo paragrafo vengono presentate le principali caratteristiche di Fex ed è possibile ritrovare la maggior parte di esse anche in Haskell. Un altro linguaggio che è stato fonte di ispirazione, ma con minore impatto, è *OCaml*, linguaggio multi-paradigma (funzionale non puro), soprattutto per la sintassi di Fex e per il costrutto delle polymorphic variants (cfr. Varianti polimorfe).

System-F I costrutti principali di Fex si basano su System-F, una versione "tipata" del lambda-calcolo che introduce un meccanismo di quantificazione universale sui tipi. Esso è definito con i seguenti quattro costrutti:

$$e := x \mid e_1\ e_2 \mid \lambda x.\ e \mid let\ x = e_1\ in\ e_2$$

Come vedremo in seguito (cfr. Ricorsione), questo formalismo risulta non essere Turing-completo. Fex (come lo stesso Haskell) basa i propri costrutti e il proprio type-system, nonché il suo algoritmo di inferenza dei tipi, su questo formalismo, tuttavia, quest'ultimo viene esteso con altri costrutti che offrono una maggiore espressività.

1.3 Caratteristiche del linguaggio

Tra le principali caratteristiche del linguaggio vi sono:

Linguaggio funzionale puro Come Haskell, Fex è un linguaggio funzionale puro. La nozione di linguaggio di programmazione *funzionale* è piuttosto lasca e non vi è una vera e propria definizione formale, tant'è che il termine viene spesso utilizzato (e, talvolta, abusato) per indicare un linguaggio avente alcune particolari specifiche attribuibili al paradigma di programmazione funzionale. Fex può essere quindi considerato funzionale poiché, semplicemente, ha numerose caratteristiche proprie del paradigma funzionale, quali: tipi di dati algebrici, pattern matching, funzioni di ordine superiore, immutabilità, polimorfismo parametrico etc.. Per quanto riguarda la nozione di *purità*, nel paradigma funzionale viene fatta spesso la distinzione tra linguaggi puri e impuri; anche qui, non vi sono vere e proprie definizioni e la questione è spesso oggetto di controversie. Una proposta di definizione è stata fornita da Amr Sabry [1]: la purità ha a che fare con il passaggio dei parametri, in particolare, un linguaggio L può essere considerato puro se, dato un qualsiasi programma p scritto in L, al variare delle strategie di valutazione delle espressione di p (call-by-value, call-by-name, call-by-reference, call-by-need, etc.), non vi sono differenze osservabili, escludendo la divergenza.

Type-system statico con type-inference Fex è un linguaggio con type-system statico. Questo significa che le proprietà sul sistema di tipi vengono verificate a tempo di compilazione, infatti, la fase di type-checking garantisce la proprietà di type-safety. Il linguaggio permette anche di omettere le indicazioni di tipo (type-hinting) nella definizione di simboli; in caso non vi sia type-hinting per la definizione di un simbolo, il compilatore inferirà il tipo "più generale possibile" per il simbolo.

"Everything is an expression" Tutti i costrutti all'interno di Fex possono essere considerati espressioni prive di side-effects; non vi sono costrutti di controllo o costrutti che modificano variabili di stato esterne al contesto locale di un'espressione. I costrutti principali sono:

1. definizioni di: tipi, variabili, proprietà (che corrispondono alle type-classes di Haskell), istanze di proprietà, alias di tipi, firme di simboli; le definizioni di simboli hanno le seguenti grammatiche:

```
S :=
    let Var Args = E          --definizione

MS :=
    let Var MPM               --definizione con patterns

Args :=
    Var ... Var
```

2. espressioni date dalla seguente grammatica:

```
E :=
    Var                 --variabile
    Datacon             --data-constructor
    Literal             --letterale: stringa, numero o carattere
    E E                 --applicazione di espressione
    lam args -> E       --lambda-astrazione
    lam MPM             --lambda-astrazione con patterns
    S in E              --costrutto let..in
    MS in E             --costrutto let..in con patterns
    match E with PM     --pattern matching

PM :=
    ME -> E | ... | ME -> E

MPM :=
    | ME ... ME = E | ... | ME ... ME = E

ME :=                   --pattern
    Var                 --variabile
    _                   --caso default
    Literal             --letterale: stringa, numero o costante
    Datacon ME ... ME   --patterns applicati a un data-constructor
```

3. un costrutto, valutato a compile-time (cfr. Parsing degli operatori), per definire una "categoria" di operatori. Le categorie di operatori hanno degli identificatori per poterle nominare, inoltre definiscono le seguenti proprietà:

   (a) una lista di operatori appartenenti alla proprietà. Sono compresi gli identificatori di variabili (gli operatori stessi sono identificatori di variabili) che possono essere utilizzati con la sintassi infissa degli operatori;

   (b) una lista di categorie di operatori i quali avranno meno precedenza degli operatori della categoria corrente;

   (c) una lista di categorie di operatori i quali avranno più precedenza degli operatori della categoria corrente;

   (d) la fissità degli operatori.

Tipi di dati algebrici Fex supporta anche i tipi di dati algebrici (che sono l'unico modo per definire nuovi tipi). Con essi, vi è anche la nozione di *data constructor*, ovvero un costrutto dotato di tipo e al quale possono essere associati dei dati. Per eseguire operazioni con i tipi di dati algebrici vi è il costrutto del pattern matching che permette di destrutturare un data constructor dai suoi dati associati:

```
type Arith =
      Plus Expr Expr
    | Minus Expr Expr
    | Times Expr Expr
    | Number Int

let eval expr =
    match expr with
          Plus e1 e2 -> eval e1 + eval e2
        | Minus e1 e2 -> eval e1 - eval e2
        | Times e1 e2 -> eval e1 * eval e2
        | Number n -> n
```

Polimorfismo parametrico Il sistema di tipi del linguaggio si basa fortemente su quello di System-F, il quale, come è stato già menzionato in precedenza, introduce i quantificatori nei tipi. Quindi Fex fornisce il supporto per le variabili di tipo, inoltre, permette di quantificarle (pur con alcune restrizioni). I quantificatori nei tipi permettono ai token "tipabili" del linguaggio di possedere più tipi nello stesso momento, infatti, questo tipo di polimorfismo permette di avere singoli algoritmi per una moltitudine di tipi, ad esempio:

```
type List' a = Empty | Cons a (List' a)

let map
    | _ Empty -> Empty
    | f (Cons x t) -> Cons (f x) (map f t)
```

Tipi higher-kinded Il linguaggio permette la manipolazione di *funzioni di tipi*, introducendo la nozione di *kind* (cfr. Il sistema di tipi). Prima di mostrare degli esempi, è necessario fare una piccola nota sulla notazione che verrà utilizzata nell'articolo: in letteratura, con la terminologia *variabile di tipo* si intende un "placeholder" per un tipo che non necessita argomenti; in questo articolo, con tale notazione si indicherà, invece, un "placeholder" valido anche per tipi che necessitano argomenti. Data la seguente definizione di tipo:

```
type M a b = DataCon1 a | DataCon2 b | DataCon3 a Char
```

dove `a` e `b` sono variabili di tipo, è possibile, ad esempio, avere tipi della forma:

```
M Int Char
M a b
M String
M x
M
```

È possibile altresì avere un tipo della forma:

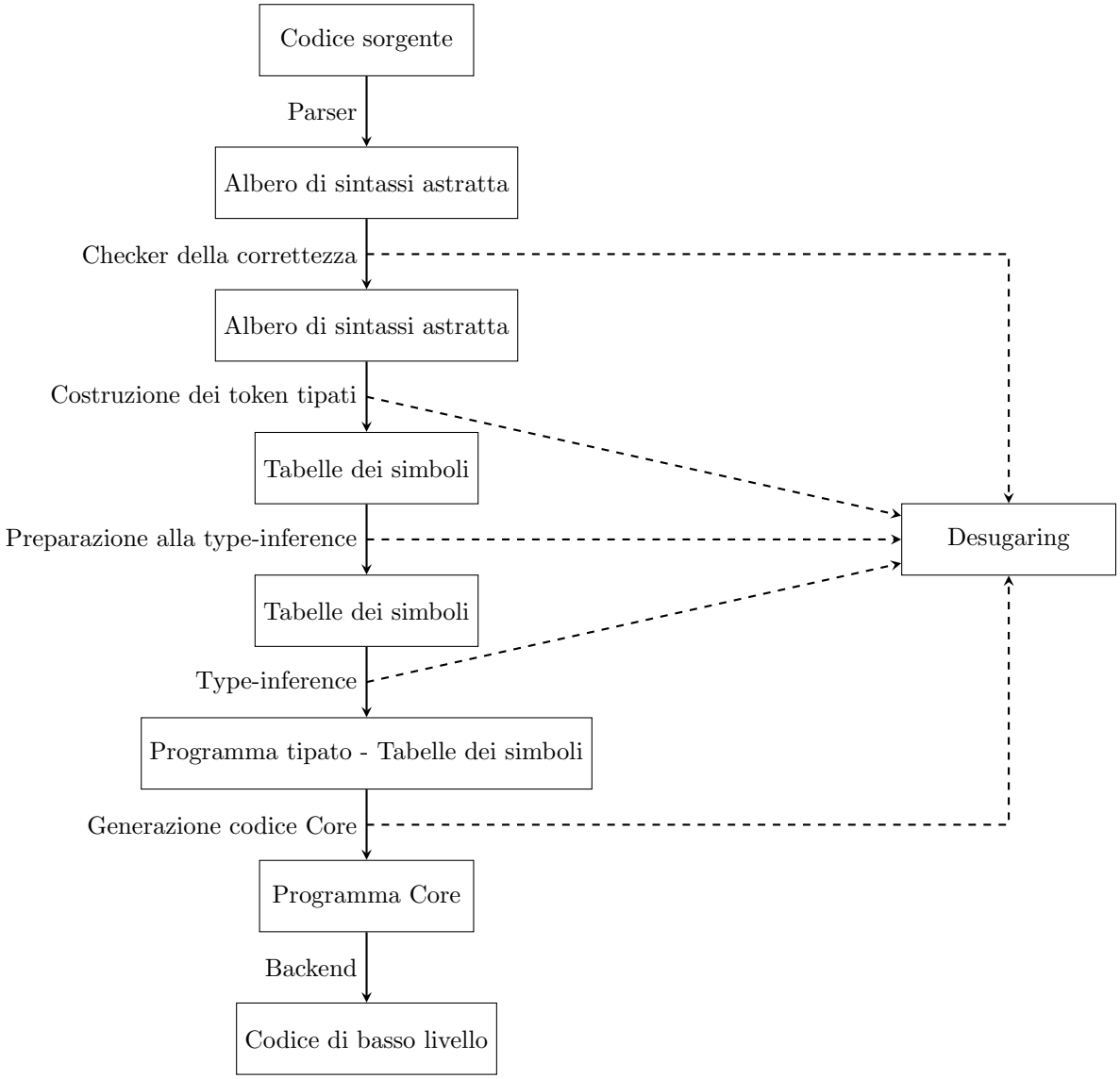
```
m Int
```

dove `m` è una variabile di tipo a cui viene applicata il tipo *Int*.

Polimorfismo ad-hoc Attraverso il meccanismo delle type-classes (all'interno del linguaggio vengono chiamate proprietà), è possibile creare funzioni polimorfe che possono essere applicate ad argomenti di tipi differenti e, a seconda dei tipi degli argomenti, viene "selezionata" un'implementazione. Ogni proprietà può avere uno, ma anche più tipi associati. Si osservi il seguente esempio:

```
type SuccessHttpCodes =
      Ok200
    | Moved301

type ErrorHttpCodes =
      Err401
    | Err404
    | Err500

property Show a =
    val show : a -> String
;;

instance Show SuccessHttpCodes =
    let show
        | Ok200 -> "Request success"
        | Moved301 -> "Resource moved permanently"
;;

instance Show ErrorHttpCodes =
    let show
        | Err401 -> "Unauthorized"
        | Err404 -> "Resource not found"
        | Err500 -> "Internal server error"
;;
```

1.4 Il compilatore



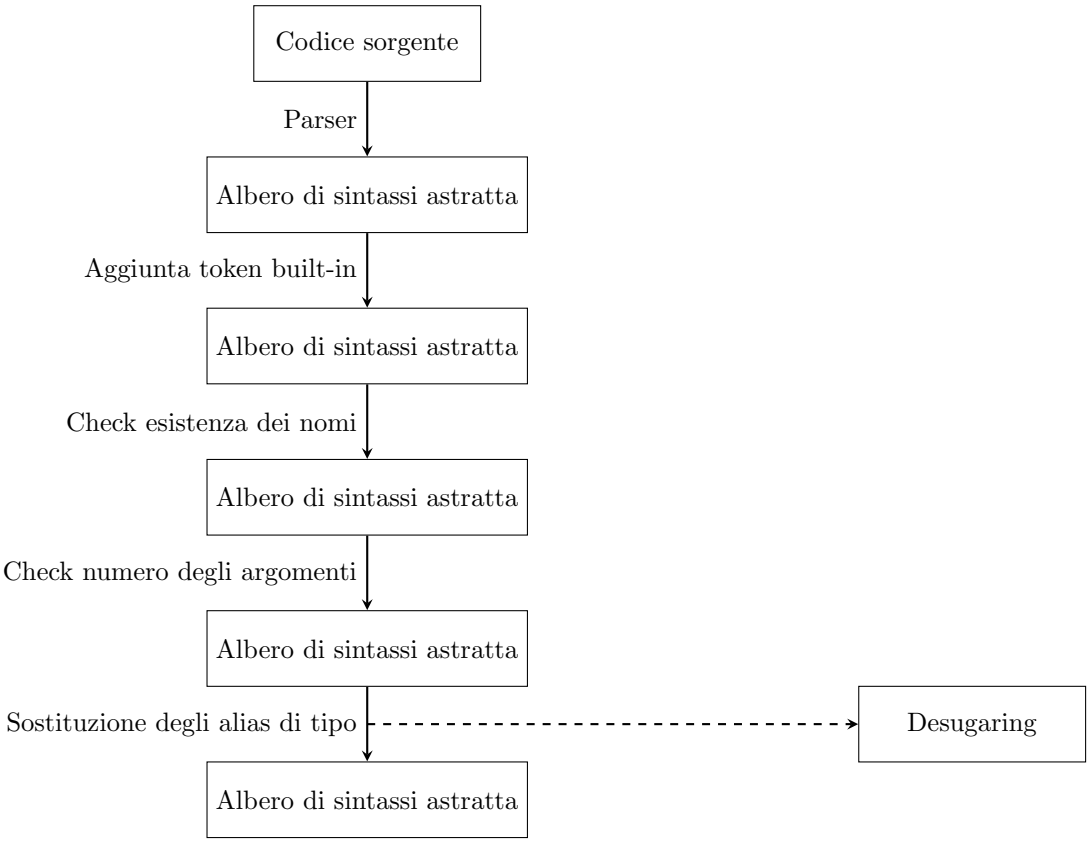
La struttura del compilatore è sequenziale, con eccezione fatta per la fase di *desugaring*. Quest'ultima si occupa di rimuovere il cosiddetto "zucchero sintattico" dalle strutture dati - che mantengono le informazioni del programma - che, durante le varie fasi di compilazione, possono subire semplificazioni. Talvolta, alcune semplificazioni vengono posticipate il più possibile per permettere al compilatore di restituire in output messaggi d'errore comprensibili per l'utente (cfr. Fasi di desugaring). Il modulo che si occupa del desugaring si presenta come una libreria che espone delle API che agiscono sulle strutture dati del compilatore. È compito del compilatore fare le chiamate ai vari sotto-moduli di desugaring. Nel corso dell'articolo, vi è una più ampia e precisa descrizione delle singole fasi di compilazione, con enfasi particolare sugli algoritmi più interessanti utilizzati per risolvere i singoli sottoproblemi e su come le varie fasi interagiscono tra loro.

2 L'albero di sintassi astratta

Nella prima parte del compilatore viene eseguito il parsing del sorgente e, se non vi sono errori di sintassi, viene prodotto l'albero di sintassi astratta (AST). Il codice che gestisce i "token" dell'AST è all'interno del modulo `Compiler.Ast.Tree`; l'entry-point dell'albero è dato dal token `Program`, il quale, come nodi figli, ha delle `Declaration` che rappresentano i principali costrutti globali del linguaggio.

```
newtype Program a = Program [Declaration a]

data Declaration a =
      ADT (AlgebraicDataType a)
    | AliasADT (AliasAlgebraicDataType a)
    | Intf (Interface a)
    | Ins (Instance a)
    | Sig (Signature a)
    | Let (SymbolDeclaration a)
    | LetMulti (MultiSymbolDeclaration a)
```

Di seguito vi è lo schema della prima parte del compilatore:



2.1 Il parser

Il parser è la prima componente del compilatore (dopo la lettura del sorgente). È stato scritto mediante la libreria open-source `Parsec` [4], la quale si basa sul concetto di parser combinator monadico. Il codice risiede nel modulo `Compiler.Syntax` ed ha una struttura gerarchica: il codice di più "basso livello"

definisce i combinators di "pezzi" primitivi dei token dell'AST (`Compiler.Syntax.Lib.SimpleParser`), dopodiché, nel modulo `Compiler.Syntax.Grammar` vi è la generazione vera e propria dei token dell'AST e, infine, vi è l'entry-point del parser, ovvero `Compiler.Syntax.Parser`. È bene notare che all'interno del modulo `Compiler.Syntax.Lib.SimpleParser` non sono visibili le API dell'AST, in quanto esso si occupa soltanto del parsing dei costrutti del linguaggio e non della generazione dei token.

2.1.1 Parsing degli operatori

Fex permette di definire operatori che vengono trattati come simboli di variabili. In questo contesto, il linguaggio espone all'utente un costrutto particolare che permettere di definire le proprietà degli eventuali operatori. Questo costrutto viene valutato durante il parsing e definisce una categoria di operatori, ecco un esempio:

```
OPERATORS_CATEGORY {#
    name : Application ;
    operators : |>, <|, `applyTo ;
    lesser than : Comparison, Numeric ;
    greater than : Functor ;
    fixity : InfixLeft ;
#}
```

Il primo "campo" è il nome della categoria ed è utile per poter identificare la categoria, infatti, il terzo e il quarto campo definiscono rispetto a quali categorie gli attuali operatori hanno, rispettivamente, meno o più precedenza. Il secondo campo definisce l'insieme di operatori che fanno parte della categoria. Infine, l'ultimo campo è una costante e definisce la fissità degli operatori: infissa senza associazione, infissa con associazione a sinistra, infissa con associazione a destra, postfissa o prefissa. La versione infissa è solo per gli operatori binari, mentre postfissa e prefissa solo per gli operatori unari. Si può notare che nella lista di operatori vi è anche un simbolo di variabile preceduto dal carattere di backtick; questo è possibile, in quanto Fex permette di utilizzare i simboli di variabili come operatori facendoli precedere dal carattere backtick. La valutazione di questo costrutto è interamente integrata nel parser ed è obbligatorio per l'utente definire le categorie degli operatori all'inizio del sorgente. La libreria `Parsec` offre delle API anche per il parsing degli operatori; l'algoritmo di gestione delle categorie si occupa di costruire una tabella di operatori, implementata semplicemente come una lista di liste, ordinando i gruppi (di categorie) in base alla loro precedenza. Le informazioni sulle categorie vengono estrapolate dal parsing del costrutto e vengono passate successivamente all'algoritmo di ordinamento delle categorie. Quest'ultimo si può ridurre all'inserimento di un elemento in una lista di liste:

```
insert(x, ll):
    match ll with
        [] -> [[x]]
        (l :: lt) ->
            if any x' in l. x' < x        // Condizione d'inserimento (1)
            then
                if areAmbigous(x, ll)
                then fail
                else [x] :: l :: lt
            else if all x' in l. x' == x  // Condizione d'inserimento (2)
            then
                if areAmbigous(x,ll)
                then fail
                else subInsert(x, l) :: lt
            else l :: insert(x, lt)
```

L'algoritmo scorre la lista finché non:

1. trova una sottolista l in cui esiste almeno un elemento minore di x; in questo caso, controlla eventuali ambiguità delle categorie, poiché l'utente potrebbe aver definito categorie tali che la nozione di ordinamento tra loro non è transitiva. La funzione `areAmbigous` nel pezzo di pseudo-codice si occupa di eseguire questo controllo. Se non esistono ambiguità tra le categorie, allora viene creata una nuova lista singoletto contenente x che viene inserita davanti alla sottolista l
2. trova una sottolista l in cui tutti gli elementi sono uguali a x; in questo caso, x viene inserito in l, eseguendo sempre prima il controllo sulle ambiguità.

In tutti gli altri casi, la corrente sottolista l contiene almeno un elemento x' tale che $x' > x$, quindi l'inserimento non può ancora avvenire. Quando la tabella è completa, essa viene passata alla funzione `buildExpressionParser` della libreria `Parsec` che si occupa di costruire il parser per le espressioni.

2.2 Check dei nomi

Dopo la fase di parsing e dell'aggiunta di token built-in del linguaggio, viene effettuato il controllo di esistenza di ogni tipo di simbolo: nomi di tipo, nomi di variabili, nomi di proprietà, etc.. Perciò deve valere la seguente condizione:

$$\forall name \in AST.\ \exists def(name) \in AST$$

dove *def* è la funzione che ritorna la definizione di un token dell'AST. Questo tipo di check è molto importante in quanto fasi successive del compilatore basano le loro computazioni sull'ipotesi che tutti i simboli siano stati definiti. Il codice risiede nel modulo `Compiler.Args`.

2.3 Check degli argomenti

Il controllo degli argomenti viene effettuato, a differenza del check dei nomi che viene eseguito su ogni tipo di nome, solo sui nomi di tipo, di alias e di proprietà. In particolare, devono valere le seguenti condizioni:

$$\forall name \in AST.$$

1. :
$$isTypeName(name) \implies \#args(name) \leq \#args(def(name))$$
2. :
$$isAliasName(name) \implies \#args(name) = \#args(def(name))$$
3. :
$$isPropertyName(name) \implies \#args(name) = \#args(def(name))$$

dove *isTypeName*, *isAliasName*, *isPropertyName* sono le funzioni che ritornano *true* se il nome in input è, rispettivamente, un nome di tipo, un nome di alias, un nome di proprietà, *false* altrimenti, *args* è la funzione che calcola gli argomenti di un token e il simbolo $\#$ è la funzione che calcola la cardinalità di un insieme. In tutti e tre i casi, gli argomenti di un simbolo sono una sequenza di variabili di tipo, ad esempio:

```
type Map k v = Entry (Tuple2 k v) | Node (Tuple2 k v) (Map k v) (Map k v)

alias IntMap v = Map Int v

property Existence e a =
    val exists : e -> a -> Bool
;;
```

Gli argomenti di `Map` sono `k` e `v`, quelli di `IntMap` sono `v`, mentre quelli di `Existence` sono `e` e `a`. La condizione (1) è meno stringente di (2) e di (3), in quanto l'utente può "manipolare" non solo tipi, ma anche funzioni di tipi, anche conosciute come *type constructor*. La condizione (2) è fondamentale per la prossima fase del compilatore. Il codice risiede nel modulo `Compiler.Args`.

2.4 Prime fasi di desugaring

Prima della generazione dei token "tipati", il compilatore si occupa di effettuare alcune fasi di desugaring. Esse si occupano dell'eliminazione o dell'aggiornamento di alcuni costrutti presenti in un programma per semplificare le fasi successive del compilatore.

2.4.1 Eliminazione degli alias di tipo

Fex offre un costrutto che permette all'utente di definire alias di tipo. Ecco un esempio:

```
alias CharAnd x = Tuple2 Char x
```

Questo tipo di costrutto viene completamente valutato in fase di compilazione: ogni occorrenza di nome di alias viene trattata come una vera e propria *macro*, quindi viene sostituita con il tipo associato all'alias. Il modulo di Desugaring si occupa di questo task (`Compiler.Desugar.Alias`). Come si nota nell'esempio, gli alias ammettono argomenti (variabili di tipo) che vengono passati alla funzione di tipo. Il linguaggio ammette anche alias di alias, tuttavia, ciò può portare a *cicli*, come ad esempio:

```
alias A = B
alias B = C
alias C = A
```

Per questo motivo, l'algoritmo di sostituzione degli alias implementa anche la "cycle detection": se esiste un ciclo di alias, il programma viene rifiutata. Inoltre, nella sostituzione è fondamentale che valga la condizione sugli alias nel check degli argomenti (cfr. Check degli argomenti):

$$\forall name \in AST.\ isAliasName(name) \implies \#args(name) = \#args(def(name))$$

Se la sopracitata condizione non fosse vera, non sarebbe possibile effettuare l'unificazione nella kind-inference (cfr. kind-inference).

2.4.2 Eliminazione delle firme di funzioni

Fex espone un costrutto, detto *signature* (in italiano "firma"), che permette di indicare il tipo di un binding. Ad esempio:

```
val id : a -> a
```

L'esempio appena mostrato indica che la variabile *id* ha tipo $\forall\alpha.\alpha \to \alpha$. Questo tipo di costrutto non è nient'altro che "zucchero sintattico" per il type-hinting dei binding. I token dei binding - `SymbolDeclaration` e `MultiSymbolDeclaration` - possono possedere delle informazioni sul proprio tipo, ad esempio, guardando la definizione di `MultiSymbolDeclaration`:

```
data MultiSymbolDeclaration a =
    MultiSymTok (SymbolName a) (Hint a) (MultiPatternMatch a) a
```

si può notare come il costruttore `MultiSymTok` prenda in input un token `Hint`. Il task del modulo `Compiler.Desugar.Sigs` è di eliminare dall'AST i costrutti `Signature` che rappresentano, appunto, le firme delle variabili e aggiungere l'informazione sul tipo contenuta in esse come type-hinting delle definizioni dei bindings.

2.4.3 Renaming delle variabili di tipo

Prima di iniziare le fasi di costruzione dei token "tipati", dato un programma P, esso viene visitato e le occorrenze delle variabili di tipo vengono tutte rinominate in modo che:

- per ogni token $t, t' \in P$, tali che t non sia lo stesso token di t', vale:

$$\forall v \in binders(t).\ \neg\exists v' \in binders(t').\ v = v'$$

dove *binders* è la funzione che ritorna le variabili di tipo che fungono da binders nel token in questione, mentre il test di uguaglianza tra variabili di tipo è definito come il test di uguaglianza dei letterali che rappresentano gli identificatori delle variabili. Chiaramente, le variabili legate vengono cambiate in base a come vengono aggiornati i binders, ad esempio, si consideri la seguente definizione:

```
type T a b = C1 Int a | C2 a (T a b)
```

Se viene effettuata la seguente sostituzione:

$$\{a \mapsto a_1, b \mapsto a_2\}$$

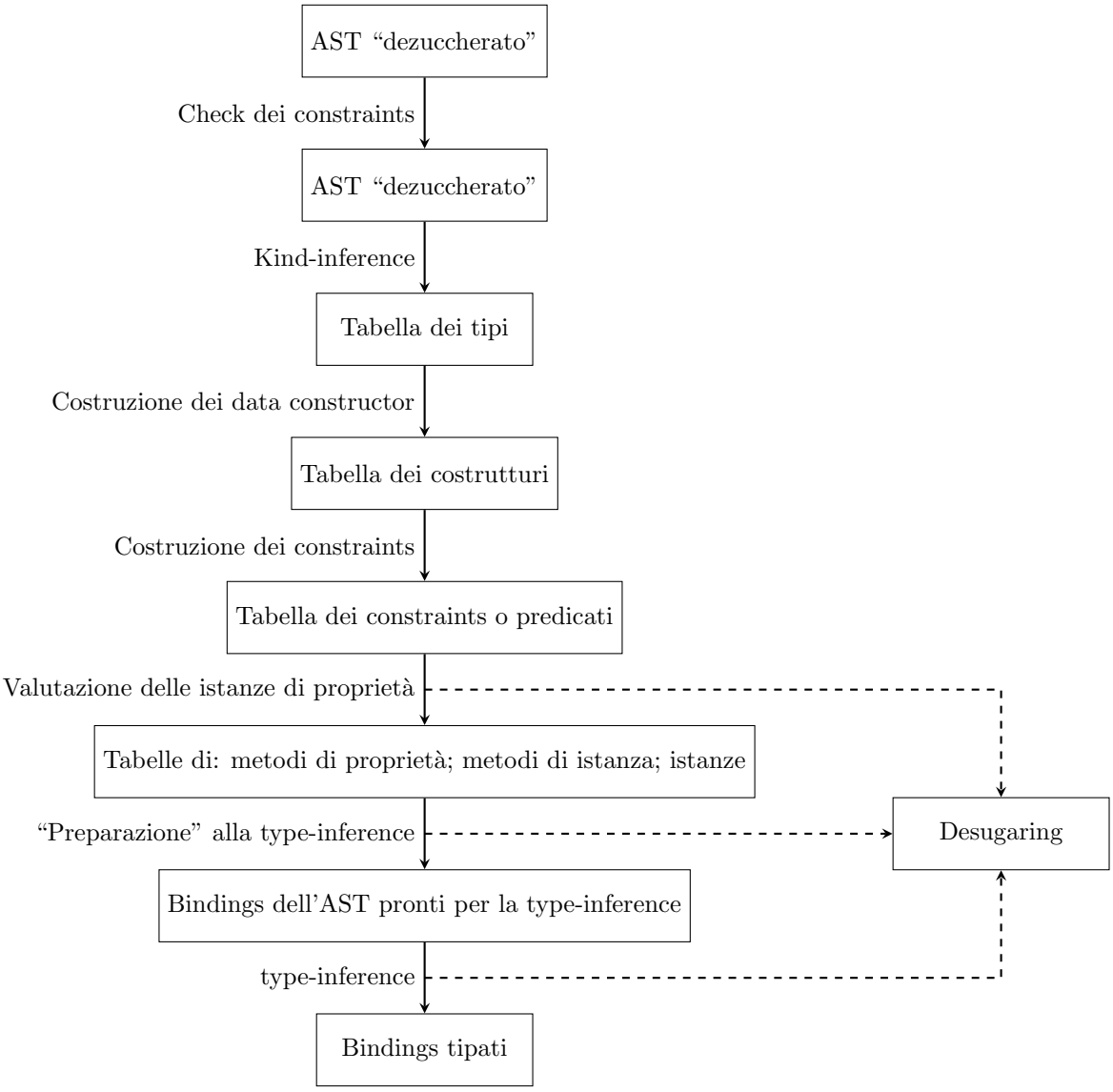
Allora la nuova definizione di T avrà la seguente forma:

```
type T a1 a2 = C1 Int a1 | C2 a1 (T a1 a2)
```

Il motivo di questo renaming verrà descritto in seguito (cfr. Kind-inference). Il codice risiede nel modulo `Compiler.Desugar.TyVars`.

3 Generazione dei token “tipati”

Dopo la generazione dell’albero di sintassi astratta e alcune fasi di desugaring, questa seconda componente del compilatore si occupa della generazione dei token “tipati”. Questi ultimi prendono questa nomea in quanto, a questo livello, compare la nozione di tipo del linguaggio.



3.1 Approccio a tabelle

A differenza della prima componente del compilatore, dove l'unica struttura dati di primo livello era l'AST, in questo caso vi sono molteplici strutture dati. Innanzitutto, nel modulo `Compiler.Ast.Typed`, vi sono le definizioni di tutti i token tipati e le operazioni su di essi; proprio in questo modulo compaiono:

1. le nozioni che riguardano i tipi del linguaggio:

```
data LangKind                --kind
data LangVarType a           --variabile di tipo
data LangHigherType a        --mono-tipo
data LangSpecConstraint a    --"constraint" o predicato
data LangQualType a          --mono-tipo qualificato
data LangTypeScheme a        --poli-tipo o schema di tipo
```

   La nozione di *kind* serve per aggiungere un'informazione ai tipi; in Fex, non vi è alcun costrutto per esprimere delle espressioni che hanno un kind come informazione aggiuntiva. Un *mono-tipo* è, come vedremo, un tipo in cui possono comparire variabili di tipo, le quali sono tutte quantificati esistenzialmente, ad esempio:

$$\exists \alpha, \beta.\ \alpha \to \beta$$

   La nozione di *predicato* rappresenta, invece, un'affermazione su un tipo e ciò è utile per restringere l'insieme di tipi adatto ad istanziare un determinato mono-tipo; come vedremo in seguito, i predicati vengono espressi attraverso il meccanismo delle *proprietà*, ad esempio, nel seguente pezzo di codice, il token `Ord` serve per esprimere predicati:

```
property Ord a =
    <metodi di proprieta'>
;;
```

   Un *mono-tipo qualificato* è un tipo sul quale vi sono uno o più predicati. Uno *schema di tipo* è un tipo che ammette variabili di tipo quantificate universalmente:

$$\forall \alpha, \beta.\ \alpha \to \beta$$

   Si faccia riferimento alla sezione sul sistema di tipi per le definizioni formali (cfr. Il sistema di tipi).

2. i token tipati che costituiscono un programma:

```
data NotedVar a            --variabili
data NotedVal a            --valori: letterali e data-constructor
data NotedMatchExpr a      --espressioni per il pattern match
data NotedExpr a           --expressioni
```

   È bene fare una precisazione su `NotedMatchExpr`. Considerando il sotto-costrutto di un *caso* del costrutto di pattern matching, sia esso della forma:

$$pattern \to expr$$

   il token `NotedMatchExpr` rappresenta un *pattern*.

3. operazioni che riguardano la manipolazione dei tipi quali unificazione, test di specificità, specializzazione, instanziazione, generalizzazione. Inoltre, vi sono le funzioni e le strutture dati per gestire il dispatch statico.

Nonostante la presenza di token tipati, non esiste una corrispondente versione tipata dell'AST con un unico entry-point, bensì le informazioni che riguardano un programma vengono memorizzate in tabelle (cfr. modulo `Compiler.Types.Tables`):

```
newtype TypesTable a          --tabella dei type-constructor
newtype DataConsTable a       --tabella dei data-constructor
newtype ConstraintsTable a    --tabella dei constraint-constructor
newtype InstsTable a          --tabella dei metodi delle istanze
newtype PropMethodsTable a    --tabella dei metodi di proprieta'
newtype ImplTable a           --tabella delle istanze
data    TypedProgram a        --tabella dei bindings tipati
```

Vedremo nel dettaglio ogni tabella nelle descrizioni delle varie fasi del compilatore. Tuttavia, è necessaria una nota su `TypesTable` e `ConstraintsTable`. Come si legge dai commenti, esse sono tabelle per memorizzare dei costruttori. Tali costruttori sono necessari alla costruzione dei tipi e dei predicati all'interno di un programma. Ad esempio, date le seguenti definizioni in Fex:

```
type Box a = Boxing a
property Stateful m =
    val getState : m a -> a
;;
```

Verranno creati dei token (presenti in `Compiler.Ast.Typed`):

```
data LangNewType a            --type-constructor
data LangNewConstraint a      --constraint-constructor
```

che rappresenteranno rispettivamente il modello per tipi `Box` e il modello per constraints `Stateful` e che verranno memorizzati nelle suddette tabelle. Mentre, quanto riguarda `Boxing`, esso rappresenta un data-construcotr.

Sempre nel modulo `Compiler.Types.Tables`, viene definito anche il cosiddetto "binding tipato":

```
type BindingSingleton a = (NotedVar a, [NotedVar a], NotedExpr a)
data TypedBinding a =
      TyNonRec (BindingSingleton a)
    | TyRec [BindingSingleton a]
```

Osservando l'implementazione di `TypedBinding`, si nota come esistano due tipi di binding. Il primo è per i binding non ricorsivi, mentre il secondo è per i binding che sono mutualmente ricorsivi fra loro (cfr. Type-inference). In seguito, vedremo perché si rende necessaria questa distinzione.

3.1.1 Confronto con GHC

Come è stato detto precedentemente, l'approccio del compilatore è quello di costruire tabelle man mano che le informazioni vengono inferite dall'AST. GHC (the Glasgow Haskell Compiler) utilizza un approccio differente, in quanto non utilizza alcuna "symbol table", bensì ogni token tipato (di GHC) nella compilazione di un programma Haskell può puntare ad altri token tipati [5]. Si crea così un grafo di strutture dati tipate. Ad esempio, GHC, per gestire le entità di type constructor e data constructor, utilizza rispettivamente i token `TyCon` e `DataCon` (si ricordi che GHC è scritto in Haskell):

```
data TyCon
data DataCon
```

Ogni token di tipo `TyCon` punterà a una lista di `DataCon` che, a loro volta, conterranno la referenza al loro costruttore di tipo. Come puntualizza Edward Y. Zang nell'introduzione dell'articolo [6], uno svantaggio di questo approccio è che il grafo è immutabile e quindi, per poter aggiornare i nodi del grafo è necessario ricostruire il grafo da zero. Tuttavia, questo problema è mitigato, in quanto gli aggiornamenti del grafo

sono parecchio rari, inoltre, man mano che GHC ottiene informazioni dal programma Haskell, accrescerà il grafo senza aggiornare i nodi preesistenti; in questo modo, non vi è alcuna necessità di costruire il grafo da zero. La scelta, nel compilatore Fex, è stata quella di un approccio a tabelle proprio per evitare casi di mutua ricorsione tra tipi e, quindi, di codice potenzialmente più complesso per l'aggiornamento dei dati che riguardano i token del programma. Infatti, sebbene alcune tabelle, una volta costruite, vengano utilizzate in modalità di sola lettura, altre tabelle, come `TypedProgram`, vengono aggiornate più volte durante le varie fasi del compilatore.

3.2 Il sistema di tipi

In questa sezione, verrà presentato il sistema di tipi di Fex. Come è stato già menzionato in precedenza, il linguaggio supporta il polimorfismo ad hoc e il polimorfismo parametrico; quest'ultimo viene implementato attraverso il concetto di variabile di tipo, la quale può essere considerata come un "placeholder" per i tipi. Il sistema di tipi che verrà presentato è fortemente basato su quello di System-F, tuttavia, vi sono alcune estensioni che permettono una maggiore espressività.

3.2.1 Mono-tipo

Di seguito, viene definito il concetto di *mono-tipo*:

$$MT \ := \ \alpha \mid T \mid MT \ MT$$

dove α è una variabile di tipo e T è un costruttore di tipo. Una condizione fondamentale sui mono-tipi è l'esistenza del tipo funzione ($\rightarrow$) nel sistema di tipi. Nella sezione sull'inferenza di tipo (cfr. Inferenza di lambda astrazione) verrà mostrato il motivo per il quale è fondamentale garantire la presenza del tipo funzione. Come si può dedurre dalla definizione stessa, non si deve confondere il concetto di mono-tipo con quello di tipo *monomorfo*, infatti i mono-tipi, a differenza dei tipi monomorfi, ammettono le variabili di tipo. Nel compilatore, la definizione di mono-tipo si trova nel modulo `Compiler.Ast.Typed`:

```
data LangHigherType a =
      LTy (LangType a)
    | HApp [LangHigherType a] a
```

dove `LangType` rappresenta tutti i tipi che possono essere definiti in Fex. Si noti come vi è un secondo caso (`HApp`), il quale rappresenta il tipo funzione. Il costruttore `HApp` prende in input una lista di mono-tipi, tuttavia il tipo funzione può avere al massimo due argomenti, perciò il modulo `Compiler.Ast.Typed` si occupa internamente di rifiutare qualsiasi valore con costruttore `HApp` che ha più di due argomenti. La causa per la quale non vi è un numero fissato di argomenti è la presenza delle funzioni di tipo, infatti, in questo modo, è possibile esprimere senza difficoltà, all'interno del compilatore, tipi come:

- `(->) a`
- `(->)`

3.2.2 Tipi higher-kinded

Fex permette all'utente di utilizzare *funzioni di tipo* (o *costruttori di tipo*). A questo scopo, viene introdotta la nozione di *kind*, il quale rappresenta un'informazione aggiuntiva per i tipi. Informalmente, dato un type-system *TS*, un kind-system *KS* si può vedere come un type-system di livello "superiore" a *TS*. Ora diamo una definizione più precisa di kind:

$$K \ := \ \kappa \mid \ * \mid K \rightarrow K$$

dove κ è una variabile di kind e $*$ è la costante primitiva di kind (detta anche "tipo"), il quale rappresenta tutti quei tipi che non hanno bisogno di parametri. Ecco alcuni esempi:

- $* \to *$ è il kind delle funzioni di tipo unarie;
- $(* \to *) \to * \to *$ è il kind delle funzioni di tipo binarie che prendono in input una funzione di tipo unaria e un tipo e ritornano un altro tipo.
- $\kappa_1 \to \kappa_2$ è il kind delle funzioni unarie che prendono in input una funzione di tipo di kind κ_1 e ritorna un'altra funzione di tipo di kind κ_2.

Data la presenza di tipi *higher-kinded* all'interno di Fex, è doveroso fare un'ulteriore precisazione che riguarda i mono-tipi: alla definizione di mono-tipo deve essere aggiunta la condizione sulla correttezza dei kind. Ad esempio, dato il tipo:

$$t_1 \to t_2$$

sapendo che il kind del costruttore di tipo funzione è $* \to * \to *$, è necessario, affinché il suddetto kind venga rispettato, che il kind di t_1 e t_2 sia $*$. Nel codice sorgente, la definizione di kind nel compilatore Fex si trova in `Compiler.Ast.Typed`:

```
data LangKind =
      LKVar String
    | LKConst
    | SubLK [LangKind]
```

Il costruttore `LKVar` rappresenta le variabili di kind, il costruttore `LKConst` rappresenta il kind *tipo* $*$, mentre il costruttore `SubLK` rappresenta l'applicazione di kinds.

3.2.3 Subtyping: predicati e tipi qualificati

La nozione di mono-tipo ammette le variabili di tipo, le quali fungono da placeholder per qualsiasi tipo dello stesso kind della variabile. Tuttavia, solamente con la nozione di mono-tipo non è possibile fare asserzioni sulle variabili di tipi, se non, appunto, che sono placeholder adatti a qualsiasi tipo. Fex supporta anche una nozione di sottotipo. Prendiamo, ad esempio, la seguente variabile di tipo:

$$\alpha : *$$

la notazione $: *$ serve, in questo caso, per rendere esplicito il kind della variabile. α è un placeholder adatto a tipi quali, ad esempio, `Int`, `List b` oppure `List Char` poiché hanno tutti kind $*$. α è un placeholder molto lasco, in quanto è adatto per tutti quei tipi di kind $*$, quindi si potrebbero fare asserzioni su α, ad esempio, si può asserire che α è un placeholder valido per tutti quei tipi S che sono sottotipi di un certo tipo T. Una definizione informale di sottotipo è la seguente: se S è sottotipo di T (scriviamo $S \leq T$), allora ogni termine di S può essere utilizzato in maniera *safe* in ogni *contesto* in cui un termine di T è richiesto, dove le definizioni di *safe* e *contesto* sono dipendenti dal formalismo. Aggiungiamo una notazione per descrivere questa nozione:

$$\alpha \leq T.\, T'$$

Possiamo quindi restringere i possibili tipi adatti a "riempire" il placeholder rappresentato da α. Più in generale, possiamo considerare $\alpha \leq T$ come un predicato P su α. Utilizzeremo, quindi, la seguente notazione:

$$P(\alpha) \Rightarrow T'$$

Ora abbiamo una sintassi per descrivere *predicati* o *constraint* sulle variabili di tipo. Tale comportamento si può estendere a qualsiasi forma di tipo, non solo alle variabili di tipo, ma per farlo è necessario prima definire la sintassi dei predicati. In Fex, un predicato o constraint ha la seguente forma:

$$C := P\ MT_1\ ...\ MT_n$$

dove P è il nome di una proprietà (cfr. Polimorfismo ad-hoc). Ora possiamo quindi definire un'estensione dei mono-tipi, in modo che su questi ultimi si possano aggiungere delle ipotesi. Per una definizione più generale possibile, ammettiamo che su un mono-tipo si possa applicare un numero arbitrario di predicati:

$$QT := MT \mid C \Rightarrow QT$$

Questa appena data è la nozione di *tipo qualificato*. La definizione si trova nel modulo `Compiler.Ast.Typed`:

```
data LangQualType a = Qual [LangSpecConstraint a] (LangHigherType a)
```

3.2.4 Schemi di tipo e polimorfismo parametrico

Finora abbiamo trattato mono-tipi e tipi qualificati, tuttavia, la sola presenza delle variabili di tipo non è sufficiente a costruire *schemi di tipi*. Si consideri il seguente esempio:

$$map : (\alpha \to \beta) \to List\ \alpha \to List\ \beta$$

In questo caso, il tipo della funzione *map* non è uno *schema di tipo*, in quanto le variabili di tipo α e β rappresentano un tipo fissato non ancora conosciuto. Per ottenere il polimorfismo parametrico, è necessario introdurre dei quantificatori, ad esempio, possiamo rifinire il tipo di *map* nel modo seguente:

$$map : \forall \alpha, \beta.\ (\alpha \to \beta) \to List\ \alpha \to List\ \beta$$

A differenza di prima, in cui *map* aveva un tipo prefissato, qui può avere molteplici tipi. Ora diamo la definizione di *schema di tipo* o *poli-tipo* in Fex:

$$PT := QT \mid \forall \alpha.\ PT$$

Questa definizione lascia spazio alla presenza di *variabili libere*, in quanto non necessariamente tutte le variabili di tipo che compaiono in un tipo qualificato sono legate a un quantificatore. Le variabili libere vengono trattate come costanti. Si noti come i quantificatori possono apparire solamente alla testa di un tipo, ad esempio, il seguente tipo non è accettato in Fex:

$$\forall \alpha.\ (\forall \alpha.\ \alpha \to \alpha) \to \alpha$$

La definizione di schema di tipo in Fex si trova nel modulo `Compiler.Ast.Typed`:

```
data LangTypeScheme a = Forall [LangVarType a] (LangQualType a)
```

La politica sulla sintassi degli schemi di tipi di Fex è omettere il quantificatore $\forall$ nelle indicazione sul tipo. Per questo, ogni qual volta vi è un costrutto di type-hinting (firme comprese), il tipo viene considerato come uno schema di tipo in cui tutte le variabili di tipo che compaiono vengono legate a un quantificatore universale, ad esempio:

```
val map : (a -> b) -> List a -> List b
```

in questo caso la funzione `map` avrà il seguente tipo:

$$map : \forall \alpha, \beta.\ (\alpha \to \beta) \to List\ \alpha \to List\ \beta$$

3.2.5 Test di uguaglianza tra tipi

Il test di uguaglianza tra tipi si differenzia parecchio tra le diverse nozioni di "tipo", inoltre, non sempre risulterà utile avere una definizione di uguaglianza tra alcune nozioni di tipo. Partendo dalla nozione di mono-tipo, due mono-tipi τ e τ' sono uguali se hanno i termini identici. Dato che Fex supporta la nozione di mono-tipo higher-kinded, è banale asserire che:

$$kindOf(\tau) \neq kindOf(\tau') \Longrightarrow \tau \neq \tau'$$

Per quanto riguarda i tipi qualificati, è necessario prendere in considerazione anche i predicati che precedono un mono-tipo. Definiamo, quindi, il test di uguaglianza tra predicati. Siano $(C\ \tau_1\ ...\ \tau_n)$ e $(C'\ \tau'_1\ ...\ \tau'_m)$ due constraints, essi sono uguali se:

$$C = C' \wedge n = m \wedge \forall i \in \{1, ..., n\}.\ \tau_i = \tau'_i$$

Ora possiamo definire il test uguaglianza per i tipi qualificati. Siano $(P_1, ..., P_n \Rightarrow \tau)$ e $(Q_1, ..., Q_m \Rightarrow \tau')$ due tipi qualificati, essi sono uguali se:

1. $\tau = \tau'$
2. $\forall i \in \{1, ..., n\}.\ \exists j \in \{1, ..., m\}.\ P_i = Q_j$
3. $\forall j \in \{1, ..., m\}.\ \exists i \in \{1, ..., n\}.\ Q_j = P_i$

Ora rimangono gli schemi di tipo. A differenza dei mono-tipi, qui le variabili possono essere quantificate, quindi non basterebbe più definire il test di uguaglianza analizzando solamente i termini del mono-tipo. Nel codice sorgente in `Compiler.Ast.Typed`, non vi è alcuna definizione di uguaglianza tra schemi di tipo poiché questa non è necessaria a nessun'altra operazione.

3.3 Polimorfismo ad-hoc

Fex supporta il polimorfismo ad-hoc esponendo costrutti chiamati *proprietà*, le quali sono concettualmente molto simili alle *type classes* di Haskell. Ecco un esempio di definizione di proprietà in Fex:

```
property Eq a =
    val (==) : a -> a -> Bool
    val (/=) : a -> a -> Bool
;;
```

Questo pezzo di codice produce idealmente due token corrispondenti ai metodi `(==)` e `(/=)`, i quali possono essere istanziati attraverso il meccanismo delle istanze che verrà presentato prossimamente. I token appena costruiti avranno le seguenti firme:

```
val (==) : Eq a => a -> a -> Bool
val (/=) : Eq a => a -> a -> Bool
```

Si noti come le firme effettive non siano le stesse di quelle date dall'utente, le quali risultano incomplete. All'inizio della dicitura dei tipi effettivi, si può notare un *constraint* o *predicato* seguito da una freccia, la quale è intesa come implicazione. I tipi dei metodi di proprietà avranno quindi la forma:

$$Pred(\overline{\alpha}) \Rightarrow ty(\overline{\alpha})$$

dove *Pred* è un predicato e *ty* è un tipo qualunque sulle variabili $\overline{\alpha}$. Il polimorfismo ad-hoc viene, dunque, supportato attraverso i predicati, i quali rappresentano delle ipotesi aggiuntive sui tipi. L'istanziazione di una proprietà consiste nel dichiarare dei tipi che fungeranno da "caso specifico" per i metodi di proprietà (i "modelli"), ad esempio, riprendendo la proprietà `Eq` del precedente pezzo di codice, in Fex possiamo definire un'istanza del tipo:

```
instance Eq Char =
    <implementazioni>
;;
```

I dettagli implementativi sono omessi. In ogni caso, i metodi di istanza avranno le seguenti firme:

```
val (==) : Char -> Char -> Bool
val (/=) : Char -> Char -> Bool
```

Si può notare come il predicato su `Eq` non esista più e ciò è dovuto all'istanziazione, ovvero è stato fissato un tipo per il quale vale il predicato `Eq`.

3.4 Estensioni del polimorfismo ad-hoc

Il meccanismo di polimorfismo ad-hoc precedentemente presentato incontra numerose estensioni in Fex, alcune delle quali impongono condizioni aggiuntive sul programma.

3.4.1 Numero arbitrario di argomenti delle proprietà

Nell'esempio con `Eq`, la proprietà possedeva soltanto un argomento, tuttavia, è possibile definire proprietà con un numero arbitrario (ma fissato alla definizione) di argomenti ad esempio:

```
property Stream s t =
    val new : t -> s t
    val yield : t -> s t -> s t
    val consume : s t -> (s t, t)
;;
```

Si noti che `Stream` ha 2 argomenti.

3.4.2 Constraints sulle proprietà

In Fex, è possibile definire una proprietà della forma:

```
property C2 a = C1 a Char =>
    <metodi>
;;
```

Questo tipo di definizione consiste nell'aggiungere il predicato `C1 a Char` alla proprietà `C2 a`. Semanticamente, questo impone la seguente condizione sul programma:

$$\forall type.\ \exists inst(C2, type) \implies \exists inst(C1, type, Char)$$

dove $inst(C, \overline{t})$ è, banalmente, l'instanza con nome di proprietà C e tipi $\overline{t}$. È possibile aggiungere più di un predicato alla stessa proprietà.

3.4.3 Polimorfismo higher-kinded

Riprendiamo il precedente esempio con `Stream`:

```
property Stream s t =
    val new : t -> s t
    val yield : t -> s t -> s t
    val consume : s t -> (s t, t)
;;
```

Gli argomenti di una proprietà sono sempre variabili di tipo. Come è già stato menzionato in precedenza, Fex supporta tipi di kind arbitrari; gli argomenti delle proprietà non fanno eccezione da questo punto di vista, infatti, nell'esempio si può notare come la variabile s abbia kind:

$$* \rightarrow *$$

mentre la variabile t ha kind:

$$*$$

3.4.4 Argomenti arbitrari delle istanze

Gli argomenti di un'istanza possono essere tipi di forma arbitraria, ad esempio:

```
property C x y =
    <metodi>
;;

instance C a (Tuple3 a (T b) (m b)) =
    <implementazioni>
;;
```

Confrontando il precedente codice Fex con il seguente codice Haskell che può considerarsi quanto meno concettualmente equivalente:

```
{-# LANGUAGE MultiParamTypeClasses #-}

class C x y where
    <metodi>

instance C a (a, T b, m b) where
    <implementazioni>
```

si ha che, compilando questo programma Haskell soltanto con l'estensione all'inizio del codice, si incorrerà nel seguente messaggio d'errore da parte del compilatore:

```
Illegal instance declaration for 'C a (a, T b, m b)'
(All instance types must be of the form (T a1 ... an)
where a1 ... an are *distinct type variables*,
and each type variable appears at most once in the instance head.
Use FlexibleInstances if you want to disable this.)
```

Come suggerisce l'ultima riga del messaggio d'errore, per definire un'istanza del genere in un programma Haskell è necessario utilizzare l'estensione `FlexibleInstances` [7]. In Fex, a differenza di Haskell, quel tipo di instanza è accettato di default. Il seguente vincolo sulla forma dei tipi riportato nel messaggio d'errore viene dunque rilassato:

$$TyCon(\overline{\alpha})$$

dove $\overline{\alpha}$ sono variabili di tipo distinte e $TyCon$ è un costruttore di tipo. In seguito, verrà mostrato quali vincoli (che riguardano i constraints e le istanze) possono essere rilassati e con quali conseguenze.

3.4.5 Constraints sulle istanze

L'utente ha anche la facoltà di definire istanze con uno o più predicati, ad esempio:

```
instance Stream m String = Monad m =>
    <implementazioni>
;;
```

questo tipo di definizione genererà metodi di istanza con le seguenti firme:

```
val new : Monad m => String -> m String
val yield : Monad m => String -> m String -> m String
val consume : Monad m => m String -> (m String, String)
```

Il constraint sull'istanza viene quindi applicato al tipo dei metodi di istanza.

3.5 Condizioni sui constraints

I constraints rappresentano un modo per "restringere" i possibili tipi che possono istanziare una o più variabili di tipo. I tipi nelle firme delle variabili (nonché nei type-hinting) possono possedere zero o più constraint; lo stesso vale, come viene mostrato nei paragrafi precedenti, anche per le proprietà e per le istanze. Esistono, però, dei vincoli sulla forma dei constraint. Come viene mostrato in [8] e [9], esistono condizioni sufficienti affinché l'algoritmo di risoluzione delle istanze di Haskell termini. Prima di presentarle, diamo alcune notazioni:
Dato un costrutto K della forma $D \Rightarrow C$, definiamo

- C come *Contesto* del costrutto K;
- H come *Testa* del costrutto K.

Ora presentiamo le *condizioni di Paterson*:

- il contesto C di una dichiarazione di type-class può menzionare solamente variabili di tipo e le variabili di tipo sono distinte in ogni singolo constraint in C;
- Per ogni dichiarazione di istanza $TyCl\ t_1\ ...\ t_n \Rightarrow C$, nessuna variabile di tipo ha più occorrenze nel contesto C rispetto alla testa $TyCl\ t_1\ ...\ t_n$.
- Per ogni dichiarazione di istanza $TyCl\ t_1\ ...\ t_n \Rightarrow C$, ogni constraint nel contesto C ha meno costruttori e variabili di tipo (presi insieme e contando le ripetizioni) rispetto alla testa $TyCl\ t_1 ... t_n$.
- Per ogni due dichiarazioni di istanze $TyCl\ t_1\ ...\ t_n \Rightarrow C$, $TyCl\ t'_1\ ...\ t'_n \Rightarrow C'$, non deve esistere una sostituzione φ tale che: $\varphi(t_1) = \varphi(t'_1)$, ..., $\varphi(t_n) = \varphi(t'_n)$.

Informalmente, l'obiettivo è avere sempre dei contesti più "piccoli" rispetto alle teste. La politica di Fex è simile e applica i seguenti vincoli:

1. Per ogni constraint c, c deve contenere almeno una variabile di tipo;
2. Per ogni istanza $h \Rightarrow ctx$, il numero di occorrenze di ogni singola variabile v nel contesto ctx deve essere minore o uguale rispetto al numero di occorrenze di v nella testa h;
3. Per ogni istanza $h \Rightarrow ctx$, le variabili di tipo di ctx devono essere innestate in meno costruttori di tipo rispetto alle variabili di tipo di h (contando le variabili tutte insieme).

Il codice che implementa questi controlli risiede nel modulo `Compiler.Constraints.Check`.

3.6 Kind-inference

La kind-inference in Fex è necessaria per poter costruire i token che rappresentano i costruttori di tipi. Prima di entrare nei dettagli, è bene fare alcune precisazioni sui kind. Innanzitutto, Fex non espone all'utente alcuna sintassi per manipolare i kind o per scrivere espressioni con i kind. Come conseguenza, la definizione di kind in Fex (cfr. Tipi higher-kinded) non viene estesa con dei quantificatori. Questo

non avviene per i tipi, infatti, la nozione di mono-tipo viene successivamente estesa con la nozione di schema di tipo, la quale introduce il polimorfismo parametrico.

L'ipotesi fondamentale (che connoteremo con HK) su cui si basa la kind-inference è sul kind del tipo *funzione*:

$$(\rightarrow) : * \rightarrow * \rightarrow *$$

In generale, l'inferenza di kind avviene "osservando" le definizioni dei tipi di dati algebrici, in particolare, guardando le definizioni dei data-constructors. La definizione di un data-constructor ha la seguente forma:

$$C\ ty_1\ ...\ ty_n$$

Supponendo che il data-constructor C *costruisca* il tipo $T\ \alpha_1\ ...\ \alpha_m$, allora C avrà tipo (cfr. Data-constructors):

$$ty_1 \rightarrow ... \rightarrow ty_n \rightarrow T\ \alpha_1\ ...\ \alpha_m$$

Come conseguenza dell'ipotesi HK si ha che:

$$\forall i \in \{1, ..., n\}.\ ty_i : *$$

e:

$$T\ \alpha_1\ ...\ \alpha_m : *$$

Si ricordi che le variabili di tipo che compaiono nei tipi ty_i sono legate ai binders $\alpha_1\ ...\ \alpha_m$. Una volta che tutti i data-constructors sono stati visitati, i kind delle variabili di tipo saranno noti, quindi si può inferire il kind del costruttore di tipo T. Durante la kind-inference, viene utilizzato un ambiente di tipizzazione (che connoteremo con Δ) in cui vengono salvate le coppie della forma:

$$\langle tipo, kind \rangle$$

dove il *tipo* è il nome di un qualsiasi tipo che compare senza parametri; sono comprese le variabili di tipo. In questo contesto, i nomi dei tipi non hanno uno *scope* in quanto è garantito che non ci siano tipi diversi con identificatori uguali nel programma. Questa condizione è banalmente garantita per i tipi concreti in quanto non è possibile definire tipi molteplici con nomi uguali, mentre, per quanto riguarda le variabili di tipo, il renaming delle variabili di tipo (cfr. Renaming delle variabili di tipo) garantisce la condizione. Inoltre, la sintassi degli identificatori garantisce che non vi siano variabili di tipo e tipi concreti con nomi uguali. In generale, vengono visitate tutte le definizioni di data-constructors nelle definizioni di tipo, seguendo queste regole:

- se un tipo ty_i ha la forma di variabile di tipo α_j, per un qualche $j \in 1, ..., m$, allora per HK, $\alpha_j : *$;
- se un tipo ty_i ha come testa un tipo concreto T' che esiste già in Δ, allora si possono inferire i kind degli argomenti di ty_i; per i tipi che esistono già in Δ e che non hanno una variabile di kind come kind associato, viene effettuato il kind-check, mentre per i tipi non presenti in Δ o che hanno una variabile di kind come kind associato, vengono inseriti in Δ con il nuovo kind inferito. Infine, deve valere che $ty_i : *$;
- se un tipo ty_i ha come testa un tipo concreto T' che non esiste in Δ, allora si inferiscono i kind degli argomenti di ty_i, senza effettuare la fase di kind-check. Infine, T' viene inserito in Δ con kind $K_1 \rightarrow ... \rightarrow K_r$, dove $K_1, ..., K_r$ sono i kind degli argomenti applicati a T';
- se un tipo ty_i ha come testa una variabile di tipo m, deve valere che m non compare come suo argomento diretto. Nel caso non valga la condizione, il programma viene respinto con un errore. Poi l'inferenza procede come per i tipi concreti;

- se un tipo arg compare come argomento i-esimo di un costruttore di tipo $tycon$, se $tycon \in \Delta$, allora sia K il suo kind, il kind inferito di arg sarà quello dell'i-esimo argomento di K. Se $tycon \notin \Delta$, allora viene creata una variabile di kind κ e in Δ viene inserito il binding $\langle arg, \kappa \rangle$;
- le variabili di tipo che compaiono soltanto nei binders avranno kind $*$;
- dopo che tutte le definizioni di tipo sono state visitate, le variabili di tipo in Δ che hanno variabili di kind come kind associato, acquisiscono il kind $*$. Questo previene un ulteriore livello di polimorfismo;
- ogni qual volta un tipo che ha una variabile di kind κ come kind associato acquisisce un nuovo kind K, le occorrenze di κ in Δ come variabili libere devono essere *specializzate* con K.

3.7 Data-constructors

Una delle tabelle di simboli che appare nel modulo `Compiler.Types.Table` è `DataConsTable`. In essa vengono salvati i *data constructor* associati ai loro tipi. I costruttori - che nell'AST vengono rappresentati dal token `ADTConstructor` - vengono trasformati nel token `NotedVal` presente nel modulo `Compiler.Ast.Typed` che consiste nella seguente coppia:

$$\langle dataConRep, type \rangle$$

dove *dataConRep* è la rappresentazione sotto forma di stringa del data constructor, mentre ty è il tipo del costruttore. Per quanto riguarda il tipo del costruttore, prendiamo in considerazione il seguente esempio di codice Fex:

```
type Foo x y = Bar Int (M x) y
```

In questa definizione di tipo di dato algebrico, vi è un solo costruttore: `Bar`. Il suo tipo è il seguente:

$$\forall x, y.\ Int \to M\ x \to y \to Foo\ x\ y$$

Il tipo di ritorno è sempre dato dalla definizione del tipo. Il modulo che si occupa di trasformare i token `ADTConstructor` in `NotedVal` e aggiungerli nella tabella `DataConsTable` è `Compiler.Types.Builder.Cons`.

3.8 Constraint constructors

Dopo il check sui constraints e la generazione di data-constructors, il compilatore si occupa di generare i cosiddetti "*constraint constructors*", ovvero token che fungono da modelli per la costruzione di predicati. Il modulo che li genera è `Compiler.Types.Tables`; esso prende in input i token dell'AST che rappresentano le proprietà (`Interface`) e da essi costruisce i token tipati `LangNewConstraint` e li inserisce nella tabella `ConstraintsTable`. È compito di questo modulo controllare che le classi non formino cicli tra loro, ad esempio, il seguente programma viene rifiutato dal compilatore:

```
property Foo a = Bar (M a) Int =>
    <metodi>
;;

property Bar x y = Foo (K x Char) =>
    <metodi>
;;
```

Si noti come i vincoli sui constraints vengano tutti rispettati.

3.9 Costruzione delle istanze

Nel modulo `Compiler.Types.Builder.Instances` vengono create le seguenti 3 tabelle:

- `InstsTable`, ovvero la tabella che contiene i bindings (sotto forma di token dell'AST) delle istanze;
- `PropMethodsTable`, ovvero la tabella che contiene i metodi delle proprietà sotto forma di token tipati;
- `ImplTable`, ovvero la tabella che contiene i constraints che derivano dalle istanze definite dall'utente;

Per quanto riguarda l'ultima tabella, data la definizione di un'istanza:

```
instance Eq Char =
    <implementazioni>
;;
```

la testa `Eq Char` ha la forma di un constraint, nonostante non rispetti uno dei vincoli sui constraints (un constraint deve avere almeno una variabile di tipo), infatti l'istanza viene salvata sotto forma di constraint (`LangSpecConstraint`). Le tre tabelle vengono create in un unico modulo per una questione di efficienza (i token delle istanze vengono visitati una sola volta). Inoltre, in questo modulo viene effettuato il controllo del vincolo sull'esistenza dei predicati delle proprietà (cfr. Constraints sulle proprietà). Prima di inserire i metodi delle istanze nella tabella `InstsTable`, viene eseguita una fase di desugaring su di essi. Il problema nasce dal fatto che, data una proprietà, può esistere un numero arbitrario di istanze e con esse, un numero arbitrario di metodi con lo stesso nome, perciò è necessario identificare univocamente i metodi di ogni singoli istanza. Il modulo `Compiler.Desugar.Names` si occupa di, dato il nome di una variabile e una sequenza di constraints (gli argomenti di un'istanza), creare un identificatore univoco che non può essere uguale a nessun altro identificatore nel programma.

3.10 Preparazione alla type-inference

Prima di effettuare la type-inference è necessario fare alcune considerazioni ed eseguire alcune computazioni. Innazitutto, il type-system di riferimento è *Hindley-Milner* (o Damas-Hindley-Milner, in seguito lo indicheremo con HM) con alcune estensioni che riguardano il costrutto del pattern matching, l'inferenza di definizioni ricorsive e la risoluzione delle istanze. Il modulo che si occupa della preparazione alla type-inference è `Compiler.Types.Prepare` Introdurremo il type-system in modo preciso nella sezione Type-inference, tuttavia, ora presentiamo alcune caratteristiche del type-system che è bene conoscere come premessa alla preparazione della type-inference. Innanzitutto, HM viene descritto formalmente da un insieme di *regole* le quali si occupano di "tipare" espressioni di un formalismo fissato: una versione del lambda-calcolo estesa con il costrutto "let..in". Una delle regole è quella sull'inferenza del tipo di un simbolo (la regola la chiameremo **Var**) la quale, informalmente, sostiene che: data l'ipotesi di un simbolo x con schema di tipo σ nel contesto di un ambiente di tipizzazione Γ e τ l'istanziazione del tipo σ, si conclude che il simbolo x ha tipo τ. Quindi, si ha che nel momento in cui viene incontrato un simbolo x in un'espressione, esso deve essere presente nell'ambiente di tipizzazione Γ, al quale, informalmente, si può pensare come una mappa di simboli e tipi associati ai simboli. Un'altra regola utile è quella sull'inferenza delle definizioni mutualmente ricorsive. In particolare, essa sostiene che, l'inferenza di definizioni mutualmente ricorsive avviene in "gruppo", ovvero un insieme $f_1, ..., f_n$ di simboli mutualmente ricorsivi tra loro viene considerato come un'unica definizione (regola **Rec**) (cfr. Ricorsione). Queste regole serviranno come premessa ad alcune fasi nella preparazione della type inference.

3.10.1 Metodi delle istanze

Vale la pena notare come in Core non esista un costrutto particolare per i metodi delle istanze, infatti, il costrutto di GHC che rappresenta le istanze (`ClsInst`) non porta con sè informazioni che riguardano i metodi [10], perciò, è in un qualche modo necessario gestirli. La politica del compilatore è quella di aggiungerli nell'insieme di bindings del programma e trattarli come qualsiasi altra funzione. La presenza di eventuali conflitti tra gli identificatori è stata già risolta (cfr. Costruzione delle istanze).

3.10.2 Simboli innestati univoci

Prima della type-inference, vengono visitate tutte le espressioni dei bindings e vengono creati nuovi identificatori per ogni definizione innestata di variabile (costrutto `let..in`). I nuovi identificatori sono resi univoci in tutto il programma. Il modulo che crea e garantisce che i nuovi identificatori siano univoci è `Compiler.Desugar.Names`. La proprietà di unicità degli identificatori innestati è molto importante, vedremo in seguito il motivo (cfr. Type-inference).

3.10.3 Clusters di definizioni mutualmente ricorsive

Come è stato già menzionato precedentemente, le definizioni mutualmente ricorsive devono essere considerate come insiemi e non singolarmente. Perciò è necessario distinguere due tipi di bindings, quelli mutualmente ricorsivi e i restanti:

```
data RawBinding =
      RawNonRec (Raw.SDUnion With.ProgState)
    | RawRec [Raw.SDUnion With.ProgState]
```

La precedente definizione è nel modulo `Compiler.Types.Prepare.Lib` e divide i bindings dell'AST in bindings non ricorsivi e bindings mutualmente ricorsivi. Rimane, quindi, da dividere i bindings del programma. Prima di presentare l'algoritmo, bisogna effettuare una precisazione molto importante. In precedenza, abbiamo aggiunto i bindings delle istanze alla lista dei bindings del programma, cambiando, però, gli identificatori, rendendoli univoci all'interno del programma. Per riconoscere i bindings mutualmente ricorsivi è necessaria una funzione che calcoli le dipendenze di una definizione. Nelle espressioni, in generale, possono essere menzionati i metodi delle proprietà, ma non possono, in alcun modo, essere menzionati i metodi delle istanze. Inoltre, il tipo dei metodi delle proprietà è sempre noto a priori e non è possibile, prima della type-inference, inferire le istanze (cfr. Dispatch statico) corrette dei metodi. Nel calcolo delle dipendenze, i metodi di proprietà possono, quindi, essere esclusi. La funzione *depsOf*, la quale prende in input un binding `b` dell'AST e la tabella `PropMethodsTable`, calcolerà le dipendenze di `b`, escludendo le dipendenze che provengono dalla tabella dei metodi di proprietà. L'insieme dei bindings di un programma può essere visto come un grafo orientato G in cui:

- i nodi sono i bindings;
- gli archi sono le dipendenze di un binding.

Il problema di trovare i clusters di definizioni mutualmente ricorsive corrisponde a trovare le *componenti fortemente connesse* del grafo G. Il modulo che implementa l'algoritmo è `Compiler.Types.Prepare.Recursion` e utilizza la libreria `Data.Graph` ([26]).

3.10.4 Sorting dei bindings

A questo punto, abbiamo una lista di `RawBinding`. Possiamo fare la seguente osservazione: i bindings possono essere visti come un grafo orientato aciclico G, in cui, come prima:

- i nodi sono i bindings;

- gli archi sono le dipendenze di un binding.

L'unica differenza rispetto a prima è l'aciclicità del grafo. Questa proprietà è garantita dal seguente fatto: se esistesse un ciclo in G, allora l'algoritmo di raggruppamento dei clusters l'avrebbe trovato e avrebbe creato un cluster di definizioni mutualmente ricorsive tra loro. La regola di inferenza **Var** impone che si conosca sempre lo schema di tipo di un simbolo, perciò, è necessario, prima di effettuare la type-inference, ordinare i bindings in base alle loro dipendenze. L'ordinamento viene effettuato nel modulo `Compiler.Types.Prepare.Sort` e l'algoritmo utilizzato è un'estensione dell'insertion-sort, dove, la nozione di ordinamento è data dalle dipendenze dei bindings, ovvero:
Siano b e b' due bindings, si ha che:

- $b > b'$, se b' è una dipendenza diretta di b oppure esistono dei bindings $c_1, ..., c_n$ tali che: $(b > c_1) \wedge (c_1 > c_2) \wedge ... \wedge (c_{n-1} > c_n) \wedge (c_n > b')$;
- $b = b'$, se la componente di b in G non è raggiungibile da nessun nodo della componente b'.

la proprietà di transitività di questa nozione di ordinamento è garantita dall'assenza di cicli all'interno del grafo. Di seguito vi è l'algoritmo di sorting di una singola componente del grafo:

```
sortComponent(bindings, remaining, component):
    match bindings, remaining with
        [], [] -> component
        [], (_ : _) -> sortComponent remaining [] component
        (b : t), _ ->
            let (inserted, component') = tryInsert b component in
                if inserted
                then sortComponent t remaining component'
                else sortComponent t (addTail b remaining) component
```

L'algoritmo prende in input i bindings di una componente, i bindings rimanenti e la componente già ordinata. Ora presentiamo l'algoritmo di inserzione di un binding in una componente già ordinata:

```
tryInsert(binding, component):
    match component with
        [] -> [component]
        (b : t) ->
            if b in depsOf(binding)
            then tryInsert binding t
            else if binding in depsOf(b)
            then (True, binding : component)
            else (False, component)
```

Ora abbiamo implementato una versione alternativa dell'insertion sort in cui gli inserimenti vengono effettuati solamente se si è a conoscenza che il binding da inserire fa parte delle dipendenze dirette di un altro binding già inserito nella componente. Inoltre, se non vi sono abbastanza informazioni per inserire un binding in una componente, il suo inserimento viene "rimandato" (viene inserito nei bindings rimanenti) e verrà effettuato in un'iterazione successiva. Il codice si trova nel modulo `Compiler.Types.Prepare.Sort`.

Dopo l'ordinamento dei bindings, è garantito che, ogni qual volta verrà inferito un binding, la premessa della regola **Var** sia vera.

3.11 Type-inference

Fex permette all'utente di indicare o non indicare il tipo di una variabile. Nel caso non venga indicato, il compilatore dovrà inferire il tipo della variabile definita, quindi dovrà fare una "scelta". Per farlo, dovrà prima inferire il tipo dell'espressione legata alla variabile, ad esempio:

```
type Maybe a = Nothing | Just a
let x = Nothing
```

Si può affermare che l'espressione legata alla variabile x abbia tipo $Maybe\ \alpha$. Ora, il compilatore deve scegliere un tipo da sostituire alla variabile di tipo α. Se la scelta fosse arbitraria, ad esempio Int, l'utilizzo di x sarebbe limitato solo a un tipo ovvero $Maybe\ Int$. Un'altra possibile soluzione potrebbe essere associare a x il tipo $Maybe\ a$. Tuttavia, anche questa scelta risulterebbe limitante in quanto, come è stato già esposto nella sezione sul type-system (cfr. Schemi di tipo e polimorfismo parametrico), la variabile di tipo α denoterebbe un tipo fissato non ancora conosciuto. L'introduzione di uno schema di tipo rappresenta una scelta "più generale":

$$x : \forall\alpha.\ Maybe\ \alpha$$

In questo caso, le occorrenze di x nel programma possono avere molteplici tipi.

3.11.1 Damas-Hindley-Milner

Come è stato già menzionato precedentemente, il sistema di tipi di Fex si basa sul type-system di Damas-Hindley-Milner (cfr. [27] [28] [29]). L'algoritmo di inferenza su cui si basa inferisce il tipo più *generale* possibile. Tuttavia, nella letteratura, vengono presentati due algoritmi principali, ma nella pratica, solo uno di essi è implementato dai compilatori dei linguaggi di programmazione. Verranno entrambi menzionati, ma solo le regole di inferenza dell'algoritmo effettivamente implementato verranno esposte:

- *Algoritmo W*, è l'algoritmo effettivamente utilizzato dalle implementazioni, in quanto tiene traccia delle sostituzioni generate dalle eventuali unificazioni (cfr. Unificazione) e le applica all'ambiente di tipizzazione. Proprio a causa della gestione delle sostituzioni, la complessità computazionale dell'algoritmo nel caso peggiore è esponenziale.
- *Algoritmo J*, è un algoritmo più efficiente di W (complessità lineare nella lunghezza dell'espressione), tuttavia, esso non gestisce le sostituzioni, o almeno, esse vengono considerate come side-effects. Viene presentato in letteratura come alternativa efficiente a W.

Come è stato già anticipato, HM viene presentato sotto forma di *regole*, le quali hanno la seguente forma:

$$\frac{Premesse}{Conclusione}\ \textbf{Regola}$$

dove le premesse sono un insieme di *giudizi* e *predicati* (non nel senso di constraints) e la conclusione è un *giudizio*. Un *giudizio* è un'affermazione sul tipo di un simbolo, ad esempio:

$$x : \sigma$$

afferma che il simbolo x ha tipo τ. HM necessita, inoltre, di un modo per "tener traccia" dei giudizi, ovvero un modo per accoppiare un simbolo con un tipo. Introduciamo, quindi, un altro componente: il *contesto* o *ambiente di tipizzazione*. In generale, i giudizi terranno conto del contesto, infatti avranno la seguente forma:

$$\Gamma \vdash_W x : \sigma$$

questa scritta afferma che sotto le ipotesi in Γ, il token x ha tipo σ. Ora verranno elencate le regole di inferenza.

3.11.2 Inferenza di variabile

Prima di presentare la regola d'inferenza delle variabili è necessario introdurre due ulteriori regole. La prima è la regola di specializzazione e introduce una nozione d'ordine parziale tra tipi:

$$\frac{\tau' = \{\alpha_i \mapsto \tau_i\}\tau \quad \beta_i \notin free(\forall\alpha_1 ... \forall\alpha_n.\ \tau)}{\forall\alpha_1 ... \forall\alpha_n.\ \tau \sqsubseteq \forall\beta_1 ... \beta_m.\ \tau'}\ \textbf{Spec}$$

Questa regola sostiene che se esiste una sostituzione $S = \{\alpha_i \mapsto \tau_i\}$ tale che $\tau' = S\tau$, allora la versione "quantificata" dei mono-tipi τ e τ' rispetta la relazione d'ordine parziale $\sqsubseteq$. Nella premessa vi è un'ulteriore condizione che afferma che le variabili quantificate di τ' non devono apparire come variabili libere in $\forall\alpha_1 ... \alpha_n.\ \tau$; questo perché le variabili non legate da un quantificatore (quindi libere) non devono essere sostituite, bensì devono essere trattate come costanti. All'interno del codice, i token tipati che implementano la type-class `SpecType` definita in `Compiler.Ast.Typed` permettono l'applicazione di una sostituzione al loro tipo. La prossima regola rappresenta l'algoritmo di instanziazione:

$$\frac{\Gamma \vdash_W e : \sigma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash_W e : \tau}\ \textbf{Inst}$$

Questa regola è utile per istanziare uno schema di tipo in mono-tipo. È chiaro che è necessario tener conto delle eventuali variabili libere nello schema di tipo σ, ma queste vengono gestite dalla regola di specializzazione. Ora, possiamo esporre la regola di inferenza di una variabile, la quale è stata già introdotta in precedenza, seppur in maniera informale.

$$\frac{x : \sigma \in \Gamma \quad \tau = inst(\sigma)}{\Gamma \vdash_W x : \tau,\ \emptyset}\ \textbf{Var}$$

È doveroso notare che nella conclusione, oltre al giudizio sul tipo della variabile, vi è un altro valore in "output", ovvero le eventuali sostituzioni generate dall'inferenza dei costrutti coinvolti. In questo non vi è alcuna sostituzione.

3.11.3 Inferenza di applicazione

La prossima regola serve per inferire l'applicazione di due espressioni:

$$\frac{\Gamma \vdash_W e_0 : \tau_0,\ S_0 \quad S_0\Gamma \vdash_W e_1 : \tau_1,\ S_1 \quad \tau' = newvar \quad S_2 = mgu(S_1\tau_0, \tau 1 \to \tau')}{\Gamma \vdash_W e_0\ e_1 : S_2\tau',\ S_2S_1S_0}\ \textbf{App}$$

Vi sono due nuovi operatori:

- *newvar*, il quale ritorna una nuova variabile di tipo α tale che $\alpha \notin free(\Gamma)$;
- *mgu*, il quale rappresenta l'algoritmo di unificazione che verrà presentato dettagliatamente nel prossimo paragrafo. Tralasciando, quindi, i dettagli implementativi, tale operatore serve per trovare il tipo più generale. Come risultato, fornisce una sostituzione S_2, la quale viene successivamente applicata al mono-tipo τ' per ottenere il tipo di ritorno del costrutto di applicazione.

Si noti che, a differenza di **Var**, in questa regola vi sono tre sostituzioni in output

- S_0 derivante dall'inferenza della prima espressione;
- S_1 derivante dall'inferenza della seconda espressione;
- S_2 derivante dall'esecuzione dell'algoritmo di unificazione tra tipi;

Nell'inferenza di e_1, l'ambiente di tipizzazione di riferimento è Γ a cui viene applicata la sostituzione S_0. Una possibile ottimizzazione è la seguente:

$$\frac{\Gamma \vdash_W e_0 : \tau_0,\ S_0 \quad S_0\Gamma \vdash_W e_1 : \tau_1,\ S_1 \quad \tau' = newvar \quad S_2 = mgu(S_1\tau_0, \tau 1 \to \tau')}{S_0\Gamma \vdash_W e_0\ e_1 : S_2\tau',\ S_2S_1}\ \textbf{App}$$

La sostituzione S_0 viene, quindi, spostata dall'output direttamente all'ambiente di tipizzazione. Questo per evitare di applicare la sostituzione all'ambiente di tipizzazione due volte: una prima di inferire e_1, una dopo il risultato di output (si presume che le sostituzioni vengano tutte applicate).

3.11.4 Unificazione

L'unificazione è quel processo di risoluzione di equazioni tra espressioni simboliche. In generale, i mono-tipi nel type-system utilizzato in HM (e conseguentemente anche quello utilizzato da Fex) possono essere considerati come termini di un'equazione. Si parla di mono-tipi, poiché l'unificazione non prevede la quantificazione delle variabili all'interno di un'equazione. Un problema di unificazione è un insieme finito di equazioni:

$$\{\tau_1 = \tau_1' \ ... \ \tau_n = \tau_n'\}$$

dove τ_i, τ_i' sono dei mono-tipi. La soluzione di un problema di unificazione è data da una sostituzione $S = \{\alpha_j \mapsto \mu_j\}$, dove α_j è una variabile di tipo e μ_j è un mono-tipo, tale che:

$$\forall i \in \{1, ..., n\}.\ S\tau_i = S\tau_i'$$

Il codice sorgente che si occupa dell'unificazione si trova nel modulo `Compiler.Ast.Typed`, in particolare, la funzione `rawUnify` implementa l'algoritmo di unificazione, osserviamo la sua firma:

```
rawUnify
    :: LangHigherType a
    -> LangHigherType a
    -> IsSpecTest
    -> Either (UnificationError a) (Substitution a)
```

Notiamo subito che vi sono tre argomenti e che il tipo di ritorno è un result-type.

```
data UnificationError a =
      UnmatchTypes (LangHigherType a) (LangHigherType a)
    | TrySwap (LangHigherType a) (LangHigherType a)
    | OccursCheck (LangHigherType a) (LangHigherType a)
    | UnmatchKinds (LangHigherType a) (LangHigherType a)

type IsSpecTest = Bool
```

Il tipo `UnificationError` possiede 4 casi d'errore e tutti prendono in input due mono-tipi:

- `UnmatchTypes`, ritornato quando non vi è alcuna soluzione all'equazione tra due mono-tipi;
- `TrySwap`, simile a `UnmatchTypes` nella semantica, ma indica all'algoritmo di non terminare subito, analizzeremo approfonditamente questo caso in seguito;
- `OccursCheck`, ritornato quando vi è il tentativo di risolvere un problema del tipo:

  $$\alpha = f \ ... \ \alpha \ ...$$

  Questo tipo di equazione porterebbe come risultato un termine infinito visto che α è sotto-termine di se stesso;

- `UnmatchKinds`, ritornato quando i kinds di due mono-tipi non coincidono. In questo caso, non vi può essere alcuna soluzione all'equazione, si pensi, ad esempio, a:

$$M\ \alpha = M\ Int\ \beta$$

Ora passiamo all'algoritmo vero e proprio, il quale sarà presentato mostrando direttamente soltanto una parte di esso all'interno del codice sorgente in `rawUnify`:

```
rawUnify monoTy monoTy' isSpecTest =
    case unification monoTy monoTy' empty of
        Left err -> Left err
        Right vvars -> Right $ elems vvars
    where
        unification lhty lhty' vvars
            | not $ sameInfrdKindOf lhty lhty' =
                Left $ UnmatchKinds lhty lhty'
            | occursCheck lhty lhty' =
                Left $ OccursCheck lhty lhty'
            | tyVarAndConcrete lhty lhty' =
                if isSpecTest
                then Left $ UnmatchTypes lhty lhty'
                else unification lhty' lhty vvars
            | bothConcrete lhty lhty' =
                if headEq lhty lhty'
                then argsUnification lhty lhty' vvars
                else Left $ UnmatchTypes lhty lhty'
            | bothTyVar lhty lhty' && argsOf lhty' > argsOf lhty =
                if isSpecTest
                then Left $ UnmatchTypes lhty lhty'
                else unification lhty' lhty vvars
            | otherwise =
                argsUnification lhty lhty' vvars

        argsUnification =
            if isSpecTest
            then unifyOnArgs
            else twoChancesUnifyArgs

        twoChancesUnifyArgs lhty lhty' vvars =
            case unifyOnArgs lhty lhty' vvars of
                Left err @ (TrySwap _ _) ->
                    case unifyOnArgs lhty' lhty vvars of
                        Left _ -> Left err
                        ok @ (Right _) -> ok
                err @ (Left _) -> err
                ok @ (Right _) -> ok

        <resto del codice>
```

I commenti nel codice sorgente sono stati rimossi. Come è stato fatto notare in precedenza, l'algoritmo prende in input tre argomenti, di cui i primi due sono i mono-tipi sui quali si vuole calcolare l'unificazione. Il terzo argomento è un flag booleano che denota se `rawUnify` deve essere un test di specializzazione (`True`) o semplicemente l'algoritmo di unificazione (`False`). Un test di specializzazione è l'equivalente dell'unificazione, ma può essere fatta in un solo "verso", ovvero se, dati due mono-tipi τ e τ' e data la

sostituzione di ritorno $S = \{\alpha_i \mapsto \tau_i\}$ conseguente alla valutazione dell'unificazione di τ e τ', vale:

$$\forall \alpha_i \in left(S).\ \alpha_i \in \tau'$$

dove con $\alpha_i \in left(S)$ si intendono solamente le variabili di tipo nella parte sinistra di una sostituzione, ovvero quelle che devono essere sostituite. Questo rende l'unificazione più stringente e ciò è utile quando è richiesto che uno dei due mono-tipi non venga modifcato, ad esempio, quando deve essere effettuato il type-check con un tipo che deriva da una annotazione di tipo (type-hinting). Infatti, ci si aspetta che, data un'espressione con type-hinting, l'espressione abbia esattamente il tipo indicato nell'annotazione di tipo.

L'obiettivo dell'algoritmo è riportarsi in una situazione tale che l'equazione tra i due mono-tipi ha la seguente forma:

$$\alpha = \tau$$

dove α è una variabile di tipo e τ è un mono-tipo. Questo perché tale equazione ha la forma di una sostituzione: $\alpha = \tau$. Tuttavia, nell'algoritmo, viene considerata la forma:

$$\tau = \alpha$$

Di fatto, però, la semantica dell'unificazione non cambia; i termini dell'equazione possono essere anche invertiti, cambiando, però, il "verso" dei controlli e delle operazioni nell'algoritmo. Per ottenere un'equazione di quella forma vi sono due operazioni fondamentali:

- *swap*, in cui, se deve essere effettuata l'unificazione di due mono-tipi τ e τ', l'ordine di quest'ultimi viene scambiato. Questa operazione è utile quando il mono-tipo τ' è più "specializzato" rispetto a τ. In generale, il mono-tipo più generale (o meno specializzato) possibile è una variabile singoletto, la quale, come è stato menzionato prima, deve stare nella parte destra dell'equazione.
- *decompose*, in cui, se deve essere effettuata l'unificazione di due mono-tipi τ e τ', viene effettuata l'unificazione tra i vari argomenti dei due mono-tipi, raccogliendo successivamente le varie sostituzioni ottenute. Questa operazione è utile quando un'operazione di *swap* non è vantaggiosa (ovvero il mono-tipo τ è già "più specializzato" di τ'). Essa permette di considerare mono-tipi più piccoli.

Analizzando il codice, la prima chiamata viene fatta a `unification`, la quale considera varie casistiche:

1. ```
   | not $ sameInfrdKindOf lhty lhty' =
       Left $ UnmatchKinds lhty lhty'
   ```

   Nel primo caso viene effettuato un controllo sui kind dei due mono-tipi. Se sono diversi, viene restituito un errore;

2. ```
   | occursCheck lhty lhty' =
       Left $ OccursCheck lhty lhty'
   ```

   Nel secondo caso viene effettuato un altro controllo: il cosiddetto *occurs check*. Se uno dei due mono-tipi ha la forma di una variabile di tipo e quest'ultima compare nell'altro mono-tipo, allora viene ritornato un errore.

3. ```
   | tyVarAndConcrete lhty lhty' =
       if isSpecTest
       then Left $ UnmatchTypes lhty lhty'
       else unification lhty' lhty vvars
   ```
   ```

Nel terzo caso vi è il primo controllo sulla struttura dei mono-tipi: se `lhty` ha come testa una variabile di tipo e `lhty'` ha come testa un tipo concreto, allora vi è un'operazione di *swap*, ovvero vi è una chiamata ricorsiva di `unification` con gli argomenti invertiti. Se l'unificazione è *strict*, viene ritornato un errore, in quanto una variabile di tipo, per definizione, non è più specializzata di un tipo concreto. In questo caso, l'operazione di *swap* non può essere effettuata in quanto verrebbe a meno il vincolo secondo il quale le variabili del primo mono-tipo (`lhty`) non devono essere sostituite.

4. ```
   | bothConcrete lhty lhty' =
       if headEq lhty lhty'
       then argsUnification lhty lhty' vvars
       else Left $ UnmatchTypes lhty lhty'
   ```

   Nel quarto caso viene testato se entrambi i mono-tipi hanno tipi concreti come testa; se così fosse, le teste devono rappresentare lo stesso tipo concreto, altrimenti viene restituito un errore, in quanto non esisterebbe alcuna sostituzione che permetterebbe l'uguaglianza tra i due mono-tipi. Se, invece, le teste sono uguali, allora viene eseguita un'azione di *decompose*, ovvero vengono effettuate delle chiamate ricorsive sugli argomenti dei due mono-tipi. Analizzeremo la funzione `argsUnification` in seguito.

5. ```
   | bothTyVar lhty lhty' && argsOf lhty' > argsOf lhty =
       if isSpecTest
       then Left $ UnmatchTypes lhty lhty'
       else unification lhty' lhty vvars
   ```

   Nel quinto caso viene controllato se entrambi i mono-tipi hanno variabili di tipo come testa e se il numero di argomenti del secondo mono-tipo è strettamente maggiore di quello del primo. In questo caso, vi è un'operazione di *swap* e la conseguente chiamata ricorsiva. Come prima, se l'unificazione è *strict*, l'operazione di *swap* non può essere effettuata e viene restituito un errore.

6. ```
   | otherwise =
       argsUnification lhty lhty' vvars
   ```

   Nell'ultimo caso, vengono valutati tutti i casi restanti: quello che viene eseguito è un'azione di *decompose* sugli argomenti dei due mono-tipi.

La funzione `argsUnification` implementa l'azione di *decompose*. Per farlo, è necessario "accoppiare" gli argomenti dei mono-tipi e questo avviene in `unifyOnArgs` che si occupa di effettuare le eventuali chiamate ricorsive. Senza entrare nei dettagli implementativi di `unifyOnArgs`, una volta che le chiamate ricorsive sono state effettuate, se non vi è stato alcun errore, allora l'output sarà una sequenza di sostituzioni. Tuttavia, le sostituzioni potrebbero non essere consistenti tra loro; si osservi il seguente esempio:

$$\tau = M\ k\ k,\ \tau' = M\ a\ b$$

dove M è un tipo concreto, mentre gli altri termini sono tutti variabili di tipo. Effettuando un'azione di *decompose*, si ottengono le seguenti sostituzioni:

$$S_1 = \{k \mapsto a\},\ S_2 = \{k \mapsto b\}$$

È chiaro che S_1 e S_2 non sono consistenti. In questi casi, `unifyOnArgs`, invece di ritornare l'errore `UnmatchTypes`, ritorna `TrySwap`. La semantica di quest'ultimo indica che è necessario effettuare un'azione di *swap* e riprovare con *decompose*. Continuando con l'esempio precedente, otterremo:

$$S_1 = \{a \mapsto k\},\ S_2 = \{b \mapsto k\}$$
   ```

In questo caso, S_1 e S_2 possono essere unite in un'unica sostituzione:

$$S = \{a \mapsto k,\ b \mapsto k\}$$

In generale, se anche il "secondo tentativo" non va a buon fine, l'algoritmo termina con un errore. La funzione che implementa questa parte è `twoChancesUnifyArgs`. Chiaramente, come si può notare dall'implementazione di `argsUnification`, se l'unificazione è *strict*, allora il "secondo tentativo" non può essere effettuato, in quanto vi è prima un'azione di *swap* che dovrebbe essere eseguita.

3.11.5 Inferenza di lambda astrazione

La seguente regola inferisce il tipo del costrutto di lambda-astrazione:

$$\frac{\tau = newvar \quad \Gamma,\ x : \tau \vdash_W e : \tau',\ S}{\Gamma \vdash_W \lambda x.e : S\tau \to \tau', S}\ \textbf{Abs}$$

Nell'inferenza di questo costrutto, l'argomento x ha come tipo una nuova variabile di tipo, alla quale viene successivamente applicata la sostituzione derivante dall'inferenza dell'espressione e. Dalla regola, si rende evidente la necessità di un tipo che consista nella "mappatura" di un tipo verso un altro tipo.

3.11.6 Inferenza del costrutto "let..in"

Prima di presentare la regola di inferenza, è necessario introdurre la regola di *generalizzazione*:

$$\frac{\Gamma \vdash_W e : \tau \quad \alpha \notin free(\Gamma)}{\Gamma \vdash_W e : \forall\alpha.\ \tau}\ \textbf{Gen}$$

Questa regola quantifica le variabili di tipo che non compaiono già come variabili libere nell'ambiente di tipizzazione. Per il costrutto *let..in*, per indicare la generalizzazione, utilizzeremo la seguente notazione:

$$\overline{\Gamma}(\tau) = \forall\overline{\alpha}.\ \tau \qquad \overline{\alpha} = free(\tau) - free(\Gamma)$$

Ora possiamo presentare la regola di inferenza del costrutto *let..in*:

$$\frac{\Gamma \vdash_W e_0 : \tau,\ S_0 \quad S_0\Gamma, x : \overline{S_0\Gamma}(\tau) \vdash_W e_1 : \tau',\ S_1}{\Gamma \vdash_W let\ x = e_0\ in\ e_1 : \tau',\ S_1S_0}\ \textbf{Let}$$

È doveroso spiegare il motivo dell'esistenza del costrutto *let..in*. Osserviamo il seguente esempio di espressione scritta nel formalismo System-F:

$$(\lambda id.\ (id\ True, id\ 42))(\lambda x.\ x)$$

un tale programma non può essere tipato, in quanto il tipo della variabile *id* non è uno schema di tipo. Si ricordi, infatti, che nel costrutto della lambda-astrazione, il tipo dell'argomento è un mono-tipo. Per questo motivo, è necessario il costrutto *let..in*:

$$let\ id = \lambda x.\ x\ in\ (id\ True, id\ 42)$$

infatti, il programma appena mostrato è *ben tipato*, in quanto il tipo di *id* è uno schema di tipo.

3.11.7 Implementazione delle regole di inferenza

Il codice sorgente che implementa le regole di inferenza (e quindi la type-inference) è situato nel modulo `Compiler.Types.Builder.Type`. Alcuni costrutti di Fex si ispirano a quelli del formalismo *System-F*, tuttavia, molti vengono estesi con sintassi differenti.

Lambda-astrazione in Fex Prendendo come riferimento la definizione del token dell'AST che rappresenta il costrutto di lambda-astrazione:

```
newtype Lambda a = Lambda ([SymbolName a], Expression a, a)
```

si può notare come non vi sia un unico argomento, bensì, una lista. Stessa argomentazione vale per l'applicazione di espressioni:

```
newtype AppExpression a = AppExpr (Expression a, [Expression a], a)
```

Con le dovute accortezze, l'algoritmo di inferenza considera questi costrutti come l'equivalente di costrutti concatenati in *System-F*. Ad esempio:

$$\texttt{lam x y z -> x + y + z} \equiv \lambda x.\ \lambda y.\ \lambda z.\ x + y + z$$

Simboli globali in Fex Un altra nota è necessaria sulle definizioni di simboli in Fex. Infatti, Fex permette la definizione di simboli globali e questo "rompe" la regola di inferenza del costrutto *let..in*. Questo problema viene risolto considerando soltanto parte delle premesse del costrutto *let..in*. Si consideri, ad esempio:

```
let x = e
```

come prima azione, viene inferito il tipo di `e`, sia esso il mono-tipo τ, poi viene effettuata la generalizzazione, quindi x avrà tipo $\forall\overline{\alpha}.\ \tau$. A questo punto, non è necessario compiere altre azioni e il binding $x : \forall\overline{\alpha}.\ \tau$ può essere aggiunto nell'ambiente di tipizzazione.

Costrutto "let..in" in Fex Il costrutto *let..in* in Fex ha nella sintassi un'altra estensione rispetto al suo omonimo in *System-F*, ovvero i bindings accettano anche argomenti:

```
let f x y = (x, y)
```

In questo caso, la politica dell'algoritmo di inferenza è trattare gli argomenti `x` e `y` come argomenti di un costrutto di lambda-astrazione (cfr. Lambda-astrazione), quindi, si può pensare a questa equivalenza in termini di type-inference con *System-F*:

$$\texttt{let f x y = (x, y)} \equiv \textit{let}\ f = \lambda x.\ \lambda y.\ (x, y)$$

3.12 Estensioni dell'algoritmo di inferenza

Fex possiede numerosi costrutti che arricchiscono la sintassi di base di *System-F*, inoltre, introduce concetti nuovi come i predicati e i tipi qualificati. Di conseguenza, l'algoritmo di inferenza di tipo di *System-F* deve essere esteso.

3.12.1 Pattern matching

Fex possiede due costrutti diversi per il pattern matching (cfr. Caratteristiche del linguaggio). Il primo costrutto permette di "ispezionare" il valore di una singola espressione, per esempio:

```
type Maybe a = Nothing | Just a

val f : List a -> String
let f l =
    match first l with
          Nothing -> "Error_404"
        | Just _ -> "Ok_200"
```

Il secondo costrutto, invece, non agisce su un'espressione, ma sugli argomenti di altri costrutti, per esempio:

```
val f : Maybe a -> b -> String
let f
    | Nothing _ -> "Error_404"
    | (Just _) _ -> "Ok_200"
```

In qualsiasi caso, vi sono sempre uno o più valori da ispezionare e uno o più casi, i quali sono costituiti da una o più *case expressions* (o *pattern*) a sinistra della freccia (nel codice, spesso vengono nominate come *matching expressions*) e un'espressione di ritorno a destra della freccia. Quindi abbiamo un costrutto della forma:

$$match\ val_1\ ...\ val_n\ with\ case_1\ ...\ case_m$$

dove val_i è un token che definiamo come *token top-level* o *token di scrutinio* e $case_j$, che chiameremo semplicemente *caso*, ha la forma:

$$me_{1j}\ ...\ me_{nj} \rightarrow e_j$$

Prima di presentare l'algoritmo di inferenza, è necessario introdurre la nozione di tipo di un caso. Il costrutto del caso si divide in due parti: i pattern e l'espressione a destra della freccia. Dato che i pattern sono legati ai token di scrutinio, si ha che il costrutto del caso ha due componenti che devono essere tipate indipendentemente l'una dall'altra:

$$me_{1j} : ty_{1j}\ ...\ me_{nj} : ty_{nj} \rightarrow e_j : tyExpr_j$$

L'algoritmo di inferenza di tale costrutto è il seguente:

- inferisci val_i, applica le sostituzioni all'ambiente di tipizzazione e poi fai lo stesso con val_{i+1}, finché non termina la lista di token di scrutinio;
- nell'inferenza dei casi, dovrà valere:

  $$\forall j.\ typeOf(val_i) \sqsubseteq typeOf(me_{ij})$$

  ovvero le case expressions non potranno avere un tipo più generale dei corrispondenti token di scrutinio;
- nell'inferenza dei casi, è necessario tener traccia dei tipi dei casi già inferiti in precedenza per due motivi:
  1. per quanto riguarda i pattern, deve valere la condizione al punto (2);
  2. per quanto riguarda le espressioni e_j, non sempre nel caso j-esimo si riesce ad inferire il tipo più specializzato tra tutte le espressioni e_j, $\forall j$;
- inferisci $case_j$, applica le sostituzioni all'ambiente di tipizzazione e ai casi precedenti, poi passa a $case_{j+1}$:
  - nell'inferenza del caso j-esimo, il quale avrà la forma $me_{1j}\ ...\ me_{nj} \rightarrow e_j$, verranno inferite, una alla volta, le case expressions me_{ij};
  - inferisci come se fosse un argomento di una lambda astrazione me_{ij} la quale avrà mono-tipo τ (esso avrà la forma di una variabile di tipo singoletto), trova il tipo dell'i-esimo valore di scrutinio, sia esso τ', poi effettua l'unificazione tra τ e τ'. Applica le sostituzioni all'ambiente di tipizzazione e ai casi precedenti, poi continua con me_{i+1j} finché non termina la lista di case-expressions;
  - inferisci e_j, la quale avrà mono-tipo τ, trova il tipo delle espressioni dei casi (basta prendere il tipo dell'espressione del caso precedente), sia esso τ', effettua l'unificazione tra τ e τ' e applica le sostituzioni all'ambiente di tipizzazione e ai casi precedenti, in modo che le espressioni dei casi abbiano tutte lo stesso tipo.

Eliminazione degli scrutini multipli Come è stato già menzionato precedentemente, Fex espone due sintassi diverse per effettuare il pattern matching, una delle quali può possedere un numero arbitrario di case-expressions. Tuttavia, il token che rappresenta un *caso* del costrutto di pattern matching può avere solamente una *matching expression* (come è stato menzionato in precedenza, è una notazione equivalente a una case-expression), la sua definizione si trova in `Compiler.Ast.Typed`:

```
data NotedCase a = NotedCase (NotedMatchExpr a) (NotedExpr a) a
```

Questo rende necessaria una fase di desugaring che avviene subito dopo l'inferenza di un costrutto di pattern matching, quindi durante la fase di type-inference. Si consideri il seguente codice Fex:

```
type Maybe a = Nothing | Just a
type Either a b = Left a | Right b

let f
    | Nothing _ -> 42
    | (Just x) (Left y) -> x + y
    | _ (Right y) -> y
```

Dopo aver inferito il tipo dell'espressione legata ad `f`, vengono creati gli argomenti di `f`, i quali sono legati alle case expressions nell'espressione di pattern matching, siano essi `v1` e `v2`. Dopodiché, vengono utilizzati i costruttori delle tuple per creare le case expressions nel costrutto di pattern matching:

```
let f v1 v2 =
    match (v1, v2) with
          (Nothing, _) -> 42
        | (Just x, Left y) -> x + y
        | (_, Right y) -> y
```

Questo metodo ha una limitazione, ovvero il numero di pattern non può superare il numero di argomenti massimo di un costruttore di una tupla di Haskell, quindi è dipendente da esso. Di solito il costruttore di tuple "più grande" in Haskell ha 62 argomenti, quindi questa limitazione viene accettata, in quanto difficilmente l'utente definisce funzioni con più di 62 argomenti, tuttavia, non è escluso che in futuro questa implementazione venga migliorata. È bene menzionare che vi è questa dipendenza da parte di Fex, in quanto le tuple in Fex sono implementate semplicemente come tuple in Haskell. Il codice sorgente che implementa questa funzionalità si trova in `Compiler.Desugar.BreadthPM`.

3.12.2 Constraints

La sintassi Fex permette di esprimere espressioni il cui tipo possiede dei predicati; ciò richiede che l'algoritmo classico di type-inference venga esteso. L'algoritmo esteso di Fex si ispira a [2], in cui Martin Sulzmann, Martin Odersky, Martin Wehr presentano un'estensione di HM che supporta anche tipi che possiedono predicati. I giudizi di tipo vengono estesi con un'ipotesi aggiuntiva sui constraints e avranno la forma:

$$CS, \Gamma \vdash_W e : \tau$$

Perciò l'ambiente di tipizzazione viene esteso con un altro "attore", ovvero un contesto di soddisfabilità, il quale sarà utile a capire quali constraints possono essere accettati e quali vengono rifiutati. Nel caso di Fex, il contesto di soddisfabilità è dato dall'insieme delle istanze. L'articolo [2] presenta inizialmente il problema di come ridefinire le regole istanziazione (**Inst**) e generalizzazione (**Gen**). Sull'istanziazione di uno schema di tipo si arriva subito alla seguente conclusione:

$$\frac{CS, \Gamma \vdash_W e : \forall \overline{\alpha}.D \Rightarrow \tau' \quad CS \vdash^e \{\overline{\tau}/\overline{\alpha}\}D}{CS, \Gamma \vdash_W e : \{\overline{\tau}/\overline{\alpha}\}\tau'} \textbf{ Inst}$$

Questa regola sostiene che, quando si istanzia uno schema di tipo $\forall\overline{\alpha}.D \Rightarrow \tau$, gli unici mono-tipi validi sono quelli della forma $\{\overline{\tau}/\overline{\alpha}\}\tau'$ e tali che il constraint $\{\overline{\tau}/\overline{\alpha}\}D$ *è soddisfatto* da uno o più constraints nel contesto di soddisfabilità CS. Di seguito è riportato in pseudo-codice l'algoritmo di soddisfabilità dei constraints, definito in `Compiler.Ast.Typed`. Esso si basa sull'unificazione *strict*:

```
isSatisfied(C => args, C' => args'):
    C == C' and
    areEq(args, args')
```

dove `areEq` è l'algoritmo che effettua il test di specializzazione tra i vari tipi di `args` e `args'`, eseguendo, successivamente, il test di uguaglianza tra mono-tipi:

```
areEq(args, args'):
    match moreSpecOneByOne(args, args', []) with
        failure -> false
        args'' -> args == args1''

moreSpecOneByOne(args, args', resArgs):
    match args, args' with
        [] [] -> resArgs
        [] _ -> fail
        _ [] -> fail
        (ty : t) (ty' : t') ->
            let subst = isMoreSpec(ty, ty') in
            let t' = apply(subst, t') in
            let resArgs = resArgs ++ [apply(subst, ty')] in
                moreSpecOneByOne(t, t', resArgs)
```

la funzione `apply` prende in input una sostituzione e uno o più token che vengono ritornati dopo che è stata applicata loro la sostituzione. La funzione `isMoreSpec` è invece il test di specializzazione (unificazione *strict*) tra due mono-tipi; infatti, come da definizione, gli argomenti dei constraints sono mono-tipi. Si ricordi che il test di unificazione può fallire. In generale, diremo che un constraint C *è soddisfatto* da un constraint C' se vale $isSatisfied(C, C')$; viceversa, diremo che un constraint C *soddisfa* un constraint C' se vale $isSatisfied(C', C)$. Inoltre, dato il contesto di soddisfabilità CS e un constraint C, diremo che C *è soddisfatto* se esiste $C' \in CS$ tale che C è soddisfatto da C'. Si noti come, a causa dell'utilizzo dell'unificazione *strict*, il test soddisfabilità tra constraints non sia commutativo. Si osservi il seguente esempio in Fex:

```
property Foo a b c =
    <metodi>
;;
```

Supponiamo che esistano le seguenti istanze di `Foo`:

```
Foo Char a b
Foo Int Int Int
Foo a String Int
```

Il constraint `Foo String String Int` è soddisfatto in quanto esiste l'istanza `Foo a String Int` che lo soddisfa. La non-commutatività si nota subito, infatti, `Foo a String Int` non sarebbe soddisfatto da textttFoo String String Int, in quanto la variabile di tipo `a` non è più specializzata (o meno generale) rispetto al tipo concreto `String`. Il constraint `Foo x Char y`, invece, non è soddisfatto in quanto nessuna delle tre istanze lo soddisfa.

Ora che è stata presentata formalmente la nozione di soddisfabilità di un constraint, possiamo procedere con la ridefinizione della regola di generalizzazione. Nell'articolo [2] viene proposta la seguente

regola:

$$\frac{CS \wedge D, \Gamma \vdash_W e : \tau \quad \overline{\alpha} \notin free(\Gamma) \cup free(CS)}{CS \wedge \exists \overline{\alpha}.D, \Gamma \vdash_W e : \forall \overline{\alpha}.D \Rightarrow \tau} \textbf{ Gen}$$

Si noti come vi è un nuovo quantificatore: $\exists \overline{\alpha}.D$. Questa notazione è utile per quantificare le variabili di tipo all'interno di un constraint.

Per quanto riguarda l'algoritmo utilizzato in Fex, vi è un ulteriore contesto/ambiente di cui è necessario tener conto, ovvero quello dei problemi di constraints correnti. Questa ulteriore estensione dell'ambiente di tipizzazione è utile per mantenere i constraints che sono presenti nell'attuale contesto di inferenza. In seguito, vedremo in che modo viene utilizzato. Lo indicheremo con Ω.

Indipendentemente dalle regole **Inst** e **Gen** ridefinite, è necessario fissare un comportamento che riguarda le altre regole di inferenza. In particolare, è necessario decidere il momento in cui viene eseguito l'algoritmo di soddisfabilità dei constraints presenti in Ω. Questa azione è detta anche *risoluzione dei constraints*. In [2], vi è la proposta di nuove regole di inferenza in cui la risoluzione viene effettuata alla fine delle premesse di ogni regola di inferenza, eccetto quella della lambda-astrazione, in cui la risoluzione non viene effettuata affatto. Tuttavia, sempre nell'articolo, vi è un'ulteriore proposta nella quale la risoluzione viene effettuata in maniera *lazy*, ovvero appena dopo la generalizzazione ed è questa la politica utilizzata da Fex. Rimangono alcune questioni aperte:

1. quand'è che un predicato viene aggiunto a Ω?
2. quali sono i constraints (se esistono) da risolvere dopo che è stata effettuata la generalizzazione?

Per rispondere a entrambe le domande, mostriamo come avviene l'inferenza di tipo in Fex in presenza di constraints. Osserviamo il seguente esempio di codice Fex:

```
property Foo a b c =
    val foo : a -> b -> c -> Char
;;

let f x y z = foo x y z
```

Andando a inferire l'espressione legata ad `f`, un token che viene incontrato è `foo`. Il suo schema di tipo associato è $\forall a, b, c.\ Foo\ a\ b\ c \Rightarrow a \to b \to c \to Char$. A questo punto, si può applicare la regola **Var**, che, a sua volta, applica la regola **Inst**. Si ricordi che la politica di Fex sulla risoluzione dei constraints è *lazy*, perciò il constraint $Foo\ a\ b\ c$ non viene risolto immediatamente, bensì viene aggiunto all'ambiente dei problemi di constraints. Ω si rende quindi necessario proprio a questo fine, ovvero ritardare la risoluzione dei constraints. È bene notare che esso non compare nelle regole di inferenza; questo è possibile in quanto la politica *lazy* può essere considerata come *side-effect* che non cambia la semantica della valutazione dei constraints. Quindi la conclusione della regola **Var** in questo caso sarà semplicemente:

$$CS, \Gamma \vdash_W foo : a1 \to a2 \to a3 \to Char$$

dove $a1$, $a2$ e $a3$ sono variabili nuove. Continuando ad applicare le regole di inferenza **App** e **Var** si arriverà ad avere, prima di applicare **Let** per generalizzare `f`, il seguente ambiente di tipizzazione:

$$\Gamma = \{x : a1, y : a2, z : a3\}$$

$$\Omega = \{Foo\ a1\ a2\ a3\}$$

$$CS = \emptyset$$

Applicando la regola **Let**, si applica anche la regola **Gen**; in questo caso, l'unico constraint nel contesto è $Foo\ a1\ a2\ a3$ e il mono-tipo da generalizzare associato a `f` è $\tau = a1 \to a2 \to a3 \to Char$. Rimane da decidere quando un constraint deve essere aggiunto allo schema e quando, eventualmente, deve essere

risolto. Innanzitutto, è necessario osservare le variabili libere del mono-tipo τ, siano esse $\overline{\alpha} = free(\tau)$. Un constraint C viene aggiunto allo schema di tipo se:

$$free(C) \subseteq \overline{\alpha} \wedge free(C) \neq \emptyset$$

La prima condizione richiede che tutte le variabili libere nel constraint siano presenti anche nel mono-tipo, mentre la seconda condizione previene l'occorrenza di constraints senza variabili di tipo. Nel caso in cui la condizione non venga rispettata, il constraint C deve essere risolto. Tornando all'esempio precedente, si ha che il constraint $Foo\ a1\ a2\ a3$ può essere aggiunto allo schema di tipo che verrà generato. `f` avrà, quindi, il seguente tipo:

$$\forall a1, a2, a3.\ Foo\ a1\ a2\ a3 \Rightarrow a1 \rightarrow a2 \rightarrow a3 \rightarrow Char$$

Si osservi ora quest'altro esempio:

```
property Bar a b =
    val bar : a -> b -> Char
;;

let z = z

let g x y = (foo x y z, bar x y)
```

Nell'applicazione delle regole di inferenza, si ottiene che l'espressione legata a `g` ha tipo `(Char, Char)` e prima di applicare la regola **Let**, è naturale pensare all'ambiente dei problemi di constraints siffatto:

$$\Omega = \{Foo\ a1\ a2\ b, Bar\ a1\ a2\}$$

e al mono-tipo da generalizzare:

$$\tau = a1 \rightarrow a2 \rightarrow Char$$

Al momento della generalizzazione, si ha che:

$$free(Foo\ a1\ a2\ b) \not\subseteq free(\tau) \wedge free(Bar\ a1\ a2) \subseteq free(\tau)$$

Di conseguenza, il constraint $Bar\ a1\ a2$ può essere aggiunto allo schema di tipo, mentre $Foo\ a1\ a2\ b$ deve essere risolto in quanto la variabile di tipo b non compare nel mono-tipo τ.

Risoluzione dei constraints Per quanto riguarda la risoluzione dei constraints, essa corrisponde alla seguente operazione:

```
solve(C, instances):
    any (fun C' -> isSatisfied(C, C')) in instances
```

Tuttavia, è necessario perfezionare l'implementazione, in quanto, al momento, non è possibile selezionare l'istanza che soddisfa il constraint (cfr. Dispatch statico). L'istanza che soddisfa il constraint, infatti, deve essere unica, in quanto è necessario non avere ambiguità sull'implementazione da scegliere. Chiaramente, se non vi sono istanze che soddisfano un constraint, il programma viene respinto con un errore. Bisogna, quindi, fissare una semantica di inferenza di istanza nel caso in cui vi siano due o più istanze che soddisfano un constraint. La scelta di Fex è quella di selezionare l'istanza "più specializzata". Definiamo ora una relazione d'ordine parziale tra constraints, siano C e C' due constraints:

- se $isSatisfied(C, C') \wedge isSatisfied(C', C)$, allora $C = C'$;
- se $isSatisfied(C, C') \wedge \neg(isSatisfied(C', C))$, allora $C > C'$;

- se $\neg(isSatisfied(C, C')) \wedge isSatisfied(C', C)$, allora $C < C'$;
- se $\neg(isSatisfied(C, C')) \wedge \neg(isSatisfied(C', C))$, allora la relazione d'ordine è indefinita.

dove l'operatore $\neg$ è la naturale negazione booleana. Di seguito, vi è definito l'algoritmo di selezione dell'istanza ispirato da [13]:

1. l'input è una sequenza cs di constraints;
2. scarta da cs tutti quei constraints c tali che esiste $c' \in cs$ che non sia c e tale che $c' > c$;
3. se alla fine vi rimane più di un constraint, allora non è possibile selezionare un'istanza; se, invece, vi è una sola istanza rimasta, allora essa viene selezionata.

Il fatto che la relazione d'ordine sia parziale non rappresenta un problema, infatti, le istanze vengono scartate solo se la relazione d'ordine è definita.

Confronto con Haskell Fex ha un sistema di tipi molto simile a quello di Haskell, il quale, a sua volta, possiede un constraint-system "costruito" sopra HM. Anche Haskell ha una politica *lazy* di risoluzione delle istanze, ma, a differenza di Fex, in caso di molteplici istanze che soddisfano un constraint, l'azione di default è quella di rifiutare il programma, a meno che l'utente non utilizzi alcune estensioni del linguaggio (cfr. [13]). La politica di Fex è più elastica e permette all'utente di avere più istanze valide per certi constraints, tuttavia, ciò rappresenta una responsabilità in più da parte dell'utente, il quale deve essere consapevole di come avviene l'inferenza di istanza. Vi è quindi un tradeoff tra elasticità e consapevolezza dell'utente.

3.12.3 Type-hinting

Fex permette all'utente di indicare il tipo di un'espressione (o di un simbolo). Questo comporta una modifica all'algoritmo di inferenza di tipo. Data un'espressione con type-hinting, lo schema generale è inferire prima l'espressione e successivamente "applicare" il type-hinting. Per quanto riguarda i casi ricorsivi delle espressioni (tutti i casi fuorché i casi di base: variabili, letterali e data-constructors), il tipo proveniente dal type-hinting viene "scomposto" in più sotto-tipi che vengono utilizzati come type-hinting nell'inferenza dei casi ricorsivi. Tale scomposizione è definita sui tipi funzione ed è chiamata, all'interno del codice sorgente, operazione di *unfolding*:

```
unfoldType(ty):
    match ty with
        (ty1 -> ty2) -> ty1 : unfoldType ty2
        _ -> [ty]
```

Questa operazione è molto utile, ad esempio, nell'inferenza della lambda astrazione. Si osservi il seguente sorgente Fex:

```
(lam x -> f x) : Char -> String
```

Il tipo `Char -> String` viene scomposto in `Char` e `String`. Nell'inferenza dell'argomento `x`, il type-hinting sarà esattamente `Char`, mentre per l'inferenza dell'espressione `f x` il type-hinting sarà `String`.

Per quanto riguarda, invece, i casi di base, lo schema è:

- cerca nell'ambiente di tipizzazione il tipo del token, sia esso lo schema di tipo σ;
- applica la regola di istanziazione su σ, sia τ il mono-tipo risultante;
- sia π il mono-tipo istanziato dallo schema di tipo proveniente dal type-hinting;

- effettua l'unificazione *strict* tra π e τ; si ricordi che tale operazione, a differenza dell'unificazione originale, non gode della proprietà commutativa, in quanto l'azione di *swap* non è ammessa. Il mono-tipo π non può essere modificato dopo la sostituzione;
- applica la sostituzione all'ambiente di tipizzazione e al token tipato.

La regola è molto simile a quella senza type-hinting, tuttavia, viene effettuata un'unificazione in più. Dato che l'insieme dei letterali e dei data-constructors all'interno del linguaggio non può avere variabili libere, un'ottimizzazione che viene applicata ai loro casi è non applicare la sostituzione all'ambiente di tipizzazione. Un'operazione che, in generale, è piuttosto dispendiosa.

3.12.4 Ricorsione

Il formalismo System-F non ammette la ricorsione, inoltre, non è nemmeno possibile trovare il tipo per il cosiddetto *combinatore Y*, un modo per definire la ricorsione in linguaggi che generalmente non la supportano. Questo rende *System-F* non Turing-completo, in quanto qualsiasi programma termina. Fex (come anche Haskell e molti altri linguaggi funzionali) ammette, invece, una sintassi per la ricorsione. Questo fatto porta con sé alcune conseguenze molto importanti. Innanzitutto, la premessa della regola **Var** non è vera per le funzioni ricorsive, in quanto con la presenza della ricorsione i simboli possono apparire in un programma prima ancora che il loro tipo venga generalizzato. Inoltre, sarebbe impossibile ordinare i bindings in base alle loro dipendenze. Per questo motivo, i bindings ricorsivi vengono raggruppati (cfr. Clusters di definizioni mutualmente ricorsive) e considerati come un unico binding della forma:

$$rec\ v_1 = e_1\ and\ v_2 = e_2\ and\ ...\ v_n = e_n$$

Tuttavia, il problema della regola **Var** non è ancora risolto, quindi è necessario definire una nuova regola che riguarda i bindings ricorsivi:

$$\frac{\Gamma, \Gamma' \vdash_W e_1 : \tau_1 \quad ... \quad \Gamma, \Gamma' \vdash_W e_n : \tau_n \quad \Gamma, \Gamma'' \vdash_W e : \tau}{\Gamma \vdash_W rec\ v_1 = e_1\ and\ v_n = e_n\ in\ e : \tau}\ \textbf{Rec}$$

dove:

$$\Gamma' = v_1 : \tau_1, ..., v_n : \tau_n$$

$$\Gamma'' = v_1 : \overline{\Gamma}(\tau_1), ..., v_n : \overline{\Gamma}(\tau_n)$$

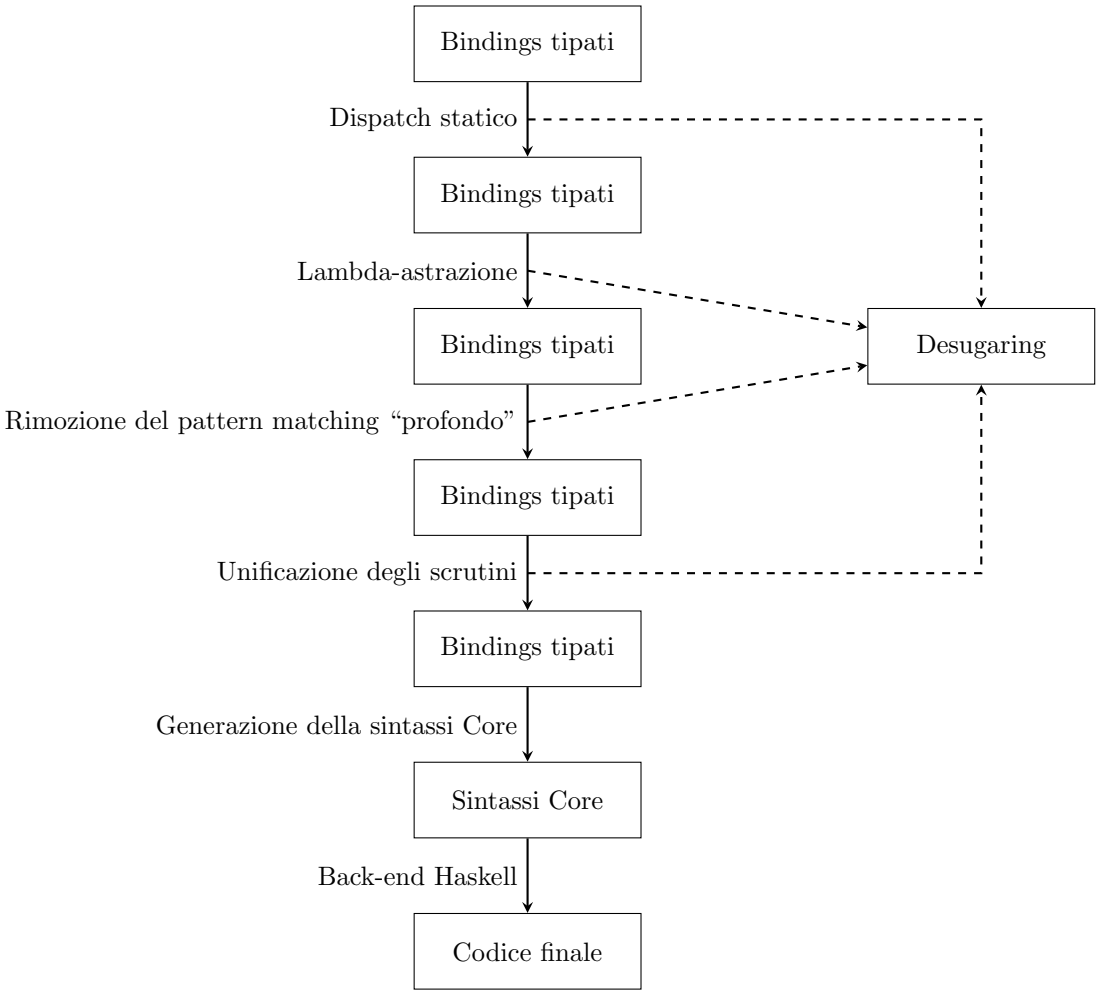
La notazione $\overline{\Gamma}(\tau)$ è stata definita nella sezione Inferenza del costrutto "let..in" e indica la generalizzazione del mono-tipo τ, tenendo conto delle variabili libere presenti in Γ. La regola inferisce prima tutte le espressioni e_i e poi applica la regola di generalizzazione **Gen** soltanto alla fine. Nell'algoritmo di inferenza di Fex, prima di inferire un cluster di bindings ricorsivi, vengono creati nuovi mono-tipi della forma di singole variabili di tipo, i quali vengono assegnati ai bindings del cluster. Durante l'inferenza, questi mono-tipi possono essere specializzati. In questo modo, si può applicare la regola **Var** ad ogni occorrenza di simbolo ricorsivo. L'istanziazione di un mono-tipo non rappresenta un problema in quanto le variabili libere non vengono istanziate. In questo caso, l'algoritmo di istanziazione corrisponde alla funzione identità.

4 Generazione del codice

L'output dell'algoritmo di type inference è un `TypedProgram`, una tabella che mantiene i bindings tipati. La generazione del codice consiste in tre sotto-fasi:

1. alcuni task che riguardano il desugaring, necessari per la generazione del codice;
2. traduzione dai token tipati di `Compiler.Ast.Typed` ai token della sintassi del linguaggio Core;
3. esecuzione del back-end di Haskell, il quale genera il codice a basso livello di target differenti.

Di seguito, lo schema delle fasi finali del compilatore:



4.1 Fasi di desugaring

Prima della vera e propria generazione del codice, sono necessarie alcune fasi di desugaring. Esse vengono eseguite dopo la fase di type-inference poiché, se il programma non può essere tipato, i messaggi d'errore risultano più comprensibili per l'utente.

4.1.1 Dispatch statico

Il polimorfismo ad-hoc viene implementato attraverso il meccanismo delle type-classes. La sintassi Core non possiede, però, costrutti per i metodi delle type-classes e per i metodi delle istanze. La politica del compilatore Fex è quella di aggiungere i metodi delle istanze ai bindings del programma, cambiando i nomi degli stessi metodi di istanze diverse (cfr. Costruzione delle istanze). Questa politica ha importanti implicazioni: innanzitutto, nelle espressioni del programma, i metodi delle istanze possono essere chiamati, ma essi possono essere nominati solamente tramite gli identificatori dei metodi delle proprietà. Ad esempio:

```
property Foo a =
    val foo : a -> a
;;

instance Foo String =
    let foo s = s
;;

let f x = (x, foo "42")
```

Come è stato già esposto in precedenza, il metodo `foo` dell'istanza `Foo Int` avrà un identificatore diverso da `foo`. Tuttavia, `foo` compare nell'espressione legata a `f`. È necessario, quindi, un modo per trasformare, eventualmente, le occorrenze dei metodi di proprietà con i giusti metodi di istanza. Si pensi a quest'altro pezzo di codice Fex:

```
type Box a = Box a

val g : Foo (Box a) => a -> Box a
let g x = foo (Box x)

let h x = g x
```

È chiaro come la chiamata di `foo` all'interno dell'espressione legata a `g` dipenda da quale istanza della forma `Foo (Box a)` venga utilizzata. Vale lo stesso per `g` nei confronti dell'espressione legata ad `h`. In linea generale, il suddetto problema è legato al polimorfismo ad-hoc e viene risolto attraverso un metodo di dispatch statico. Il polimorfismo ad-hoc permette di creare funzioni che hanno comportamenti diversi a seconda del loro tipo. Il dispatch statico si occupa di "scegliere" correttamente le funzioni a disposizione nel programma (i metodi d'istanza) che implementano i relativi metodi di proprietà nel programma. A differenza di molti linguaggi object-oriented, in cui la selezione dei metodi avviene a run-time attraverso meccanismi di dispatch dinamici, in Fex la selezione viene risolta completamente a compile-time, evitando così l'overhead che può essere causato da un algortimo di dispatch dinamico. In particolare, il dispatch statico in Fex viene implementato attraverso la *monomorfizzazione* [11], già introdotta, in realtà, nella costruzione delle istanze. Ora renderemo la monomorfizzazione generale a tutte le funzioni del programma; l'implementazione che seguirà si ispira a un articolo sul blog di Jeremy Mikkola [12]. Si ripensi alla precedente funzione `g`. Il suo schema di tipo ha con sè dei predicati. In questo caso, durante la fase di type-inference, alla funzione `g` verrà aggiunto un parametro che corrisponderà al predicato `Foo (Box a)`, ovvero stiamo trattando il tipo di `g` come se fosse:

```
val g : Foo (Box a) -> a -> Box a
```

È necessario estendere la definizione di tipo qualificato:

$$QT := MT \mid C \mid C \Rightarrow QT$$

Quindi, un tipo qualificato in Fex può possedere soltanto constraints. Questa eventualità viene tuttavia limitata a pochi casi possibili e, comunque, Fex non espone una sintassi per esprimere tipi qualificati che possiedono soltanto constraints.

Sempre durante la fase di type-inference, ogni volta che la funzione `g` viene utilizzata nel programma, oltre alla normale applicazione della regola di inferenza **Var** e alla generazione del token tipato che rappresenta una variabile, verrà generata e applicata a `g` un'ulteriore variabile. Quest'ultima viene associata al relativo problema di constraint e viene salvata nell'ambiente di tipizzazione Ω. Quindi, gli elementi in Ω hanno, in realtà, la seguente forma:

$$\langle C, v \rangle$$

dove C è un constraint e v è l'identificatore di una variabile, detta anche *variabile di dispatch*. Al momento della generalizzazione, dai problemi di constraint non risolti vengono estratte le variabili associate, le quali diventano parte degli argomenti del binding. Ad esempio, l'implementazione della funzione `h` a livello di token tipati sarà:

```
let h v1 x = g v1 x
```

dove `v1` è una variabile di dispatch. Le espressioni che contengono le variabili di dispatch sono definite in `Compiler.Ast.Typed`:

```
data DispatchTok a =
      DispatchVar (NotedVar a) a
    | DispatchVal (OnlyConstraintScheme a) a
```

Il primo caso (`DispatchVar`) rappresenta una variabile di dispatch. Il secondo caso, invece, rappresenta un *valore di dispatch*, il quale è costituito da uno schema di tipo contenente solamente constraints ed è utile per "selezionare" l'implementazione giusta di una funzione.

La fase di type-inference, oltre a un programma tipato, ritorna anche una sequenza di nomi di simboli da ridefinire. Questi ultimi sono tutti quei simboli che compaiono nel programma nella cui espressione vi sono valori di constraints. Se in un'espressione un simbolo ha solo valori di dispatch applicati (e quindi non variabili di dispatch), allora è necessario generare un nuovo simbolo il cui identificatore univoco viene creato a partire dai valori di dispatch, come è avvenuto per i metodi di istanza (cfr. Costruzione delle istanze). Nel modulo `Compiler.Desugar.AdHoc` viene implementato un algoritmo di monomorfizzazione che, ricorsivamente, cerca nelle espressioni tutte le occorrenze di simboli a cui sono applicati valori di constraints e genera le conseguenti implementazioni. Ovviamente, la generazione di nuove implementazioni può dar vita a nuove espressioni con valori di constraints, quindi l'algoritmo agisce ricorsivamente finché non esistono più implementazioni da generare:

1. l'input è una serie di bindings bs in cui cercare i valori di dispatch;
2. $\forall b \in bs$, vengono cercati, nell'espressione legata a b, tutte le occorrenze di simboli che hanno applicati solo valori di dispatch;
3. ogni occorrenza viene sostituita con un nuovo identificatore univoco in tutto il programma costituito da:
   $$\langle symid \rangle \langle dispatchval_1 \rangle ... \langle dispatchval_n \rangle$$
   dove $\langle symid \rangle$ è l'identificatore originale e $\langle dispatchval_i \rangle$ è l'i-esimo valore di dispatch applicato;
4. $\forall sym$ che viene sostituito, sym viene aggiunto alla lista rs di simboli da creare, con le informazioni sui valori di dispatch che servono per creare il nuovo identificatore;
5. $\forall r \in rs$, viene creato un nuovo binding in cui:

- l'identificatore è costruito utilizzando il metodo menzionato precedentemente;
- gli argomenti di dispatch spariscono;
- nell'espressione associata, vengono cercate tutte le occorrenze delle variabili di dispatch, le quali vengono sostituite dai valori di dispatch associati a r;
- applica i passi (3) e (4) per ogni occorrenza di simbolo che ha soltanto valori di dispatch applicati;

6. ripeti (5) finché non terminano i simboli in rs.

4.1.2 Lambda-astrazione

Riportiamo la definizione di binding:

```
type BindingSingleton a = (NotedVar a, [NotedVar a], NotedExpr a)
```

Come si può notare dalla definizione, un binding è una tripla costituita da un identificatore, degli argomenti e un'espressione. In questa fase, vengono modificate tutte le espressioni in modo da eliminare gli argomenti dalla tripla. Di preciso, gli argomenti del binding diventano argomenti di un costrutto di lambda-astrazione che ha come espressione l'espressione legata del binding. Si ricordi che, durante la type inference, gli argomenti di un binding vengono trattati come argomenti di un costrutto di lambda-astrazione (cfr. Lambda-astrazione in Fex).

4.1.3 Rimozione del pattern matching profondo

Si consideri il seguente sorgente Fex:

```
type Success = Ok200
type Error = Undefined | Err401 | Err404 | Err500
type ErrType = ServerError | ClientError

type Maybe a = Nothing | Just a
type Either a b = Left a | Right b

let f x =
    match x with
          Nothing -> Just ServerError
        | Just (Right Ok200) -> Nothing
        | Just (Left Err401) -> Just ClientError
        | Just (Left Err404) -> Just ClientError
        | Just _ -> Just ServerError
```

Il codice relativo alla funzione `f` è semanticamente equivalente a:

```
let f x =
    match x with
          Nothing -> Just ServerError
        | Just x1 ->
            (match x1 with
                  Right x2 ->
                    (match x2 with
                        Ok200 -> Nothing
                    )
                | Left x2 ->
                    (match x2 with
                          Err401 -> Just ClientError
                        | Err404 -> Just ClientError
                        | _ -> Just ServerError
                    )
            )
```

La differenza tra le due versioni è che nella seconda i data-constructors appaiono solo come teste (e non come argomenti) nelle case expressions. In seguito, vedremo perché è importante che valga questa proprietà (cfr. Generazione del codice Core). Questa fase di desugaring trasforma le espressioni di pattern matching, modificando le case expressions, in modo che ad ogni loro argomento che contiene un data-contructor, viene creata una nuova expressione di pattern matching innestata. Il codice relativo a questa fase si trova in `Compiler.Desugar.DeepPM`.

4.1.4 Unificazione degli scrutini

Questa è un'altra fase di desugaring che riguarda le espressioni di pattern matching. Le motivazioni di questa fase verranno chiarite in seguito (cfr. Generazione del codice Core). Prima di entrare nei dettagli di questa fase, è necessario definire cosa sono le *variabili di scrutinio* o più semplicemente *scrutini*. Sia *match e with* $m_1 \to e_1$ ... $m_n \to e_n$ un'espressione di pattern matching, chiameremo *scrutinio* la variabile legata direttamente al valore dell'espressione e. Ad esempio:

```
let f x y =
    match g x y with
          Just v -> Just (v + y)
        | a -> Just y
        | b -> b
```

Nell'espressione legata ad `f`, le variabili di scrutinio sono `a` e `b`, mentre la variabile `v` non lo è. Ciò che avviene nella fase di desugaring è la trasformazione delle case-expressions in modo che una sola variabile di scrutinio venga utilizzata, quindi il codice Fex precedentemente mostrato diventerà:

```
let f x y =
    match g x y with
          Just v -> Just (v + y)
        | v1 -> Just y
        | v1 -> v1
```

Si noti che le variabili delle sotto-espressioni vengono opportunamente aggiornate.

4.2 Generazione del codice Core

Core è un linguaggio minimale utilizzato da GHC come rappresentazione intermedia di Haskell. La generazione del codice Core è la fase finale del front-end del compilatore Fex. Core è una variante esplicitamente tipata (a differenza di Haskell e Fex) del formalismo System-F [3]. L'obiettivo del compilatore Fex è costruire dei token tipati e manipolarli in modo che la traduzione dal programma tipato Fex all'equivalente Core sia più lineare possibile. GHC espone al cliente delle API che permettono di utilizzare le funzionalità del compilatore [14], comprese quelle per creare e manipolare un programma Core. Di seguito vi è il tipo di dato algebrico utilizzato nelle API di GHC per rappresentare un'espressione Core:

```
data Expr b
  = Var   Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Tick  (Tickish Id) (Expr b)
  | Type  Type
  | Coercion Coercion

type Arg b = Expr b

type Alt b = (AltCon, [b], Expr b)

data AltCon
  = DataAlt DataCon
  | LitAlt  Literal
  | DEFAULT

data Bind b = NonRec b (Expr b)
            | Rec [(b, (Expr b))]
```

I commenti sono stati rimossi, cfr. [15] per il codice sorgente originale. Il tipo `Expr` rappresenta un'espressione Core, mentre l'alias di tipo `Alt` rappresenta un *caso* del costrutto di pattern matching. Il tipo `Bind`, invece, rappresenta i bindings di Core. L'obiettivo finale è costruire un token `CoreProgram`:

```
type CoreProgram = [CoreBind]

type CoreBind = Bind CoreBndr

type CoreBndr = Var
```

Il tipo `Var` è rappresenta le variabili in Core con un informazione sul loro tipo. Nel contesto di Core, con il *variabile* si intende un concetto di ampio spettro, infatti, `Var` comprende le variabili di un programma, le variabili di tipo, le variabili di kind, etc.. Come si può notare dalle strutture dati utilizzate nelle API di GHC e dalle fasi precedenti del compilatore Core, l'obiettivo del compilatore Fex è creare token tipati che possano essere successivamente trasformati in token Core nel modo più lineare possibile. Alcuni esempi sono:

- la definizione di binding di Core è molto simile a quella utilizzata dal compilatore Fex (cfr. Approccio a tabelle);
- i costrutti di pattern matching, dopo le varie fasi di desugaring, non possiedono costruttori innestati nelle case expressions, come si può notare dal costruttore di un *caso* in Core:

```
type Alt b = (AltCon, [b], Expr b)
```

dove la variabile di tipo `b` rappresenterà una variabile Core (`CoreBndr`);

- il costrutto di pattern matching di Core richiede come secondo parametro una variabile di scrutinio:

  ```
  | Case  (Expr b) b Type [Alt b]
  ```

  Il fatto che gli scrutini siano stati tutti unificati (cfr. Unificazione degli scrutini) rappresenta un vantaggio in termini di implementazione;

- i bindings non hanno la nozione di argomenti, infatti, gli argomenti dei bindings di Fex sono stati trasformati in argomenti dei costrutti di lambda-astrazione (cfr. Lambda-astrazione);

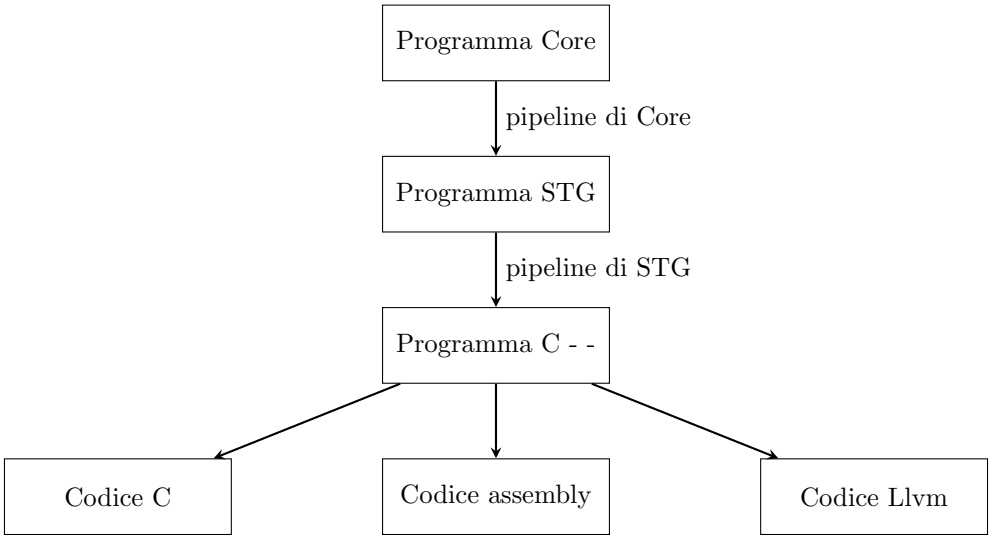
- Le definizioni dei token tipati in Fex portano con sè l'informazione sul proprio tipo in modo che i token Core possano essere costruiti. Si ricordi che Core è un linguaggio esplicitamente tipato, quindi ogni costrutto che ha la nozione di tipo, ha il tipo esplicitato nella sintassi.

I dettagli implementativi dell'effettiva traduzione si trovano nel modulo `Compiler.Codegen.ToCore`.

Traduzione dei tipi e dei costruttori Il modulo `Compiler.Codegen.Type` si occupa di trasformare i token di Fex che rappresentano i tipi nei corrispettivi token di Core [16]. Il tipo di dato algebrico nelle API di GHC per rappresentare un tipo in Core è `Type`. A differenza di Fex in cui per ogni nozione diversa di tipo è stato creato un token diverso, `Type` racchiude molte nozioni di tipo, tra cui: mono-tipo, schema di tipo, variabili di tipo, etc.. Dal punto di vista delle API di GHC, `Type` è un tipo di dato astratto, infatti, sempre il modulo `Type` espone una serie di funzioni per costruire i tipi in Core. Altri tipi di dati fondamentali in Core sono `TyCon` e `DataCon`, definiti rispettivamente in [17] e [18], i quali rappresentano rispettivamente i type-constructors e i data-constructors. In precedenza, quando è stato presentato il tipo di approccio (a tabelle) che utilizza il compilatore Fex per costruire e salvare i token tipati, è stato fatto un confronto con GHC (cfr. Confronto con GHC) che utilizza un approccio molto diverso. Infatti, in GHC i token vengono salvati in una struttura a grafo. Questo metodo si ripercuote soprattutto a livello programmatico, infatti, prendendo in considerazione `TyCon` e `DataCon`, vi è una mutua dipendenza tra i due token. Questo implica che per costruire un costruttore di tipo è necessario avere a disposizione i data-constructors associati, mentre per costruire un data-constructor è necessario avere a disposizione il costruttore di tipo associato. La conseguenza è la scrittura di codice mutualmente ricorsivo, tuttavia, si ricordi che Haskell è un linguaggio con un modello di valutazione delle espressioni *lazy* e questo aiuta a prevenire la ricorsione infinita in codice mutualmente ricorsivo.

4.3 Back-end

La fase finale di compilazione di un programma Fex consiste nel compilare un programma Core in target differenti. La pagina di documentazione ufficiale di GHC espone nei dettagli il ciclo di compilazione di un programma Haskell (e di conseguenza di un programma Core) [19]. Lo schema semplificato è il seguente:



STG è un'ulteriore rappresentazione intermedia di Haskell, ma, a differenza di Core, non è un linguaggio tipato. Uno degli obiettivi più importanti di STG è rendere efficiente la valutazione delle espressioni sull'hardware standard. C - - è un linguaggio C-like, sviluppato con l'obiettivo di essere generato dai compilatori di linguaggi ad alto livello. Nella traduzione da un formalismo a un altro, vi sono vari passaggi (pipeline) in cui vengono effettuate, oltre alla traduzione effettiva, alcune ottimizzazioni.

I tre target principali sono:

- Codice C, che viene restituito in output come *pretty-printing* dal codice C - -;
- Codice Assembly specifico di una macchina;
- Codice Llvm, portabile e ottimizzato per molte architetture.

5 Sviluppi futuri

Fex è un linguaggio sperimentale in evoluzione. Vi sono alcune *features* che arricchiscono il linguaggio che potrebbero essere implementate in futuro. Di seguito, sono elencate alcune caratteristiche ritenute interessanti da aggiungere al linguaggio. Nella presentazione delle funzionalità aggiuntive, verranno analizzate prima le basi teoretiche, discutendo i vantaggi (e, eventualmente, gli svantaggi) che vengono portati al linguaggio e i problemi che vengono risolti, dopodiché, in ogni sezione, verranno discusse una o più possibili implementazioni.

5.1 Tipi lineari

Un sistema di tipi *lineari* è una caratteristica dei linguaggi di programmazione (specialmente funzionali) molto interessante, la quale porta numerosi vantaggi all'utente in termini pratici, tuttavia, non è una *feature* largamente adottata nei linguaggi mainstream. Vi è stata una proposta di implementazione in GHC [20] che ha successivamente dato vita a un'estensione di Haskell. Questa sezione si ispirerà alla proposta che è stata fatta per GHC: verranno date alcune definizioni al fine di introdurre la nozione di *tipo lineare* e verrà, a sua volta, proposta una bozza di implementazione nel compilatore Fex. Due vantaggi pratici che offre un sistema di tipi lineari sono: l'utilizzo e il controllo della mutabilità con interfacce *pure*; maggiore controllo in computazioni che riguardano l'IO. In generale, i tipi lineari offrono maggiori garanzie all'utente programmatore.

Innanzitutto, prima di dare qualsiasi definizione, è bene precisare come nell'articolo [20] l'approccio utilizzato non sia dividere i tipi in due macro-insiemi (lineari e non-lineari), bensì la *linearità* viene associata al tipo funzione. Informalmente, una funzione è *lineare* se consuma il suo argomento una e una sola volta. Indicheremo con $\triangleright$ il tipo funzione lineare. Il motivo per il quale la linearità viene definita sul tipo funzione e non sulla totalità dei tipi è che questo permette di avere retrocompatibilità su un type-system già esistente. Per quanto riguarda la nozione di *valutazione* (o *consumazione*) di un valore, è bene introdurre la politica che viene utilizzata da Haskell: la *valutazione lazy*. Questo tipo di valutazione delle espressioni permette di valutare le espressioni non nel momento in cui vengono legate alle variabili, bensì quando c'è l'effettivo bisogno del loro risultato per effettuare altre computazioni. Questa politica permette di ritardare il più possibile la valutazione delle espressioni. Precisiamo cosa si intende con *consumare un valore esattamente una volta*, diamo questa definizione proposta anche nell'articolo di riferimento:

- per consumare un valore di tipo atomico (ovvero di un tipo con costruttore di tipo senza argomenti, ad esempio il tipo `Char`) esattamente una volta, lo si valuti;
- per consumare un valore di tipo funzione esattamente una volta, si applichi a esso un argomento e si consumi il risultato esattamente una volta;
- per consumare un valore di un tipo di dato algebrico esattamente una volta, si esegua il pattern matching su di esso e si consumi tutte le sue componenti lineari esattamente una volta.

Con questa definizione possiamo rispondere alla seguente domanda sul tipo dei data-constructors: quale tipo dovremmo assegnare a un data-constructor? Prendiamo come esempio il costruttore delle coppie `(,)`:

- $(,) : \forall \alpha, \beta.\ \alpha \triangleright \beta \triangleright (\alpha, \beta)$
- $(,) : \forall \alpha, \beta.\ \alpha \to \beta \to (\alpha, \beta)$

La prima scelta è quella corretta, infatti, se il risultato dell'espressione `(,) x y` è consumato esattamente una volta, allora, per definizione, `x` e `y` vengono consumati esattamente una volta.

Questo sistema di tipi può portare, tuttavia, ad alcuni problemi. Si pensi, come viene mostrato in un esempio in [20], alla seguente implementazione della funzione `map` di Haskell:

```
map _ [] = []
map f (x : t) = f x : map f t
```

La funzione `map` può avere i seguenti due tipi:

- $map : \forall \alpha, \beta.\ (\alpha \triangleright \beta) \to [\alpha] \triangleright [\beta]$
- $map : \forall \alpha, \beta.\ (\alpha \to \beta) \to [\alpha] \to [\beta]$

I due tipi sono incompatibili tra loro. Introduciamo, quindi, un livello di polimorfismo sulla *molteplicità* dei tipi funzione:

- $map : \forall \alpha, \beta, p.\ (\alpha \to_p \beta) \to [\alpha] \to_p [\beta]$

Con la notazione appena mostrata, stiamo asserendo che `map` accetta come argomenti una funzione con molteplicità illimitata (ovvero non-lineare) e una lista con molteplicità p, inoltre, la funzione in input accetta un argomento di molteplicità p. Definiamo formalmente la molteplicità:

$$\pi := 1 \mid \omega \mid p \mid \pi + \pi \mid \pi * \pi$$

dove ω è la molteplicità illimitata e p è una variabile di molteplicità. Inoltre, valgono i seguenti assiomi:

- $+$ e $*$ sono associativi e commutativi;
- $*$ ha precedenza maggiore di $+$;
- $\forall p.\ p * 1 = 1 * p = p$;
- $\omega * \omega = \omega$;
- $1 + 1 = 1 + \omega = \omega + \omega = \omega$.

Consideriamo $\to \equiv \to_\omega$ e $\triangleright \equiv \to_1$. Si noti come valga la seguente affermazione:

$$\forall \pi, \mu.\ (vars(\pi) = \emptyset \land vars(\mu) = \emptyset) \implies \pi + \mu = \omega$$

dove $vars$ è la funzione che calcola l'insieme delle variabili che compaiono in una molteplicità. Tuttavia, se aggiungiamo l'elemento neutro di $+$, sia esso 0, allora la precedente affermazione non vale più. Dato questo sistema di tipi lineari, è ora necessario aggiornare l'algoritmo di type inference in modo che inferisca anche le molteplicità dei vari tipi funzione. Per questo, in [20] viene definito un lambda-calcolo le cui regole d'inferenza inferiscono la molteplicità corretta dei tipi funzione. Non specificheremo di seguito le regole d'inferenza, tuttavia, è bene fare alcune precisazioni. Innanzitutto, nelle regole di inferenza, si può pensare ai tipi nell'ambiente di tipizzazione come l'input di un giudizio e alle molteplicità come l'output. Esistono due regole diverse per quanto riguarda le variabili e i data-constructors e questo è dato dalla diversa definizione di linearità per le funzioni e per i data-constructors. Per quanto riguarda, invece, il costrutto di lambda-astrazione, visto che deve essere creato un tipo funzione e a priori non si conosce la molteplicità di questo tipo, a esso viene associata una variabile di molteplicità, la quale non deve esistere già come variabile libera. Per la regola di applicazione, è necessario guardare la molteplicità della freccia (nel tipo dell'espressione che applica) e a seconda della sua molteplcità, le variabili libere nell'argomento dell'applicazione dovranno essere utilizzate con la stessa molteplicità.

5.1.1 Bozza di implementazione in Fex

Ora che è stato presentato un sistema di tipi lineari, si può delineare una bozza di implementazione in Fex. Il sistema di tipi lineari da aggiungere a quello già esistente di Fex si ispira a quello appena presentato, riutilizzando le varie nozioni incontrate. Un primo aggiornamento che può essere effettuato è alla nozione di mono-tipo, in particolare al tipo funzione, il quale ora si presenterà:

$$MT \rightarrow_{\pi} MT$$

Dove π è la nozione di molteplicità fornita in precedenza (anch'essa da aggiungere al sistema di tipi). Dopodiché, sarà necessario estendere l'algoritmo di type inference, in particolare:

- è necessario aggiungere un "attore" all'ambiente di tipizzazione, sia esso Π, il quale associa ai token una molteplicità. Anche in questo ambiente sarà necessario tener conto delle variabili libere, in quanto la nozione di molteplicità stessa può possedere variabili di molteplicità;
- estendere le regole di inferenza in modo che, oltre alla normale type inference, eseguano anche l'inferenza di molteplicità. Si presume che, come per i tipi, in Fex le molteplicità possano o non possano essere esplicitate;
- per quanto riguarda le regole di inferenza, al momento della generalizzazione, sarà necessario generalizzare anche le eventuali variabili di molteplicità che non compaiono libere in Π.

Dopodiché, le occorrenze del tipo funzione, per qualunque molteplicità, possono essere tradotte in Core come il classico tipo funzione senza la nozione di molteplicità.

5.2 Effetti

Gli *effetti algebrici* rappresentano un modo per ottenere più controllo su comportamenti impuri da parte dei programmi. Essi si basano sul fatto che tali comportamenti impuri derivano da un insieme di *operazioni* come, per esempio, `read` e `write` per la lettura e la scrittura su file, `raise` per le eccezioni o `fork` e `wait` per la concorrenza. L'idea alla base è avere un sistema di tipi che comprende una nuovo insieme di tipi, detti *tipi effetti*, in grado di inglobare l'insieme di *operazioni* che portano *side-effects* (di qualsiasi genere). Linguaggi come Koka [21] implementano un sistema di tipi che supporta anche gli *effetti*. In questa sezione, verranno discusse le principali caratteristiche di un sistema di *effetti*, seguendo l'articolo [22], poi verrà proposta un'implementazione nel compilatore Fex, analizzando i cambiamenti da portare e gli eventuali ostacoli.

Nell'articolo [22] viene, innazitutto, fissato un linguaggio. L'approccio utilizzato qui è lo stesso, tuttavia, il linguaggio che verrà definito sarà leggermente diverso da quello nell'articolo:

$$v := e \mid fun\ x \rightarrow c \mid h$$

$$h := handler\{return\ x \rightarrow c,\ op_1(x;k) \rightarrow c_1,\ ...,\ op_n(x;k) \rightarrow c_n\}$$

$$c := return\ v \mid op(v; y.c) \mid do\ x \leftarrow c_1\ in\ c_2 \mid v1\ v2 \mid with\ v\ handle\ c$$

dove e sono i costrutti di System-F, estesi anche con valori costanti. v rappresenta i *valori*, h rappresenta gli handler e c le computazioni. È necessaria una discussione di alcuni costrutti appena definiti:

- il costrutto $return\ v$, dove v è un valore, rappresenta la computazione pura ed è utile ogni qual volta viene utilizzato un valore in un contesto in cui ci si aspetta una computazione;
- la chiamata di un'*operazione* $op(v; y.c)$ passa un parametro v all'operazione op, il risultato viene legato a y e, infine, viene valutata la computazione c, chiamata anche *continuazione*. Un'operazione rappresenta un'esecuzione con side-effect;

- il costrutto *do* $x \leftarrow c_1$ *in* c_2 è chiamato *sequenziazione*. Viene valutata prima la computazione c_1, poi il risultato viene legato alla variabile x e, infine, si valuta c_2. Questo costrutto non è da confondersi con la chiamata di un'operazione, in quanto, sebbene simili, la sequenziazione è utile per mettere in sequenza qualsiasi computazione;
- un *handler* definisce delle azioni da eseguire al momento della chiamata di certe operazioni (specificate dall'handler stesso). L'handling della computazione pura è opzionale;
- il costrutto *with* v *handle* c valuta la computazione c utilizzando l'handler nel valore v per gestire le computazioni in c;

Nel formalismo appena presentato, vi è una divisione tra computazioni pure e impure. A questo punto, si può definire un sistema di tipi. Per farlo, estenderemo il sistema di tipi di System-F:

$$VT := PT \mid PT \rightsquigarrow CT \mid CT \Rrightarrow CT$$

$$CT := VT!\{op_1,\ ...,\ op_n\}$$

È doveroso fare alcune precisazioni riguardo a queste definizioni. Innanzitutto, VT è il tipo dei valori, mentre CT è il tipo delle computazioni. Con PT si intende uno schema di tipo, è indifferente se il suo caso base sia un tipo qualificato (cfr. Schemi di tipo e polimorfismo parametrico) oppure un monotipo. Il caso $CT \Rrightarrow CT$ rappresenta il tipo degli handler, invece, il caso $PT \rightsquigarrow CT$ rappresenta il tipo funzione che ha come tipo di "ritorno" un tipo di computazione. Le regole di inferenza per il formalismo in questione sono abbastanza intuitive e non verranno riportate (esse sono comunque definite in [22]). I tipi delle computazioni racchiudono le possibili operazioni che fanno parte dell'effetto. Questo fatto permette di catturare in maniera precisa i side-effects prodotti dalle operazioni.

5.2.1 Bozza di implementazione in Fex

Un sistema di effetti offre un meccanismo generale per gestire le computazioni impure. Fex è un linguaggio funzionale puro e, al momento, non ha un meccanismo per gestire tutto ciò che non riguarda il "mondo puro". Un sistema di effetti potrebbe essere un modo per aggiungere computazioni impure, come input/output, eccezioni, multi-threading, mantenendo separato il "mondo puro" da quello "impuro".

Per aggiungere un sistema di effetti a Fex, è necessario, innanzitutto, aggiungere una nuova sintassi per tutti quei costrutti necessari all'implementazione degli effetti, in particolare bisogna aggiungere:

- una sintassi per definire gli effetti, ovvero i tipi delle computazioni. Questo diventa opzionale se alcuni effetti vengono considerati come built-in, tuttavia, ciò limiterebbe molto le possibilità del programmatore che non sarebbe in grado di estendere l'insieme degli effetti;
- una sintassi per definire le operazioni, ma questo dovrebbe essere una conseguenza della possibilità di definire effetti;
- una sintassi per definire gli handler, infatti, senza di essi, non sarebbe possibile gestire i side-effects;
- una sintassi che permette di utilizzare gli handler. Nel formalismo presentato, un costrutto del genere è rappresentato da *with* v *handle* c;

Dopodiché, è necessario definire un sistema di effetti. Un possibile approccio è quello di Koka [23], il quale, potendo inferire tutti i side-effects in una computazione, connota il tipo ritorno con gli effetti opportuni. In particolare, l'assenza di side-effects è connotata con *total*, l'effetto per le funzioni matematiche. In ogni caso, si rende necessaria un'estensione del sistema di tipi di Fex, infatti, dati i nuovi

costrutti del linguaggio, bisogna quanto meno avere un tipo per le computazioni e un tipo per gli handler. L'aggiornamento delle regole di inferenza avviene di conseguenza, tuttavia, per quanto riguarda queste ultime, vale la pena fare alcune precisazioni. Per tipizzare una computazione nel modo giusto, è necessario eseguire un'analisi statica della computazione e individuare tutte le operazioni definite in un effetto. Inoltre, è possibile eseguire l'analisi statica degli handler e decidere se un handler gestisce tutte le operazioni utilizzate in una computazione. Tuttavia, se, come in Koka, vi sono effetti che comprendono, ad esempio, anche la divergenza di un programma, non esiste un'operazione specifica associata all'effetto di divergenza. In questo caso, è necessario un'analisi statica specifica, infatti, l'unico "costrutto" con cui si ottiene la divergenza in Fex è la ricorsione, quindi, ogni qual volta vi sono uno o più bindings ricorsivi, il loro tipo di ritorno può essere l'effetto divergenza. Il fatto che un qualche simbolo f abbia come tipo l'effetto divergenza non implica che la valutazione di f provochi l'effettiva divergenza del programma. Questo perché il problema della fermata non è decidibile.

Una volta che sono stati generati gli eventuali token tipati, si rende necessario estendere la generazione del codice. Anche Haskell possiede un meccanismo per dividere il mondo puro da quello impuro: viene utilizzata la monade `IO`, la quale ingloba un token che rappresenta lo stato del mondo impuro. A differenza di un sistema ad effetti come quello presentato precedentemente, questo meccanismo utilizzato da Haskell è meno fine, in quanto tratta la (quasi) totalità delle computazioni impure con la monade `IO`. Questo implica un potenziale ostacolo: la "compilazione" dei side-effects potrebbe dipendere da come viene "compilata" la monade `IO`. In ogni caso, è necessario un *mapping* dagli effetti definiti nel linguaggio ai "token" di basso livello del compilatore. A questo proposito, esiste un modulo delle API di GHC (`PrimOp`) che espone tutte quelle operazioni che non possono essere definite in un sorgente Haskell.

5.3 Varianti polimorfe

Le *varianti polimorfe* rappresentano un meccanismo di estendibilità dei tipi, in particolare esse risolvono il task di, dato un tipo, estenderlo nuovi costruttori. Questo è un problema ricorrente nella programmazione e viene risolto anche dai linguaggi orientati agli oggetti tramite meccanismi di sotto-tipaggio ed ereditarietà, tuttavia, le varianti polimorfe costituiscono un modo più conciso ed elegante di estendere i tipi. Esse sono state implementate in OCaml, ma non sono diffuse in molti altri linguaggi di programmazione. Esse rompono il vincolo informale secondo il quale un data-constructor appartiene a uno e un solo type-constructor; si ricordi che tale vincolo è valido in Fex e Haskell. In questa sezione, verranno brevemente presentati i principali vantaggi delle varianti polimorfe, dopodiché, non verrà presentata un'implementazione in Fex, bensì verranno discusse alcune proposte di implementazioni provenienti da altre fonti.

Come mostra Garrigue in [24], uno dei vantaggi principali delle varianti polimorfe è quello già citato precedentemente: dopo aver fissato un tipo di dato, estendere le API dei costruttori del tipo di dato. Si osservi il seguente pseudo-codice:

```
type HttpCode = Ok200 | Err404 | Err500
```

dove `HttpCode` è un tipo e `Ok200`, `Err404` e `Err500` sono data-constructors associati solamente al tipo `HttpCode`. In seguito diremo che, dato un data-constructor dc e un type-constructor tc, $dc \in tc$, se dc è associato a tc e a nessun'altro costruttore di tipo. Poniamo che un utente stia sviluppando il codice per un server http e, inizialmente, rappresenti i codici di risposta http con il tipo `HttpCode`. Un metodo per evitare di riscrivere il codice relativo al server ogni volta che è necessario un nuovo codice di risposta è definire a priori `HttpCode` con tutti i possibili codici di risposta http. È chiaro che è una soluzione parecchio onerosa per l'utente e molti dei codici potrebbero rimanere inutilizzati. Inoltre, alcune funzioni implementate nel codice del server potrebbero gestire insiemi molto ristretti di codici http, ad esempio:

```
let handleReq json =
    let res = process(json);
    lookupAndSend res

val lookupAndSend : (HttpCode, String) -> HttpAction ()
let lookupAndSend res =
    match res with
          (Ok200, msg) -> send(msg)
        | (Err404, path) -> send("path " ++ path ++ " not found")
        | (Err500, _) -> send("Excuse me, my fault")

let server =
    listen(fun req -> handleReq(req))
```

Sarebbe veramente tedioso per l'utente fare pattern matching nella funzione `lookupAndSend` su tutti i codici http esistenti. Utlizzando le varianti polimorfe, possiamo isolare i casi da gestire per `lookupAndSend`:

```
val lookupAndSend : ({< `Ok200, `Err404, `Err500}, String) -> HttpAction ()
let lookupAndSend res =
    match res with
          (`Ok200, msg) -> send(msg)
        | (`Err404, path) -> send("path " ++ path ++ " not found")
        | (`Err500, _) -> send("Excuse me, my fault")
```

Utilizziamo la sintassi {< `Ok200, `Err404, `Err500} per indicare che il tipo è polimorfo (anche in assenza di variabili di tipo) e che può essere ristretto in seguito. Supponiamo di avere poi il seguente pezzo di codice:

```
val possibleRes : List {> `Ok200, `Err404, `Err500}
let possibleRes = [`Ok200, `Err404, `Err500]
```

Anche in questo caso, il tipo {> `Ok200, `Err404, `Err500} è polimorfo, ma, a differenza di prima, questo tipo può essere esteso con altri varianti polimorfe. Successivamente, l'utente potrebbe aggiungere il seguente codice:

```
val defaultRes : List {> `Err400, `Tmp307, `Ok200, `Err404, `Err500}
let defaultRes = [`Err400, `Tmp307] ++ possibleRes
```

Al di là della sottile differenza tra < e > che riguarda la tipizzazione delle varianti polimorfe, questo è un esempio di come possono essere utilizzate le varianti polimorfe per estendere un tipo. Un altro effetto benefico portato dall'uso delle varianti polimorfe è la condivisione di tipi di dato tra librerie inizialmente separate. Infatti, senza le varianti polimorfe, sarebbe necessario costruire l'astrazione per "far comunicare" le librerie nel modo giusto, utilizzando un attore di terze parti che funga da interfaccia comune tra le librerie coinvolte. Invece, con le varianti polimorfe, è possibile definire lo stesso tipo nelle librerie coinvolte, permettendo alle librerie di essere indipendenti tra loro.

5.3.1 Implementazioni

In letteratura, vi sono numerose proposte di implementazione delle varianti polimorfe. Di seguito, ne elencheremo alcune, analizzando eventuali vantaggi e svantaggi:

- come propone inizialmente Garrigue [24], una possibile compilazione delle varianti polimorfe può essere effettuata facendo un mapping da varianti a valori numerici interi. Questa soluzione è semplice da implementare e piuttosto efficiente, tuttavia, presenta un grosso svantaggio: è necessario conoscere tutte le varianti polimorfe di un programma. Il problema nasce se il linguaggio

permette la compilazione separata di sorgenti, infatti, a quel punto non è possibile conoscere contemporaneamente tutte le varianti polimorfe in un programma;

- sempre Garrigue [24] propone un'implementazione altrettanto efficiente e che risolve il problema della compilazione separata: un mapping dai nomi delle varianti ai loro valori hash. Sebbene questo metodo sia semplice, efficiente e permetta la compilazione separata, presenta anch'esso uno svantaggio: cosa succede se due varianti polimorfe diverse hanno lo stesso valore hash? Questo evento è piuttosto raro (se si utilizza una "buona" funzione hash) e proprio per questo risulta un ottimo compromesso tra i vantaggi che porta e i casi "sfortunati", tuttavia, è necessario comunque gestire questa eventualità. Garrigue [24] propone di emettere un errore a tempo di compilazione, il quale è sempre una scelta migliore di un errore a run-time. È chiaro che l'utente dovrà cambiare manualmente l'implementazione dei sorgenti. Questo diventa particolarmente sconveniente quando i sorgenti in questione non sono stati scritti dall'utente stesso, ma sono esterni;

- Kagawa [25] tratta l'implementazione delle varianti polimorfe in Haskell e propone una soluzione di più "alto livello": un mapping dalle varianti polimorfe alle type-classes di Haskell. Questa soluzione, come vedremo, è piuttosto "elegante" in quanto non è necessario estendere il linguaggio, tuttavia, presenta qualche ostacolo. Presentiamo, prima di tutto, la sintassi scelta da Kagawa [25]:

```
variant cs => a in VariantName ts where
    Constr_1 :: ty_11 -> ... -> ty_1n -> a
    ...
    Constr_m :: ty_m1 -> ... -> ty_mn -> a
```

Le varianti polimorfe sono rappresentate dai simboli dopo la keyword `where`, i quali hanno degli identificatori che iniziano con una lettera maiuscola. Si può notare come la dichiarazione di una variante sia molto simile a quella di una type-class. Tuttavia, vi è una restrizione, ovvero che la variabile di tipo `a` deve apparire come tipo di ritorno di ogni variante polimorfa. Il contesto `cs` specifica le superclassi, che, a loro volta, dovranno essere delle varianti. `ts` è una sequenza di argomenti (variabili di tipo) come per le type-classes. Il costrutto per definire una variante subisce la seguente trasformazione:
prima:

```
variant cs => a in VariantName ts
```

dopo:

```
class cs => VariantName a ts | a -> ts
```

Come si nota dal codice, sono state utilizzate due estensioni (le quali hanno buon supporto da parte dei compilatori Haskell): `MultiParamTypeClasses` e `FunctionalDependencies`. Kagawa [25] estende, successivamente, la sintassi Haskell con altri due tipi di dichiarazioni (i record e le istanze di varianti) e spiega come sia necessario un aggiornamento dell'algoritmo di inferenza di tipo. In ogni caso, questo metodo di implementazione ha il vantaggio di essere direttamente traducibile (escludendo la modifica all'algoritmo di inferenza di tipo) in codice Haskell, quindi, risulta essere solamente zucchero sintattico.

6 Conclusione

Il paradigma funzionale permette, in generale, la scrittura di codice componibile e dichiarativo e ciò comporta una maggiore manutenibilità ed espandibilità dei software (talvolta anche a discapito delle prestazioni). Un sistema di tipi come HM offre parecchie garanzie e questo previene comportamenti non voluti da parte dei programmi, compresi errori a run-time. Linguaggi di programmazione *debolmente* tipati quali Javascript o C forniscono poche garanzie in termini di sistema di tipi. Questo porta ad avere un minor controllo sui programmi e da ciò conseguono una serie di problematiche che spesso riguardano anche la sicurezza. Altri linguaggi, come Python o Java, che vengono considerati *fortemente* tipati, non prevengono comunque una vasta gamma di errori che riguardano i tipi e che vengono identificati soltanto a run-time, avendo come effetto negativo che tali errori, se non gestiti, portano a un crash dei programmi. Fex si propone come linguaggio funzionale puro, inteso come in grado di isolare le computazioni deterministiche. Fex si propone altresì di gestire il "mondo impuro" tramite il sistema di tipi con una granularità più fine rispetto a quella di Haskell (si ricordi che, per gestire i comportamenti "impuri", Haskell utilizza, in generale, la monade `IO`), arricchendo il sistema di tipi stesso. Quest'ultimo obiettivo non è stato raggiunto, in quanto il sistema di tipi risulta, al momento, meno espressivo rispetto a quello di Haskell.

Bibliografia

[1] Amr Sabry - What is a purely functional language? - in: Journal of Functional Programming, Volume 8, Issue 1, January 1998, pp. 1 - 22

[2] Martin Sulzmann, Martin Odersky, Martin Wehr - Type Inference with Constrained Types - in: Theory and Practice of Object Systems · January 1999

[3] Stephen Diehl - Dive into GHC: Targeting Core - url: `https://www.stephendiehl.com/posts/ghc_03.html` - ultima visita: 21/02/2023

[4] libreria Parsec - url: `https://hackage.haskell.org/package/parsec` - ultima visita: 21/02/2023

[5] Data types for Haskell entities - url: `https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/entity-types` - ultima visita: 21/02/2023

[6] Edward Y. Zang - Backpack without symbol tables - in: May 11, 2016 - url: `http://web.mit.edu/~ezyang/Public/backpack-symbol-tables.pdf` - ultima visita: 21/02/2023

[7] Haskell Flexible Instances extension - url: `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/instances.html#extension-FlexibleInstances` - ultima visita: 21/02/2023

[8] Haskell instance resolution termination conditions - url: `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/instances.html#instance-termination-rules` - ultima visita: 21/02/2023

[9] Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, Peter J. Stuckey - Understanding Functional Dependencies via Constraint Handling Rules - in: Journal of Functional Programming, Volume 17, Issue 1, January 2007, pp. 83 - 129

[10] Definizione di ClsInst in GHC, version 8.10.7 - url: `https://hackage.haskell.org/package/ghc-8.10.7/docs/InstEnv.html#t:ClsInst` - ultima visita: 21/02/2023

[11] Monomorphization - url: `https://doc.rust-lang.org/book/ch10-01-syntax.html#performance-of-code-using-generics` - ultima visita: 21/02/2023

[12] Type inference for Haskell, part 15 - url: `https://jeremymikkola.com/posts/2019_01_15_type_inference_for_haskell_part_15.html` - ultima visita: 21/02/2023

[13] Overlapping instances in Haskell - url: `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/instances.html#overlapping-instances` - ultima visita: 21/02/2023

[14] The GHC API, version 8.10.7 - url: `https://hackage.haskell.org/package/ghc-8.10.7` - ultima visita: 21/02/2023

[15] CoreSyn, version 8.10.7 - url: `https://hackage.haskell.org/package/ghc-8.10.7/docs/CoreSyn.html` - ultima visita: 21/02/2023

[16] Type, version 8.10.7 - url: `https://hackage.haskell.org/package/ghc-8.10.7/docs/Type.html` - ultima visita: 21/02/2023

[17] TyCon, version 8.10.7 - url: `https://hackage.haskell.org/package/ghc-8.10.7/docs/TyCon.html` - ultima visita: 21/02/2023

[18] DataCon, version 8.10.7 - url: `https://hackage.haskell.org/package/ghc-8.10.7/docs/DataCon.html` - ultima visita: 21/02/2023

[19] GHC Commentary: The Compiler - url: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/ - ultima visita: 21/02/2023

[20] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, Arnaud Spiwack - Linear Haskell - in: arXiv:1710.09756

[21] Effect Typing - url: https://koka-lang.github.io/koka/doc/book.html#why-effects - ultima visita: 21/02/2023

[22] Matija Pretnar - An Introduction to Algebraic Effects and Handlers. Invited tutorial paper - in: Electronic Notes in Theoretical Computer Science, Volume 319, 21 December 2015, Pages 19-35

[23] Effect types - url: https://koka-lang.github.io/koka/doc/book.html#sec-effect-types - ultima visita: 21/02/2023

[24] Jacques Garrigue - Programming with Polymorphic Variants - url: https://caml.inria.fr/pub/papers/garrigue-polymorphic_variants-ml98.pdf - ultima visita: 21/02/2023

[25] Koji Kagawa - Polymorphic Variants in Haskell - in: Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell, 17 Semptember 2006, Pages 37-47

[26] Data.Graph - url: https://downloads.haskell.org/ghc/latest/docs/libraries/containers-0.6.6/Data-Graph.html - ultima visita: 21/02/23

[27] Hindley, R. (1969). The Principal Type-Scheme of an Object in Combinatory Logic. Transactions of the American Mathematical Society, 146, 29–60. - url: https://doi.org/10.2307/1995158 - ultima visita: 21/02/23

[28] Robin Milner - A theory of type polymorphism in programming - in: Journal of Computer and System Sciences, Volume 17, Issue 3, December 1978, Pages 348-375

[29] Luis Damas, Robin Milner - Principal type-schemes for functional programs - in: POPL'82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, January 1982, Pages 207-212

[30] Inside Erlang, The Rare Programming Language Behind Whatsapp's Success - url: https://www.fastcompany.com/3026758/inside-erlang-the-rare-programming-language-behind-whatsapps-success - ultima visita: 25/02/23

[31] Cardano docs - url: https://docs.cardano.org/introduction - ultima visita: 25/02/23

[32] Hack - url: https://hacklang.org/ - ultima visita: 25/02/23

[33] Flow - url: https://flow.org/ - ultima visita: 25/02/23

[34] Template Meta-programming for Haskell - in: Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, 3 October 2002, Pages 1-16

[35] The LLVM Compiler Infrastructure - url: https://llvm.org/ - ultima visita: 25/02/23

[36] Fex-lang GitHub repository - url: https://github.com/bogo8liuk/Fex-lang - ultima visita: 27/02/23