YaleNUSCollege

A Study of Control and Type-Flow Analyses for Higher-Order Programming Languages

Gabriel Petrov

Capstone Final Report for BSc (Honours) in
Mathematical, Computational and Statistical Sciences
Supervised by: Ilya Sergey
AY 2020/2021

Yale-NUS College Capstone Project

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.

2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

√ Unrestricted access
Make the Thesis immediately available for worldwide access.

o Access restricted to Yale-NUS College for a limited period
Make the Thesis immediately available for Yale-NUS College access only from _____ (mm/yyyy) to _____ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
_____.

After this period, the Thesis will be made available for worldwide access.

o Other restrictions: (please specify if any part of your thesis should be restricted)
_____
_____

Gabriel Phoenix Petrov, Cendana College
Name & Residential College of Student

_____ 03/04/2021
Signature of Student Date

Prof. Ilya Sergey _____ 03 April 2021
Name & Signature of Supervisor Date

Acknowledgements

I can scarcely put into words how grateful I am to the following people, for shaping my experiences during my undergraduate education and helping me become the person I am today.

To my first computer science professor, Prof Aquinas Hobor, for inspiring me to pursue a journey in the area, despite the fact that prior to his class I didn't even know it was a field. Thank-you, for I have so far found that journey to be nothing short of incredible.

To my capstone advisor, Prof Ilya Sergey, without whom this thesis would not exist, for his gracious counsel, for his many classes, for describing the field of computer science as a forever incomplete jigsaw puzzle in sophomore year, and for helping me place the first few pieces down in their correct spaces.

To Tram and all of my friends - Adair, Alaukik, Alvin, Karolina, Leyli, Michael, Max, Ryan and more - for far too many moments to count.

To my sisters and my parents, the former, for growing up far faster than I could've thought possible and the latter for never failing to enable me in my endeavours, for always maintaining their quiet, but unyielding support. I know I don't rely on it often, but its presence is a comfort.

Finally, I am grateful to a Chuhu, whose smile somewhere out there never ceases to make the world shine a little more warmly.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

A Study of Control and Type-Flow Analyses for Higher-Order Programming Languages

by Gabriel PETROV

Higher-order programming languages take inspiration from existing models of computation, such as System F, in that they provide data and type abstraction. Whilst these abstractions often make it difficult to reason about the way a program behaves, there are methods that allow for just that. Control-Flow Analysis is a powerful static algorithm used to approximate what values a program's variables might take on during runtime. Analogously, Type-Flow Analysis statically approximates what type expressions may flow to various type variables. In this thesis, we semantically define and present an OCaml implementation of a Type and Control-Flow Analysis for System F. We further discuss its applications in compiler design, specifically related to the unsolved problem of monomorphization in functional programs.

Keywords: Control-Flow, Type-Flow, System F, Monomorphization

Contents

Chapter 1

Introduction

1.1 Motivation

Computers are only able to understand the language of binary, incredibly long sequences of ones and zeroes, which in turn are largely unreadable to humans. Thus, in order to bridge the gap between engineers and hardware, and by extension have our computers more seamlessly execute the more complex tasks required of them, the need arose for automatized translators of sorts that could convert human-understandable language-like code into machine-executable binary. These translators are called compilers.

Whilst the idea is simple, modern compilers are quite incredible and intricately complex in their design. The actual task of translation aside, compilers also generate many optimizations statically, i.e. without running the program, which results in more secure, automatically parallelized, more easily verifiable and generally higher-performance code, either in execution time or memory usage. Optimizations include "dead code elimination", which removes unreachable code, "reaching definitions" that replaces repeated arithmetic with constants and many others.

```
int c;
int a = 1, b = 10;
if (a > b)
    c = -1;
else
    c = 100;
...
```

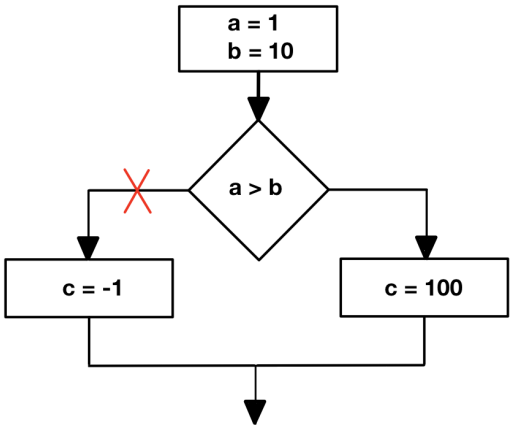

FIGURE 1.1: C code (left) and its CFG (right)

Consider Figure 1.1 that presents a small program in *C* and the program's control-flow graph (CFG). By simply following every vertex from the top in the CFG, we can establish at compile time, that the left branch of this program will never be reached: it is, in essence, "dead code". In fact, the whole if-statement could be replaced by the contents of the "else" branch. However, in order to determine this useful information, a compiler needs to know how a program behaves, which is where we introduce the first subject of this capstone: Control-Flow Analyses or CFAs. CFAs attempt to determine the flow of a program, as well as what values various program parameters take on, and are the reason that many optimizations a compiler generates for a program are even possible.

1.2 CFAs for Higher-Order Languages

Naturally, powerful algorithms such as this do not come for free. In some programming languages functions are treated as "first-class citizens": they can be passed as arguments, returned as output, assigned to

variables and called upon by those variables. This paradigm of programming is known as higher-order, or functional programming, which often results in less code redundancy, improved modularity and arguably less error-prone code, and is supported by an array of commonly used languages, including Haskell, OCaml, Scala, C++, and even Python. Among them, however, there is a problem known as the **higher-order control-flow problem**. In a language without higher-order functions, such as in our C example above, the arguments of a function or arithmetic are readily available, they are written in plain text within the code; thus the only challenge that remains is to simply parse the program, as its textual flow represents its control-flow.

However, in higher-order languages, the source of arguments of a function may not be immediately determinable from the code; instead they may rely on another "first-class" function elsewhere. To see how the control flow graphs of such languages become quickly convoluted due to their unique treatment of procedures, consider the following OCaml code:

```
let foo = fun f -> f 3 in
let bar = fun g -> g 4 in
let inc = fun x -> x + 1 in
foo inc + bar inc
```

Upon reading the definition of the procedure `foo` and later `bar`, it is not at all clear where `f` in the application `f 3` and `g` in the application `g 4` will come from, since both `f` and `g` are just function parameters. It instead depends on where the functions `foo` and `bar` will be used. What values will flow to `f` and `g` will only be determined at runtime, however what *might*

flow to `f` and `g` is what a CFA aims to track. Flow of values, however, is only one part of the story. You also need flow of types, which is where Type-Flow Analyses (TFAs) come in.

1.3 TFAs for Higher-Order Languages

Higher-order programming languages grant the user the freedom to implement functions that will work with any type, be they integers, strings, booleans, or any other. These functions are known as "polymorphic". In languages that allow for polymorphic types, such as Java, C#, C++, OCaml, Haskell and many others, it's perhaps intuitive to assume a degradation in runtime performance due to these universal types. To combat this, polymorphism is handled at compile time using optimizations such as monomorphization. Monomorphization is a strategy used to turn universally quantified code into specific and perhaps duplicated code by creating new entities for every concrete type used. An example is the easiest way to understand, so consider Figure 1.2.

```
template <T>
T max (T a, T b) {
    return (a>b?a:b);
}

int main () {
    int a = 0, b = 5;
    int c = max(a, b);

    float d = 3.5, e = 7.5;
    float f = max(d, e)
    return 0;
}
```

FIGURE 1.2: Polymorphic Code

We see in the example above that although `max` is defined with the polymorphic type `T`, it is only used with the `int` and `float` types. The

compiler will therefore generate two specialized versions of the `max` function:

```
int max_d (int a, int b) {
    return (a>b?a:b);
}
float max_f (float a, float b) {
    return (a>b?a:b);
}
```

This is the goal of monomorphization: conversion from polymorphic types to concrete, monomorphic ones. In order to allow for this optimization however, we need some way to know what types will flow to our `max` function in the first place. This we do by using TFAs.

1.4 Goals and Outline

The answer to the question "What values and types is the variable `f` able to take on?" is quite non-trivial and is the problem that the class of algorithms known as CFAs and TFAs attempt to solve. The rest of this capstone works towards the development of a Type and Control-Flow Analysis for functional languages. We've chosen System F as our representative of higher-order programming. System F, which is covered extensively in the following chapter, is a minimalistic, but highly expressive lambda calculus that has been the source of inspiration for many modern day functional languages, such as Haskell, the ML family, and, to varying extents, Java, C#, Scala and Rust [4, 7]. In order to build our TFA,

we must engineer our CFA and to do that, we first need to implement System F. Specifically, the contributions of this thesis are that it:

- presents a TCFA algorithm for a System F-like language
- presents an OCaml implementation of that TCFA
- presents testing and applications for it.

The remainder of this report is split into four chapters. Chapter 2 will describe System F, present a toy implementation and go further in-depth on the topic of CFAs & TFAs. Chapter 3 begins by formally defining a CFA and engineering an OCaml implementation of it, before describing the technical results obtained by this thesis, namely the expanded TCFA algorithm. Chapter 4 works through several examples of increasing complexity that demonstrate the TCFA in practice and discusses an edge case of TCFAs. Finally, Chapter 5 quickly explores what this thesis could not, showing future avenues of inquiry that would be interesting to follow through with and wraps it up with some concluding thoughts.

Chapter 2

Background And Overview

This chapter is a crash-course introduction to a few of the core concepts used in the rest of this capstone.

2.1 Lambda Calculus

Lambda calculus is a simple, yet extremely expressive formal system designed to denote computation, invented by Alonzo Church. Lambda calculus consists of three design elements, or terms, and the reduction operations that can be done on them. These terms include:

Variables :	x, y, z
Abstractions :	$\lambda x.t$
Application :	$t_1\ t_2$

TABLE 2.1: Lambda Calculus

where t, t_1 and t_2 are all lambda terms and t_2 must evaluate to a "value". A value in lambda calculus could be an integer, a boolean or, as it is a higher-order language, another function abstraction.

One of the simplest and arguably more important reduction operation in lambda calculus is β-reduction, whereby in the case of a function application $(\lambda x.t)\ v$, every bound appearance of the variable x in t is replaced by the value v, like so:

$$(\lambda x.t)\ v \implies t[v/x]. \tag{2.1}$$

To gain some intuition, let's take a look at an example. Data was first encoded in lambda calculus through the use of Church numerals, which are a way to express the natural numbers. The logical method to encode a natural number n is to use n of some term. Alonzo Church decided to use applications. The natural number n is thus encoded by a higher-order function that applies another function n number of times. Thus

$$\begin{aligned} 0 &= \lambda f.\lambda x.x \\ 1 &= \lambda f.\lambda x.f\ x \\ 2 &= \lambda f.\lambda x.f\ (f\ x), \end{aligned}$$

and so on. Church numerals are not meant to be a practical representation of numbers, but they do exist to show that no other terms are required to express computations. We can get the successor of these encodings via the procedure

$$succ = \lambda n.\lambda f.\lambda.xf(n\ f\ x).$$

Running $succ$ on $0 = \lambda f.\lambda x.x$ gives $\lambda f.\lambda.xf((\lambda f.\lambda x.x)\ f\ x) = \lambda f.\lambda x.f\ x$ via the β-reduction rules presented above in eq. (2.1), resulting in the

Church encoding of 1. We could imagine how by applying the successor function on a Church numeral n m number of times, we would achieve the addition of two Church encodings. The expressiveness of lambda calculus is truly wondrous.

A step up from lambda calculus lies simply-typed lambda calculus (STLC), which adds base types, such as integers or booleans, as well as the function type contructor $\rightarrow$ that builds function types, to our syntax. We add to Table 2.1 in the following way:

Types :	$Int \vert Bool \vert T \rightarrow T \vert \ldots$, where $T \in$ Types
Abstractions :	$\lambda x : T.t$

TABLE 2.2: Additional STLC Syntax

2.2 System F

System F further builds upon the notion of STLC by introducing polymorphic types, i.e. types that use the universal quantifier. This however introduces the need for variables that range over types as well as typed applications and typed abstractions. Thus, we add capitalized variables like X, Y, Z and the following two new terms to Table 2.2:

Type Abstractions :	$\Lambda X.t$
Type Application :	$t\ [X]$

TABLE 2.3: Additional System F Syntax

Though unlike lambda calculus, System F isn't Turing complete due to its lack of recursion (specifically expressing recursion with types), since System F builds upon lambda calculus it remains a simple, yet remarkably expressive way to represent a large number of programs. Its polymorphism is something that many modern languages have drawn inspiration from, thus studying it can improve the state of the art. Its simplicity also lends itself well to the many optimizations that compilers run on the languages they're compiling. Thus, some compilers, particularly those that work with functional programming languages, will often convert the native code into a form of System F (with recursion) and then run their analyses on that now Turing-complete System F, before continuing with the compilation. An example of this would be GHC that converts Haskell code into an intermediate representation called Core, where Core is one such example of pseudo-System F [3, 10].

During the initial explorations into this topic, we found it useful to develop a toy System F evaluator in OCaml. The terms that our toy System F language can take on are the same as discussed above, but notably is the notion of a value, which has changed partially from how we presented it in the lambda calculus section:

```
type value =
  | IntV of int
  | TypV of ty
  | Closure of environment * var * exp
and environment = (var * value) list
```

where in our implementation the type *ty* is allowed to take on *Int*, a variable type, a function type, and the polymorphic type. Returned values in

our toy System F evaluator therefore can be integers, types or closures.

The evaluator itself is reproduced below:

```
let rec eval env e =
  match e with
    | IntE i -> IntV i
    | VarE x -> lookup x env
    | LamE (arg, t, body) -> Closure(env, arg, body)
    | AppE (e1, e2) ->
        (match(eval env e1, eval env e2) with
           | (Closure(cenv, x, body), v) ->
              eval ((x,v)::cenv) body
           | _ -> failwith "applied non-function")
    | TypLamE (v, e') -> Closure(env, v, e')
    | TypAppE (e', t) ->
        (match(eval env e', t) with
           | (Closure(cenv, x, body), v) ->
               eval ((x,TypV v)::cenv) body
           | _ -> failwith "applied non-type")
```

The rules for integers and variables are quite intuitive: constants evaluate to themselves, whereas variables are looked up in the environment. Lambda abstractions, both on types and not, return closures. Finally applications use the provided closures to evaluate the body of those closures with a β-reduction operation (see eq. (2.1)).

This implementation of a System F evaluator is somewhat incomplete. It stops as soon as a closure is reached, but never tries to replace variables

inside the body of those closures with their corresponding values (i.e. β-reduction isn't implemented properly: it's only implied by the captured pair in the environment, meaning that another pass of the body is necessary to actually have the body reduced). Regardless, it is a toy example, aimed to aid familiarization with System F and provide a stepping stone into the main body of this capstone. In the following chapter we develop a language called **Fun** for "functional" and present how type and control flow information can be obtained statically, without evaluating the expressions.

2.3 Control-Flow Analyses

CFAs have already been introduced in Chapter 1. Among them there are different tiers of CFA algorithms, each with rising complexity. The naïve implementation of a CFA would be to assume that all lambda expressions present within the current function's scope (or worse yet, the whole program) can flow into every function application.

A somewhat more involved CFA would be the 0-CFA algorithm proposed by Shivers [8]. Shivers' algorithm is a context-insensitive control-flow analysis, meaning that every value is abstracted to the syntax from which it came, every function is abstracted to the term that created it, whilst its environment is ignored. This algorithm is still quite imprecise, however its benefit presents itself in its cubic-time bound, making it one of the more efficient algorithms in its class [5]. Broadly speaking, 0-CFA is implemented by the following logic:

1. Every function abstraction flows to itself

2. For every application $F\ t$, if the function abstraction $\lambda x.t'$ flows to F and the value v flows to the application's argument t, then v flows to x.

3. For every application $F\ t$, if the function abstraction $\lambda x.t'$ flows to F and the value v flows to the abstraction's body t', then v flows to the abstraction $F\ t$. [5]

Finally, there exist the even more precise k-CFA class of CFA algorithms, that improve the control-flow analysis precision via context sensitivity. An example of this would be Shivers' proposed 1-CFA algorithm that can perhaps amend 0-CFA's logic in the following way: the function abstraction $\lambda x.t$ flows to the variable y, when the function abstraction $\lambda y.t'$ is called from $F\ e$ [5, 9].

2.4 Type-Flow Analyses

A somewhat related analysis is presented by Matthew Fluet: a type-flow analysis [2]. We gained some intuition about TFAs in Chapter 1, but let's take a closer, more in-depth look. If a CFA claims that three functions - one of type $int \rightarrow int$, one $bool \rightarrow bool$ and one $char \rightarrow char$ - flow into the variable x, then if we can algorithmically assert somehow that x's static type can only be functions of type $int \rightarrow int$ and $bool \rightarrow bool$, then we can get a better approximation of our program, a much better one at that if x is originally defined as polymorphic. This information, the author claims, can by generated by a TFA. Thus, as CFAs provide useful approximations because it is unlikely that program variables are

bound to every function present at runtime, TFAs provide useful approximations because it is unlikely that types of variables are bound to every type at runtime.

It is worth noting that Fluet's analysis addresses System F with polymorphic recursion and makes multiple compromises in order to accommodate for the possibility of the infinite sets of types that can appear at runtime due to this polymorphism (we examine this problem in Section 4.2). Our goal in this manuscript is to study a more conservative version of System F without polymorphic recursion, as we hope to come up with a more principled and robust way to implement these analyses.

Chapter 3

Engineering TCFA

This chapter begins by formally defining a Control-Flow Analysis algorithm and presenting an OCaml implementation of it, in order to gain intuition about CFAs, before turning to the details of the brainchild of this manuscript: the Type and Control-Flow Analysis.

3.1 CFA Algorithm

The CFA algorithm presented in this section gathers control flow from a lambda calculus equivalent language **Fun**, defined below.

```
type label = int
type con = int
type var = string

type term =
  | ConT of con
  | VarT of var
  | FunT of var * e      (* λx.e *)
  | OpT of e * e
```

```
  | AppT of e * e          (* e1 e2 *)
and e = term * label
```

Fun only differs from lambda calculus via its inclusion of constants, binary operations and labels. Labels are a design choice necessary to identify different terms. However there are examples of control flow analyses that dispense with the use of labels: in fact many compilers use an intermediate representation in "continuation passing style", thus "labelling" every subterm by the use of a variable. We found labels to be more intuitive, but their use further presents the flexibility of the CFA and its ability to work with general functional programming languages [1].

Before moving on to the implementation of the CFA, we ought to first examine the theory behind how the algorithm is supposed to work. The result of a CFA is a pair, comprised of an abstract cache C and an abstract environment ρ that respectively map labelled points and program variables to abstract values. Abstract values meanwhile are simply abstractions of the form $\lambda x.e$. Constants are not considered to be abstract values as this is purely a Control Flow Analysis, without a Data Flow component, though it can be extended to include data flow. Thanks to the way C and ρ are designed, neither labelled points, nor program variables necessarily have to be distinct, but greater precision is of course to be gained by making sure they are unique.

Now let's consider the following example ex_1:

$$((\lambda x.x^1)^2\ (\lambda y.y^3)^4)^5$$

Ideally, our analysis will be able to conclude that $\lambda y.y^3$ flows to both x (via β-reduction, see eq. (2.1)) and the labelled points 1, 4, 5, and that $\lambda x.x^1$ flows to the labelled point 2. An acceptable, albeit somewhat useless result would be that both $\lambda x.x^1$ and $\lambda y.y^3$ flow to all labelled points and all program variables. However, an unacceptable result would be one that attempted to convince us that nothing flows to x, aka that x is unbound in the example. This would be semantically incorrect. Our analysis is allowed to have some degree of imprecision, but it must be sound.

Now that we know what, concretely, we are attempting to obtain, what remains of course is how our CFA manages to obtain sound and meaningful flow information about any expression e_* written in any **Fun**-like language. We do this by generating a set of constraints and conditional constraints of the form

$$lhs \subseteq rhs$$

$$\{t\} \subseteq rhs \implies lhs \subseteq rhs',$$

where t is a function abstraction term ($\lambda x.b$), lhs is of the form $\{t\}$, $C(l)$ or $\rho(x)$, and rhs is of the form $C(l)$ or $\rho(x)$, allowing us to represent how higher order bodies flow to one another [1]. The former reads "all evaluations of lhs may be observed at rhs", whereas the latter reads "rhs' may evaluate to lhs, provided that t is included in what rhs may evaluate to". After these constraints are obtained, a least solution to them needs to be computed, which will leave us with the control flow information of our expression.

Armed with these ideas, we turn to the implementation of our CFA. Below we first represent our two mappings for C and ρ, however it is worth noting that as neither of them is an actual map, these are simply the pure syntax used to represent the two:

```
type abstract =
  | AbsC of label    (* C(l) *)
  | AbsEnv of var        (* ρ(x) *)
```

Next we define our constraint types:

```
type constr =
  | TSub of term * abstract                (* {t} < rhs*)
  | ASub of abstract * abstract            (* lhs < rhs *)
  | Cond of
      term * abstract * abstract * abstract
      (* {t} < rhs => lhs < rhs*)
```

Finally, we move on to generating the constraints, which we do via the function C_*. There are five bodies that **Fun** expressions e_* may evaluate to, namely any of the **Fun**'s terms, paired with some label. Each of these versions of e_* will generate a different constraint, as in Table 3.1. The rule for constants collects no control-flow information, as we mentioned above. The rule for variables show that whatever a variable may evaluate to can be observed at the variable's program point. The rule for function abstractions show that the term can occur at the abstraction's label and recursively calls our constraint-generating function C_* on the abstraction's body. The rule for binary operations simply recursively calls C_* on both branches.

$$
\begin{array}{ll}
[ConT] & C_*[c^l] = \emptyset \\
[VarT] & C_*[x^l] = \{\rho(x) \subseteq C(l)\} \\
[FunT] & C_*[(fun\ x \Rightarrow\ \ e)^l] = \{\{fun\ x \Rightarrow e\} \subseteq C(l)\} \\
 & \qquad\qquad\qquad\qquad \cup\, C_*[e] \\
[OpT] & C_*[e_1\ op\ e_2] = C_*[e_1] \cup C_*[e_2] \\
[AppT] & C_*[(t_1^{l_1}\ t_2^{l_2})^l] = C_*[t_1^{l_1}] \cup C_*[t_2^{l_2}] \\
 & \qquad\qquad\qquad\qquad \cup \{\{t\} \subseteq C(l_1) \implies C(l_2) \subseteq \rho(x) \\
 & \qquad\qquad\qquad\qquad\quad \mid t = (fun\ x \Rightarrow t_0^{l_0})\} \\
 & \qquad\qquad\qquad\qquad \cup \{\{t\} \subseteq C(l_1) \implies C(l_0) \subseteq C(l) \\
 & \qquad\qquad\qquad\qquad\quad \mid t = (fun\ x \Rightarrow t_0^{l_0})\}
\end{array}
$$

TABLE 3.1: Constraint based CFA

Finally, the rule for applications recursively calls C_* on the abstraction and the argument, and then collects two conditional constraints. The first of the two reads "For all terms t of abstraction form $(\lambda x.t_0^{l_0})$, if t can be observed at program point l_1, then what flows to the labelled point l_2 flows to x", whereas the second reads "For all terms t of abstraction form $(\lambda x.t_0^{l_0})$, if t can be observed at program point l_1, then what flows to the labelled point l_0 flows to the labelled point l".

Transforming these constraints into code was relatively straightforward. We define a set type *Cons* that collects constraints and with it define the function `getConstraints` which takes a function abstraction list and an expression and returns a constraint set. We've elected to skip over the more simpler rules and below reproduce only the application rule:

```
| AppT ((t1, l1), (t2, l2)), l ->
    let c1 = getConstraints fn_ts tfn_ts (t1, l1) in
    let c2 = getConstraints fn_ts tfn_ts (t2, l2) in
```

```
List.fold_left (fun cons t -> match t with
  | FunT (v, ty, (t0, l0)) ->
    ConsS.add (Cond (t, AbsC l1, AbsC l0, AbsC l))
    (ConsS.add (Cond (t, AbsC l1, AbsC l2, AbsEnv v))
        cons)
  | _ -> failwith "bad fn term list"
) (ConsS.union c1 c2) fn_ts
```

Running `getConstraints` on our ex_1, $((\lambda x.x^1)^2\ (\lambda y.y^3)^4)^5$, returns the following constraints:

$$\{fun\ x \Rightarrow x^1\} \subseteq C(2);$$
$$\{fun\ y \Rightarrow y^3\} \subseteq C(4);$$
$$\rho(x) \subseteq C(1);$$
$$\rho(y) \subseteq C(3);$$
$$\{fun\ x \Rightarrow x^1\} \subseteq C(2) \implies C(1) \subseteq C(5);$$
$$\{fun\ x \Rightarrow x^1\} \subseteq C(2) \implies C(4) \subseteq \rho(x);$$
$$\{fun\ y \Rightarrow y^3\} \subseteq C(2) \implies C(3) \subseteq C(5);$$
$$\{fun\ y \Rightarrow y^3\} \subseteq C(2) \implies C(4) \subseteq \rho(y);$$

Now, as discussed above, we would like to find the least solution to this set. To do this, we turn to a graph formulation of the constraints. The graph's nodes will be a C(l) and $\rho(x)$ for all labels and variables in our expression. The solution will output a data field D, that maps entities to all entities that flow to them. A subset of the constraints produced by C_* will be described by an edge between nodes as so: a constraint $lhs \subseteq rhs$

will define an edge from *lhs* to *rhs*, whereas a constraint $\{t\} \subseteq rhs \implies lhs \subseteq rhs'$ gives rise to an edge from *lhs* to rhs' and from *rhs* to rhs'.

To find the least solution, what we must do is a sort of topological traversing of this newly constructed graph. We initialize D with all constraints of form $\{t\} \subseteq rhs$, because there is nothing to compute in those cases. We define a worklist W of nodes whose outgoing edges we would like to traverse, as well as an edge array E that for each node contains the constraints that allow computation of successor nodes. However, to make the algorithm efficient, we make sure to only traverse an edge from n_1 to n_2 if our $D[n_1]$ contains a term that wasn't previously there.

We define four helper functions to solve these constraints. They are `initialization`, `add`, `build_graph` and `iteration`.

As the name suggests `initialization` initializes our three data structures. We then use `build_graph` to populate the edge array E as described above, performing the initial assignments to D via the function call `add(q,d)`, which adds d to $D[q]$ and q to our worklist if d isn't already a subset of $D[q]$ (as discussed above).

Finally, we iterate through our worklist with `iteration` while the worklist isn't empty. For every edge of form $lhs \subseteq rhs$ we run `add(rhs, D[lhs])`; for every edge of form $\{t\} \subseteq rhs \implies lhs \subseteq rhs'$, in order to adhere to the conditional nature of our conditional constraint, we traverse the edge with `add(rhs', lhs)` only if in fact $t \in D[rhs]$. Finally we return our data array D [1].

In our ex_1, after the initialization and building of the graph we obtain a worklist $W : [C(4); C(2)]$, a data array with nodes $C(1)$ through $C(5)$,

$\rho(x)$ and $\rho(y)$ of which only $C(2)$ and $C(4)$ are non-empty and respectively point to $(\lambda x.x^1)$ and $(\lambda y.y^3)$, and an edge array with all the appropriate edges recorded. Once we iterate through our worklist, adding and removing from it as described above, we are left with a data array that looks like so,

$$C(1) \xleftarrow{flow} \{fun\ y \Rightarrow y^3\}$$
$$C(2) \xleftarrow{flow} \{fun\ x \Rightarrow x^1\}$$
$$C(3) \xleftarrow{flow} \emptyset$$
$$C(4) \xleftarrow{flow} \{fun\ y \Rightarrow y^3\}$$
$$C(5) \xleftarrow{flow} \{fun\ y \Rightarrow y^3\}$$
$$r(x) \xleftarrow{flow} \{fun\ y \Rightarrow y^3\}$$
$$r(y) \xleftarrow{flow} \emptyset$$

and is consistent with our original guess from when we first presented this example.

Thus we have our Control-Flow Analysis that can correctly obtain control-flow information about a program. The challenge that remains now is expanding this control-flow analysis to be able to gather type-flow information as well.

3.2 TCFA Algorithm

Expanding the CFA to incorporate types quickly proved to be quite involved. The groundwork was fairly simple; it required a new type to be defined, as below:

```
type ty =
  | IntTy
  | FunTy of ty * ty          (* ty → ty *)
  | VarTy of var
  | ForAllTy of var * ty      (* ∀X.T *)
```

and also for two new terms to be included in **Fun** and the *FunT* term to be edited in order to record its bound variable's type:

```
type term =
  ...
  | FunT of var * ty * e
  ...
  | TypLamT of var * e        (* type abstractions: \X. body *)
  | TypAppT of e * ty         (* type app: e [T] *)
```

The final change was to extend our abstract to be able to record type environments, like so:

```
type abstract =
  ...
  | AbsTEnv of ty             (* τ(T) *)
```

Now, having added the two new terms, we must edit our constraint collecting C_* function accordingly. We edit and add to Table 3.1 with the rules presented in Table 3.2. Editing our *FunT* rule is quite simple. To retrieve the constraints from a type abstraction is also fairly straightforward, as it closely followed the logic of a standard lambda abstraction.

Generating a constraint for the type application $t^l[T]$, was a bit more challenging to write. Similarly to a standard application, the constraint

$$
\begin{array}{ll}
[FunT] & C_*[(\mathit{fun}\ x : (T)\ \Rightarrow e)^l] = \{\{\mathit{fun}\ x : (T) \Rightarrow e\} \subseteq C(l)\} \\
 & \qquad\qquad \cup\, C_*[e] \\
[TypLamT] & C_*[(\mathit{tfun}\ X \Rightarrow e)^l] = \{\{\mathit{tfun}\ X \Rightarrow e\} \subseteq C(l)\} \\
 & \qquad\qquad \cup\, C_*[e] \\
[TypAppT] & C_*[(t_0^{l_0}\ [T])^l] = C_*[t_0^{l_0}] \\
 & \qquad \cup \{\{t\} \subseteq C(l_0)\ \Longrightarrow\ \tau(T) \subseteq \tau(X) \\
 & \qquad\quad \mid t = (\mathit{tfun}\ X \Rightarrow e)\}
\end{array}
$$

TABLE 3.2: Constraint-based TCFA

function must first extract all terms of type abstraction form. From then on, the process was somewhat more involved.

Initially, it seemed as if the two applications would differ in that the type application would generate different constraints depending on the type being applied (T), which could be four kinds of type: a ground type, meaning a type that contains no type variables within it, a variable type, a function type, or a polymorphic type.

If T were a ground type or a variable type, then the rule had to show that for all terms of type abstraction form $\backslash X.b$, T flows into X.

If T were a function type or a polymorphic type, it wasn't quite clear at first what the rule would be, but types are types, and just as in a standard application $e_1\ e_2$, e_2 can reduce to several different forms, so too could T. So eventually, the rule that was settled on was the same, regardless of T's internals, namely the constraint $\{t\} \subseteq C(l)\ \Longrightarrow\ \tau(T) \subseteq \tau(X)$.

Now came the biggest challenge of the project, namely collecting some type-flow information from our program. Unfortunately, simply using our constraint solver from the CFA module was not enough and provided

no type-flow information at all. This is because our solver does something different. On a conditional branch $\{t\} \subseteq C(l) \implies \tau(T) \subseteq \tau(X)$, the solver checks whether t is in fact something that program point l may evaluate to, and then sets the constraints that all types that flow to T will now flow to X. However, whilst that is useful, we would like to specifically find that T flows to X. We append our solver to include this information when the conditional part of our conditional constraint is satisfied.

Another issue is that despite tracking type-flow information for bound types, some type-flow information depends on our control-flow. To understand how, we must take a step back and look at the bigger picture. Consider this next example ex_2:

$$\Big(\lambda f : (\forall T, T \to T).\ (f\ [Int])\ 42\Big)\ id,$$

where *id* refers to the identity function that we've seen the type-less version of twice in ex_1, namely $\lambda x.x$, that now becomes $\Lambda X.\lambda x : (X).\ x$.

The crucial observation to be made is how types flow upon evaluation. The standard control-flow analysis gives us that *id* might flow to f. We see that f is instantiated with type Int inside of its body. Thus, ideally we would like to know that if *id* flows to f, then what flows to T (f's type), will flow to X (*id*'s type). Control flow affecting type flow.

A problem arises here, which is that we only learn *id* flows to f upon solving the constraints, not from the constraints themselves. Thus if we want to collect our type-flow information, we'll have to do it during our graph traversal.

We first need an environment that will match every variable with that

variable's type. This environment can be collected through use of a type-checker: an algorithm that will check the correctness of every variable's type within a program and can collect these pairs in the process. Next, during our graph traversal upon reaching an edge with conditional constraint form - $\{t\} \subseteq rhs \implies lhs \subseteq \rho(x)$ - we lookup said environment for x's type X. We then find all terms that flow to $\rho(x)$ by looking up our data array D and extract all types Y that are logically equivalent to X from all bound variables present within those terms. Finally, we update our data array with $\forall Y,\ d[\tau(X)] := d[\tau(X)] \cup Y$.

With that, we have successfully obtained the type and control-flow information in a program. In the following chapter we will go through some concrete examples and take a closer look at the algorithm's steps.

Chapter 4

Experiments

This chapter focuses on several case studies of increasing complexity and shows how our TCFA works in practice.

4.1 Case Studies

First, let's examine a simple type instantiation example on the identity function *id* with type Int:

$$\left(\left(tfun\ X \Rightarrow (fun\ x : (X) \Rightarrow x^1)^2\right)^3\ [Int]\right)^4$$

We run our `getConstraint` function, which generates the following constraints according to Table 3.2:

$$\{fun\ x : (X) \Rightarrow x^1\} \subseteq C(2);$$

$$\{tfun\ X \Rightarrow (fun\ x : (X) \Rightarrow x^1)^2\} \subseteq C(3);$$

$$\rho(x) \subseteq C(1);$$

$$\{tfun\ X \Rightarrow (fun\ x : (X) \Rightarrow x^1)^2\} \subseteq C(3) \implies \tau(Int) \subseteq \tau(x);$$

In order, we obtain our first constraint via the FunT rule when we reach program point 2; the second via the TypLamT rule upon reaching program point 3; the third via the VarT rule upon reaching program point 1 and finally the fourth conditional constraint via our TypAppT rule at the type application with Int.

To solve these constraints we initialize our data array with our function and type function terms flowing respectively into program points 2 and 3; our edge array with an edge from $\rho(x)$ to $C(1)$, from $C(3)$ to $\tau(x)$ and from $\tau(Int)$ to $\tau(x)$ and our worklist with $C(3)$ and $C(2)$.

Our iteration first takes a look at $C(3)$'s outgoing edges. As the nature of our conditional constraint demands, we will only explore these edges if in fact $\{tfun\ X \Rightarrow (fun\ x : (X) \Rightarrow x^1)^2\} \subseteq C(3)$, which a quick look at our data array tells us is in fact true. We thus append our data array to reflect that $\tau(Int)$ (and Int itself) indeed flows to $\tau(X)$. Our worklist looks up $C(2)$'s outgoing edges and upon finding nothing, the program terminates, leaving us with a data array that contains three noteworthy entries: $fun\ x : (X) \Rightarrow x^1$ flows to $C(2)$, $tfun\ X \Rightarrow (fun\ x : (X) \Rightarrow x^1)^2$ flows to $C(3)$ and Int flows to $\tau(x)$.

Now let's take a look at a more complicated example:

$$\Big(fun\ g : \Big((\forall S, S \to S) \to Int\Big) \Rightarrow \Big(g^1\ id^4\Big)^5\Big)^6 \Big(fun\ f : (\forall T, T \to T) \Rightarrow \Big((f^7\ [Int])^8\ 42^9\Big)^{10}\Big)^{11}$$

where $id = tfun\ X \Rightarrow (fun\ x : (X) \Rightarrow x^2)^3$ and henceforth

$$G = fun\ g : \Big((\forall S, S \to S) \to Int\Big) \Rightarrow \Big(g^1\ id^4\Big)^5$$

and

$$F = fun\ f : (\forall T, T \to T) \Rightarrow \left((f^7\ [Int])^8\ 42^9 \right)^{10}.$$

To avoid repeating already covered concepts, we will skip over the more straightforward sections of the algorithm, as the idea we are most interested in is whether the control-flow will affect our type-flow in such a nested example, where ideally we'd like to know that as F flows to g then g's argument type will flow to any equivalent types in F and also that as id flows to F, then F's argument type will flow to any equivalent types in id.

The function `getConstraint` generates 26 constraints. Our data array is initialized with the following information:

$$\begin{aligned} C(3) &\xleftarrow{flow} \{fun\ x : (X) \Rightarrow x^2\} \\ C(4) &\xleftarrow{flow} \{tfun\ X \Rightarrow (fun\ x : (X) \Rightarrow x^2)^3\} \\ C(6) &\xleftarrow{flow} \{G\} \\ C(11) &\xleftarrow{flow} \{F\}, \end{aligned}$$

our edge array with a large number of edges and our worklist with $C(4)$, $C(3)$, $C(6)$ and $C(11)$.

As the algorithm iterates, the first point of interest occurs during the exploration of $C(6)$'s outgoing edges. Of those, the one of note is the conditional constraint $\{G\} \subseteq C(6) \implies C(11) \subseteq \rho(g)$. As G does indeed flow to $C(6)$, we check our data array and find that F flows to program point 11, hence, F flows to $\rho(g)$.

But as we mentioned previously, in System F, control-flow affects type

flow. It is at this point that our TCFA looks up our environment for g's type $\Big((\forall S, S \to S) \to Int\Big)$, where only the arguments' type $(\forall S, S \to S)$ interests us. We then search through F and find all equivalent types, thus discovering T. Finally, we change our data array to reflect that S flows to $\tau(T)$. We append our worklist to reflect that we must now explore $\rho(g)$'s outgoing edges and continue iterating.

Similarly, a few iterations later, we find that $\{F\} \subseteq C(1) \implies C(4) \subseteq \rho(f)$, and as at this point F does in fact flow to $C(1)$, we check our data array and see that id flows to $C(4)$, hence id flows to $\rho(f)$. Following the same steps as above, we manage to extract the information that f's argument type T flows to the equivalent type X within id.

The algorithm terminates a few iterations later, producing quite a bit of information, but the noteworthy ones have already been covered. We also manage to collect that Int flows to X, much in the same way it was done in the first example of this chapter.

4.2 TFA and Higher-Kinded Types

There is one issue with this TCFA that remains unsolved and is quite tricky. Recall the notion of Church numerals, used as a representation for natural numbers in lambda calculus (see Section 2.1). Now let's consider one last case study *break*. In the interest of space and legibility, we revert back to standard System F notation in this Section. The term *break* relies on two other expressions,

$$ctwo = \Lambda X.\lambda f : (\forall X, X \to X).\ \lambda x : (X).\ f\ (f\ x)$$

and

$$\begin{aligned} cexp = \lambda m : (\forall X\,.((X \to X) \to X) \to X). \\ \lambda n : (\forall X.((X \to X) \to X) \to X). \\ \Lambda X.\ (n\ [X \to X])\ (m\ [X]), \end{aligned}$$

which refer to the second Church numeral and to the procedure that performs exponentiation on Church numerals. Meanwhile *break* looks like so:

$$((cexp\ ctwo)\ ctwo)\ [Int].$$

Running our TCFA on *break* collects quite a bit of information, but most notably

$$D[\tau(X)] = X \to X;\ Int;\ X.$$

At first glance, this looks exactly correct, and semantically it is. Unfortunately, this is the bane of all graph-traversing algorithms: a cycle. The tautology "what flows to X, flows to X", we can ignore. However, an attempt at expansion on the function type, in order to get a concrete one, would result in $X \to X$ flows to X, hence $(X \to X) \to (X \to X)$ flows to X, etc, all flowing to X... When instantiated with Int and expanded, $Int \to Int$ and $(Int \to Int) \to (Int \to Int)$, and $((Int \to Int) \to (Int \to Int)) \to \ldots$ are seemingly all possible values of X.

The issue is difficult to fix. It is a problem characteristic to universally quantified types that are instantiated with other universal types, thus making the first "higher-kinded" [6]. An extremely conservative approach would be to disallow any function types using type abstraction variables (like $X \to X$). This is not ideal. However, as far as we know, no

other less conservative approach has been rigorously proven. Perhaps there is a way to address this circularity that TCFAs run into with languages that do not implement full-blown polymorphic recursion, or for specific applications like monomorphization. This is further discussed in the next chapter's Section 5.1.

Chapter 5

Future Work and Conclusion

This chapter describes some exciting future avenues of inquiry and caps this manuscript off with some concluding thoughts.

5.1 Future Work

The focus of this work has been to extend established Control-Flow Analyses with an additional Type-Flow Analysis that works with System F. Future work would focus on three avenues: additions to our current TCFA, randomized stress testing of this project, and finally exploring ways to interface with existing state of the art compilers.

An interesting addition to our current TCFA would be to extend it with the ability to collect Data Flow information. In simple terms this would involve extending our set of values to include abstract values besides abstractions, specifically constants. This would of course involve appending our constant rule in Table 3.1, changing it to something similar to our variable rule, but also our binary operator rule would likely have to be appended as well. We would finally have to explore how the

addition of the data flow information can affect our control flow, in order to glean more precise information about the program.

Furthermore, in order to stress test this project well, the ability to interface with a random System F generator would prove especially effective. Although the actual checking of the results obtained would for now have to remain manual, as designing a test oracle that could decide autonomously whether a certain output is correct or not sounds remarkably difficult, the task of interfacing with such a generator doesn't intuitively seem like it would present too serious a challenge. As our language **Fun** is effectively System F, all that would be required is likely a simple lexer/parser that reads through whatever examples the random System F generator creates, transforms them into **Fun** and runs them through our TCFA. Whilst testing, especially compiler testing, is arguably not as common a practice as it perhaps should be, luckily such random System F generators do exist.

Finally, the ability to interface with a state of the art compiler for higher-order languages would be the pinnacle of future work. The one challenge that remains is the long-standing issue of monomorphization in System F and functional programs in general. To avoid higher-kinded types, a robust proof of whether forms of recursion could be implemented without running into cases such as our example from Section 4.2, would be a great leap forward in the topic of TFAs. The TCFA presented here could then help compilers obtain useful control-flow information about their programs, but also allow them to perform monomorphization on polymorphic functions, thanks to the type-flow information obtained.

5.2 Conclusion

The results obtained in Chapter 3 that formally define and present an OCaml implementation of a Type and Control Flow Analysis, with the ability to collect practically precise control and type-flow information from a program.

In this work we've presented a way to extend established Control-Flow Analyses with an additional Type-Flow Analysis on functional programming languages, but in a systematic and robust manner, without attempting to implement polymorphic recursion, and thus without making the many concessions one would have to make in that case. We do this by use of the lambda calculus System F, which is a simple, but rich way to express programs of higher-order languages. We demonstrate the TCFAs ability to work with examples of varying degrees of complexity, and present an edge case with an interesting problem that raises questions worthy of future exploration. With that, we conclude this capstone.

Bibliography

[1] Chris Hankin Flemming Nielson Hanne Riis Nielson. *Principles of Programming Analysis*. Springer, 1999, pp. 141–209.

[2] Matthew Fluet. "A Type- and Control-Flow Analysis for System F: Technical Report". In: *Rochester Institute of Technology Scholar Works* (2013).

[3] GHC Core Team. *Glasgow Haskell Compiler: The Core Type*. 2020. URL: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/core-syn-type.

[4] Jean-Yves Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. Université Paris 7, 1972.

[5] Matthew Might. *k-CFA: Determining types and/or control-flow in languages like Python, Java and Scheme*. 2010.

[6] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[7] John C. Reynolds. "Towards a theory of type structure". In: *Programming Symposium*. Vol. 19. LNCS. Springer, 1974, pp. 408–423.

[8] Olin Shivers. “Control-flow analysis in Scheme”. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Languages Design and Implementation, M. D. Schwartz, Ed. Atlanta, Georgia*. ACM. 1988, 164–174.

[9] Olin Shivers. “Control-flow analysis of higher-order languages or taming lambda”. PhD thesis. Pittsburgh, Pennsylvania: School of Computer Science, Carnegie Mellon University, 1991.

[10] Vladislav Zavialov. *Haskell to Core: Understanding Haskell Features Through Their Desugaring*. 2020.