Gradual Program Analysis

Samuel Estep

A Senior Thesis submitted in partial fulfillment
of the requirements for graduation
in the Honors Program
Liberty University
Spring 2020

Acceptance of Senior Honors Thesis

This Senior Honors Thesis is accepted in partial
fulfillment of the requirements for graduation from the
Honors Program of Liberty University.

____
Ethan C. Smith, Ph.D.
Thesis Chair

____
Daniel J. Majcherek, Ph.D.
Committee Member

____
David E. Schweitzer, Ph.D.
Assistant Honors Director

____
Date

Abstract

Dataflow analysis and gradual typing are both well-studied methods to gain information about computer programs in a finite amount of time. The gradual program analysis project seeks to combine those two techniques in order to gain the benefits of both. This thesis explores the background information necessary to understand gradual program analysis, and then briefly discusses the research itself, with reference to publication of work done so far. The background topics include essential aspects of programming language theory, such as syntax, semantics, and static typing; dataflow analysis concepts, such as abstract interpretation, semilattices, and fixpoint computations; and gradual typing theory, such as the concept of an unknown type, liftings of predicates, and liftings of functions.

Gradual Program Analysis

Introduction

This thesis lays out the background material for the Gradual Program Analysis project investigated by this author in collaboration with Jenna Wise, Jonathan Aldrich, and Joshua Sunshine from Carnegie Mellon University; Éric Tanter from the University of Chile; and Johannes Bader from Facebook. The project's initial goal was to understand how the static analysis tool Facebook Infer manages different types of uncertainty when analyzing source code for null-pointer bugs. Later goals included systematizing this uncertainty management via techniques inspired by the work on gradual program verification (Bader, Aldrich, & Tanter, 2018), implementing this new program analysis as a new checker in Infer, and generalizing the approach to other types of analyses besides null-pointer analysis. The following sections in this introduction elaborate on these motivating factors more. The rest of this thesis will lay out the relevant technical details.

Facebook Infer

Infer is a static analysis tool developed by Facebook, which is used to detect bugs in source code written in the languages Java, C, C++, and Objective-C. The tool includes several built-in bug checkers, such as Eradicate, which finds null-pointer bugs. Infer also includes a framework for defining new checkers based on abstract interpretation.

Null-pointer Analysis

We initially developed gradual analysis in the specific context of *gradualizing* null-pointer analysis, and then later generalized it to other analyses. A pointer is a number that refers to, or *references*, a location in a computer's memory. A program can *dereference* a pointer to make use of the data which the pointer references. However, in many popular languages, a pointer can also be *null*, which means that it actually references nothing. In this case, it is a fatal error to dereference that pointer. This is one of the most common types of bugs in computer programs, to the point that Tony Hoare (2009), the inventor of the null pointer, has called it his "billion-dollar mistake". The goal of null-pointer analysis is to automatically find null-pointer bugs in computer programs, or otherwise ensure the absence of such bugs.

False Positives

A static analysis program reads the source code of another program and returns zero or more *warnings* about potential issues. If a *sound* static analysis returns no warnings, that serves as a proof that the program has no issues of the type for which the analysis was searching. However, this approach tends to produce many false positives, making it difficult for programmers to distinguish signal from noise when using such an analysis tool. Thus, many tools in industry make unsound assumptions in order to cut down on the number of false positives produced. These unsound tools can have false negatives, in which no warnings are produced but the analyzed program still has bugs.

Gradual Verification

Similar to static analysis, the technique of static verification allows programmers to have the computer check certain properties of their programs. The ongoing work on gradual verification seeks to make these tools easier to use. Gradual verification acknowledges that some properties are difficult to verify statically, and thus instead modifies programs to automatically check that those properties are satisfied at runtime. This prevents the program from wandering into undesired behavior. The gradual analysis project takes inspiration from gradual verification, also inserting runtime checks to augment its (hopefully reduced) set of static warnings.

Relevant Literature

This work essentially combines gradual typing with abstract-interpretation-based static analysis, also known as dataflow analysis. Both topics are explored in much existing literature.

Dataflow Analysis

CMU program analysis class. The Dataflow Analysis section below will draw heavily from the program analysis course at Carnegie Mellon University (Aldrich, 2019), specifically the lectures “Introduction, Program Representation, and Syntactic Analysis” and “Dataflow Analysis and Abstract Interpretation”.

Abstract interpretation. Cousot and Cousot (1977) introduced the technique of abstract interpretation for static analysis.

Kildall fixpoint algorithm. A few years before that, Kildall (1973) had introduced an efficient algorithm that can be used for computing the fixpoint of a dataflow analysis.

Gradual Typing

Siek and Taha (2006) developed *gradual typing* to combine static typing and dynamic typing within the same language.

Refined criteria for gradual typing. After the term became popular in the following decade, Siek, Vitousek, Cimini, and Boyland (2015) introduced a formal property called the *gradual guarantee* to make precise the distinction between languages that are gradually typed and languages that are not.

Abstracting Gradual Typing. More recently, the *Abstracting Gradual Typing* (AGT) framework provides a systematic way to transform a static type system into a gradual type system that automatically satisfies the gradual guarantee (Garcia, Clark, & Tanter, 2016).

Null-pointer Analysis

Besides Infer's Eradicate, the gradual analysis project drew inspiration from a few other modern null-pointer analysis tools.

Granullar. The Granullar type system (Brotherston, Dietl, & Lhoták, 2017) uses gradual typing and nullability type annotations to analyze for null pointers in only some parts of the code, by letting the programmer set boundaries between checked and unchecked code.

NullAway. Uber's NULLAWAY (Banerjee, Clapp, & Sridharan, 2019) reduces the annotation burden of static pluggable type checking for null-pointer exceptions through targeted unsound assumptions. They aim for no false negatives in practice on checked code.

Preliminaries

Syntax

Programming languages must specify the set of permissible programs. In an *imperative* language, that specification would probably include a specification of the set of permissible *instructions*, such as that shown in Figure 1.

This imperative syntax definition can be thought of as a shorthand for defining a set of strings. The "$x, y, z \in \textsc{Var}$" part means that we already have a set of names VAR for variables in our programs, and that in the rest of the specification, the letters x, y, and z can each be replaced by any of those names in order to form a valid instruction. Then $I ::= \cdots$ defines the set I as the union of several smaller sets of instructions, each of which is separated by

$$\begin{array}{rcl} x, y, z & \in & \textsc{Var} \\ I & ::= & x \texttt{ := null} \mid x \texttt{ := cons } y\texttt{, } z \mid x\texttt{, } y \texttt{ := \&}z \\ & & \mid x \texttt{ := } y \mid \texttt{if } x \mid \texttt{if not } x \end{array}$$

Figure 1. Syntax for an example imperative language.

a vertical bar |. For instance, "x `:=` y" corresponds to the set $\{x \texttt{ := } y : x, y \in \textsc{Var}\}$. As more concrete examples, if we have `foo`, `bar`, `baz` $\in$ VAR, then

`foo := cons bar, bar` and `baz := null` and `if foo` and `bar, baz := &foo`

are all valid instructions in the set I.

In a *functional* language, programs are sometimes referred to as *terms*, which might be syntactically defined as follows:

$$x \in \textsc{Var}$$

$$t ::= x \mid \lambda x.t \mid t\ t$$

$$v ::= \lambda x.t$$

This is a recursive definition, so if $x, y, f \in \textsc{Var}$ then

$$x \quad \text{and} \quad \lambda x.x \quad \text{and} \quad (\lambda x.x)\ y \quad \text{and} \quad \lambda f.\lambda x.f\ x \quad \text{and} \quad \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

are all valid terms. Note that parentheses are sometimes necessary for disambiguation. Finally, the nonrecursive definition v defines *values*, which are essentially terms that are fully evaluated. The next section will clarify this notion of evaluation.

Small-step Operational Semantics

It is also valuable to formalize the runtime behavior of programs. One way to do this is to define a relation $\longrightarrow$ on the set of program states. In functional languages such as the lambda calculus defined syntactically above, a program state is just a term. We read $t_1 \longrightarrow t_2$ as "t_1 *reduces* to t_2 in one step." This relation might be defined using inference rules as follows (Pierce, 2002):

$$\textsc{App1}\,\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \textsc{App2}\,\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \textsc{AppAbs}\,\frac{}{(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}}$$

This syntax defines an inference rule by listing the premises above the line and the conclusion below, with the rule name written to the side. For instance, the App1 rule says that if t_1 reduces to t_1' in one step, then the*function call* $t_1\ t_2$ reduces to $t_1'\ t_2$ in one step. In other words, simplify function definitions before applying those functions to values. In the AppAbs rule, the syntax $[x \mapsto v_2]t_{12}$ means to take t_{12}, and replace every instance of the variable x with the value v_2. As it turns out, this substitution is not completely trivial, since sometimes variable names are not unique, but we will not go into those subtleties here.

This relation is of course not transitive, but these step reductions can be chained to fully evaluate terms. Often the notation $\longrightarrow^*$ is used to denote the transitive closure of $\longrightarrow$; for example, consider the following reduction (Trunov, 2016), using two AppAbs steps:

$$(\lambda x.x\ y)\ (\lambda y.y\ z) \longrightarrow (\lambda y.y\ z)\ y \longrightarrow y\ z, \qquad \text{so} \qquad (\lambda x.x\ y)\ (\lambda y.y\ z) \longrightarrow^* y\ z$$

Dataflow Analysis

The first half of gradual program analysis is static program analysis, more specifically dataflow analysis.

Abstract Interpretation

For simplicity, we will just use the natural numbers $\mathbb{N}_0 = \{0, 1, 2, \dots\}$ to represent the set of pointers. Conventionally, `null` $= 0$. Then for the language defined in Figure 1, at any given point during program execution, each variable could take on any of these pointer values in $\mathbb{N}_0$. We could formalize this mapping from variables to pointer values (as well as their evolution over time, and the memory which the pointers reference) in the form of semantics, but we will not do that here. We will define an abstract interpretation instead of a concrete one.

First we choose a set $L = \{N, P, \top\}$ of abstract values. Each of these represents a set of concrete values in $\mathbb{N}_0$, as shown in the following correspondence:

$$N \leftrightsquigarrow \{0\}, \quad P \leftrightsquigarrow \mathbb{Z}^+ = \{1, 2, \dots\}, \quad \top \leftrightsquigarrow \mathbb{N}_0 = \{0, 1, 2, \dots\}$$

Then in our analysis, we will work with maps of the form $\sigma : \textsc{Var} \rightharpoonup L$. For instance at a particular program point we might have $\sigma(x) = N$, $\sigma(y) = P$, and $\sigma(z) = \top$, meaning that we know x will definitely be null when the program reaches that point, y will definitely not be null at that point, and z may or may not be null.

Flow Function

Then the question arises of how we abstractly evaluate our program from one point to the next. In other words, if the abstract state is σ before an instruction $\iota \in I$, how can we update σ to reflect the set of possible program states after ι is executed? We do this by defining a *flow function*:

$$f[\![x \texttt{ := null}]\!](\sigma) = \sigma[x \mapsto N]$$

$$f[\![x \texttt{ := cons } y\texttt{, } z]\!](\sigma) = \sigma[x \mapsto P]$$

$$f[\![x\texttt{, } y \texttt{ := \&}z]\!](\sigma) = \sigma[z \mapsto P][x \mapsto \top][y \mapsto \top]$$

$$f[\![x \texttt{ := } y]\!](\sigma) = \sigma[x \mapsto \sigma(y)]$$

$$f[\![\texttt{if } x]\!](\sigma) = \sigma[x \mapsto P]$$

$$f[\![\texttt{if not } x]\!](\sigma) = \sigma[x \mapsto N]$$

On the left side of each equality, we specify the subset of instructions to which that line applies. The instruction template is typeset in $[\![]\!]$ for clarity, but it is just a parameter to the function f. Then on the right, we return a new map $\sigma' : \textsc{Var} \rightharpoonup L$, in this case always by updating one or more of the existing mappings in σ. For instance, if ι is $x \texttt{ := } y$ and $\sigma(x) = \top$ and $\sigma(y) = N$, then after passing through our flow function to get $\sigma' = f[\![\iota]\!](\sigma)$, we have $\sigma'(x) = \sigma'(y) = N$.

Control Flow Graph

The instructions `if` x and `if not` x allow us to do logical decision-making by culling program states in which x is null or non-null, respectively. We represent this branching by arranging instructions into a *control flow graph*, such as the one shown in Figure 2.

To evaluate this program concretely, we first assign the local variable `a` the value `null`. We then `cons`truct a *pair*, both of whose cells take on the contents of the variable `a`, which is in this case `null`; thus the new pair is (`null`, `null`). We store that pair somewhere in memory and

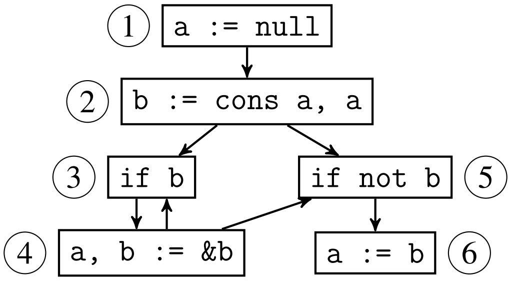

Figure 2. Control flow graph for an example program.

assign to `b` a pointer value that is the location in memory where we stored the pair. Since `b` is now not `null`, we follow the `if b` branch. We dereference `b` to get its two contents `null` and `null`, and assign them to `a` and `b` respectively. (Of course, `a` was already `null`, so this doesn't change `a`). Now that `b` is `null`, we follow the `if not b` branch, and assign `a` to take on the value of `b`, which again doesn't change `a`.

Semilattice with Join

When evaluating a program concretely as we did in the previous section, we are only ever at a single program point. We follow the evaluation deterministically. However, this has the drawback that we do not know how long the program will take to finish executing. It may even have an infinite loop, and thus never finish. This is the reason we use abstract interpretation in the first place: it allows us to obtain definitive information about how a program will execute, in a finite amount of time.

Recall the flow function f from earlier. If we have a map $\sigma : \textsc{Var} \rightharpoonup L$ that describes all the possible program states before an instruction ι executes, then we can take $\sigma' = f[\![\iota]\!](\sigma)$ to describe all the possible program states after ι executes. But how do we know what to use for σ?

For instance, in Figure 2, there are two incoming edges to node ⑤. If we take the analysis data directly from ②, then we have $\mathtt{a} \mapsto N$ because that is how f interprets `a := null`. On the other hand, if we take the analysis data from ④, then we have $\mathtt{a} \mapsto \top$, because that is how f interprets `a, b := &b`. We need some way to combine both of these analysis results to encompass all of the possible states leading into ⑤.

To do this, we can give L a semilattice structure via the subset relation $\subseteq$, resulting in the Hasse diagram shown in Figure 3. This partial order induces an associative, commutative, idempotent (Davey & Priestley, 2002) join operation $\sqcup : L \times L \to L$ defined by $N \sqcup P = N \sqcup \top = P \sqcup \top = \top$ where the rest of the cases are given by the aforementioned properties. Since $\sqcup$ is idempotent, if all the incoming edges to a node agree with each other, then σ is the same as all of them. But if two incoming edges disagree, this operation allows us to take, for instance, $\sigma(\mathtt{a}) = N \sqcup \top = \top$ for the case mentioned in the previous paragraph.

Fixpoint Algorithm

To tie all these pieces together, we use a fixpoint algorithm (Kildall, 1973) to compute the final table mapping variables to abstract values in L for every point in the program, as shown in Table 1. We start with an empty table, depicted on the lefthand side. Then we apply the flow function to node ①, yielding $\mathtt{a} \mapsto N$, and propagate these results to node ② which follows. Applying f to ② then yields $\mathtt{b} \mapsto P$, which we propagate to ③ and ⑤. Next we analyze ③ and propagate to ④, which retains the same results because we already had $\mathtt{b} \mapsto P$. After analyzing ④, we now have $\mathtt{a} \mapsto \top$ and $\mathtt{b} \mapsto \top$, so we $\sqcup$ these into ③ and ⑤. Since the analysis leading into ③ has been updated, we need to analyze it again and propagate the resulting $\mathtt{a} \mapsto \top$ to ④. Finally, we analyze ⑤ and propagate the results to ⑥.

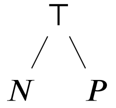

Figure 3. Hasse diagram for our null-pointer analysis semilattice, ordered by ⊆.

Table 1

Analyzing the program in Figure 2 until reaching a fixpoint.

			①		②		③		④		③		⑤	
	a	b	a	b	a	b	a	b	a	b	a	b	a	b
①														
②			N		N		N		N		N		N	
③					N	P	N	P	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
④							N	P	N	P	$\top$	P	$\top$	P
⑤					N	P	N	P	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
⑥													$\top$	N

The preceding narration illustrates the general idea of the algorithm, which is that, starting at the entry point to the program, we take the analysis at a node, run it through f, and use $\sqcup$ to propagate those results to the nodes which follow. If any of those nodes actually changed, we add them to the queue of nodes we will run through f next. Once the queue becomes empty, the algorithm terminates and we have our final analysis table. Conveniently, there is room for ambiguity in this algorithm; the one we describe here is chosen for its decent performance, but using the properties of L and f, one can prove that the same fixpoint will eventually be reached by any algorithm that does not neglect specific nodes in the graph.

As shown in Table 1, these final analysis results give information about the original program: for instance, `b` will never be `null` when ④ is executed, and will always be `null` when ⑥ is executed. However, our analysis is not completely precise: for instance, even though `a` is always `null`, our analysis deems it $\top$ at all program points past ②, because we update it from a value retrieved from memory, which our analysis does not model very well. In general, Rice's theorem (Rice, 1953) tells us that it is impossible to create a perfectly precise, sound static analysis. This is where the ideas from gradual typing come in.

Gradual Typing

Type systems are another well-studied tool for getting static information about programs. The field of gradual typing seeks to address the inherent imprecisions and inconveniences of type systems by allowing them to be not just *pessimistically* uncertain, but also *optimistically* uncertain. This section mirrors the presentation in Garcia et al. (2016).

Typing Rules and Judgments

We start by defining syntax for a slightly more complicated lambda calculus, with types.

$$\begin{aligned}
T &\in \textsc{Type}\\
x &\in \textsc{Var}\\
b &\in \textsc{Bool}\\
n &\in \mathbb{Z}\\
t &\in \textsc{Term}\\
T &::= \mathsf{Int} \mid \mathsf{Bool} \mid T \to T\\
t &::= n \mid b \mid x \mid \lambda x : T.t \mid t\ t \mid t + t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid t :: T
\end{aligned}$$

Intuitively, a type corresponds to a set of values with operations on values in that set. Thus Int is the type of integers, and Bool is the type of booleans—that is, true or false. We can simply write out an integer in $\mathbb{Z}$ or a boolean in the set $\textsc{Bool}$ as a term. Then terms also include variables x, lambda abstractions (functions) $\lambda x : T.t$, and applications (function calls) $t\ t$. Note that the syntax for a lambda abstraction now requires that we declare the type of the parameter x. We also include some syntax specific to the new types we have introduced, such as sums and branches. Finally, we have syntax $t :: T$ for *casting* a term to be of type T. This is useless in the statically typed context, but it will become more useful when we gradualize this language.

In contrast to the fixpoint calculation required for dataflow analysis, type systems often proceed using inference rules such as the ones in Figure 4 to produce *typing judgments* on terms. These inference rules follow the same format as the ones we used to define operational semantics of the untyped lambda calculus earlier. If there exists a valid typing derivation for a given term, then that term is said to be *well-typed*. For instance, the following derivation shows that in a context where x is of type Int, the term $x + 42$ is also of type Int:

$$\text{(T+)}\,\frac{\text{(Tx)}\,\dfrac{x : \mathsf{Int} \in x : \mathsf{Int}}{x : \mathsf{Int} \vdash x : \mathsf{Int}} \qquad \text{(Tn)}\,\dfrac{}{x : \mathsf{Int} \vdash 42 : \mathsf{Int}} \qquad \mathsf{Int} = \mathsf{Int} \qquad \mathsf{Int} = \mathsf{Int}}{x : \mathsf{Int} \vdash x + 42 : \mathsf{Int}}$$

This knowledge could then be used to prove, for instance, that in any context, if you take the function $\lambda x : \mathsf{Int}.x + 42$ and apply it to the number 8, the result is of type Int:

$$\text{(Tapp)}\,\frac{\text{(T}\lambda\text{)}\,\dfrac{x : \mathsf{Int} \vdash x + 42 : \mathsf{Int}}{\vdash \lambda x : \mathsf{Int}.x + 42 : \mathsf{Int} \to \mathsf{Int}} \qquad \text{(Tn)}\,\dfrac{}{\vdash 8 : \mathsf{Int}} \qquad \mathsf{Int} = \mathrm{dom}(\mathsf{Int} \to \mathsf{Int})}{\vdash (\lambda x : \mathsf{Int}.x + 42)\ 8 : \mathsf{Int}}$$

The conclusion of this particular typing judgment should be obvious: simply replace x with 8, then add it to 42 to yield 50, which is an integer. However, in general it is not always feasible to evaluate a term to determine its type. Some terms may take an exponential (or greater) amount of

$$\textsc{Tx}\,\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{Tn}\,\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \textsc{Tb}\,\frac{}{\Gamma \vdash b : \mathsf{Bool}}$$

$$\textsc{Tapp}\,\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad T_2 = \mathrm{dom}(T_1)}{\Gamma \vdash t_1\; t_2 : \mathrm{cod}(T_1)}$$

$$\text{T+}\,\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad T_1 = \mathsf{Int} \qquad T_2 = \mathsf{Int}}{\Gamma \vdash t_1 + t_2 : \mathsf{Int}}$$

$$\textsc{Tif}\,\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash t_3 : T_3 \qquad T_1 = \mathsf{Bool}}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : \mathrm{equate}(T_2, T_3)}$$

$$\text{T}\lambda\,\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x : T_1 . t) : T_1 \to T_2} \qquad \text{T::}\,\frac{\Gamma \vdash t : T \qquad T = T_1}{\Gamma \vdash (t :: T_1) : T_1}$$

Figure 4. Typing rules for our simply-typed lambda calculus.

time to reduce in relation to their size, whereas the time it takes to typecheck a given term is always linear in the size of that term.

Unknown Types

We would like to insist that every term be well-typed, since it can be shown that this ensures the program will not get stuck (that is, reduce to a nonfinal point at which further reduction is undefined). Unfortunately, this requirement is problematic for a couple of reasons. First, our type system is *strongly normalizing*, so every well-typed term will eventually fully reduce, producing a value. This implies that our simply-typed lambda calculus is not Turing-complete: it is impossible to knowingly exclude all non-terminating programs without also excluding some interesting terminating ones as well. As a trivial example, it is impossible to write an interpreter for our language in itself, but there are also many other interesting programs we cannot write using this type system. More sophisticated type systems, such as System F

(Girard, 1971), provide far greater expressive power while still retaining strong normalization, but in the end, that property is incompatible with Turing-completeness.

Second, it can be very difficult to write types even for programs that are possible to write in a given type system. Often, when initially designing a program, the programmer does not have the clearest idea in their head of what the types should be. For these cases, *dynamically-typed* languages such as Python are often used to iterate quickly on an idea. Unfortunately, as a program grows into a larger-scale system, the absence of static types tends to make maintenance more and more difficult. At that point, the system is already written in a dynamic language, so the addition of static types would require first porting the program to a different language, which is almost never feasible. It would be desirable to have a more friendly migration path from a dynamically typed program to a statically typed one; this is the essence of *gradual typing*.

As mentioned previously, this thesis uses the AGT approach to gradual typing, which starts by extending the set of types to include an *unknown type* ?:

$$\begin{aligned}
\widetilde{T} &\in \textsc{GType} \\
\widetilde{t} &\in \textsc{GTerm} \\
\widetilde{T} &::= ? \mid \mathsf{Int} \mid \mathsf{Bool} \mid \widetilde{T} \to \widetilde{T} \\
\widetilde{t} &::= \ldots \mid \lambda x : \widetilde{T}.\widetilde{t} \mid \widetilde{t} :: \widetilde{T} \mid \ldots
\end{aligned}$$

Notice that $\textsc{Type} \subseteq \textsc{GType}$ and $\textsc{Term} \subseteq \textsc{GTerm}$. Thus, any program in the original (static) language is also a program in this extended *gradual* language. However, since ? is now a type, this language also allows an embedding of the untyped lambda calculus we discussed previously. This new language provides the entire spectrum of *typedness*, from completely untyped to completely typed.

Concretization

In order to use our gradually typed language, we must give meaning to the new types which we have defined. AGT does this via a function $\gamma : \textsc{GType} \to \mathcal{P}(\textsc{Type})$ given by

$$\gamma(\mathsf{Int}) = \{\mathsf{Int}\} \qquad \gamma(\mathsf{Bool}) = \{\mathsf{Bool}\}$$

$$\gamma(\widetilde{T}_1 \to \widetilde{T}_2) = \gamma(\widetilde{T}_1) \mathrel{\widehat{\to}} \gamma(\widetilde{T}_2) \qquad \gamma(?) = \textsc{Type}$$

using the following operator:

$$\cdot \mathrel{\widehat{\to}} \cdot : \mathcal{P}(\textsc{Type}) \times \mathcal{P}(\textsc{Type}) \to \mathcal{P}(\textsc{Type})$$

$$\widehat{T}_1 \mathrel{\widehat{\to}} \widehat{T}_2 = \{T_1 \to T_2 : T_1 \in \widehat{T}_1, T_2 \in \widehat{T}_2\}$$

As an example, $\gamma(? \to \mathsf{Bool}) = \{T \to \mathsf{Bool} : T \in \textsc{Type}\}$. Next, given this interpretation of gradual types as sets of static types, we will lift predicates and functions on static types to predicates and functions on gradual types, which will allow us to lift our entire type system.

Lifting Predicates

Say we have a predicate or relation $P \subseteq \textsc{Type} \times \textsc{Type}$. An example would be equality—that is, $P(T_1, T_2) \iff T_1 = T_2$. As mentioned before, the general theme of gradual typing is optimistic uncertainty, so we would like to lift this predicate to a new predicate $\widetilde{P} \subseteq \textsc{GType} \times \textsc{GType}$ that, in a way, gives its parameters the benefit of the doubt. In other words, if $\widetilde{T}_1$ and $\widetilde{T}_2$ are certain, static types, then $\widetilde{P}(\widetilde{T}_1, \widetilde{T}_2)$ should reflect P itself. But if one or the other is an uncertain gradual type, then $\widetilde{P}(\widetilde{T}_1, \widetilde{T}_2)$ should optimistically hold true if there is a possibility that $\widetilde{T}_1$ and $\widetilde{T}_2$ could be compatible under P.

We formalize this notion of optimism as follows. First, we define

$$\widehat{P} \subseteq \mathcal{P}(\textsc{Type}) \times \mathcal{P}(\textsc{Type})$$

$$\widehat{P}(\widehat{T}_1, \widehat{T}_2) \quad \text{iff} \quad \exists (T_1, T_2) \in \widehat{T}_1 \times \widehat{T}_2 \quad \text{s.t.} \quad P(T_1, T_2)$$

which we call the *collecting lifting*. Then we define the gradual lifting

$$\widetilde{P} \subseteq \text{G}\textsc{Type} \times \text{G}\textsc{Type}$$

$$\widetilde{P}(\widetilde{T}_1, \widetilde{T}_2) \quad \text{iff} \quad \widehat{P}(\gamma(\widetilde{T}_1), \gamma(\widetilde{T}_2))$$

using our concretization function γ defined earlier.

Returning to our example of equality, if we let $\sim$ be the gradual lifting of $=$, then we find that the lifting is still reflexive and symmetric, but no longer transitive. For example, $\mathsf{Int} \to ? \sim \mathsf{Int} \to \mathsf{Int}$, but also $\mathsf{Int} \to ? \sim \mathsf{Int} \to \mathsf{Bool}$. The intuitive explanation for this is that gradual typing is intrinsically *short-sighted*: it checks whether something immediately makes sense (for instance, it is not true that $\mathsf{Int} \to ? \sim \mathsf{Bool} \to ?$), but is not global in scope.

Abstraction

In order to lift predicates it is sufficient to define a concretization function γ and use collecting semantics. However, when we next try to lift functions, we will need one extra piece. Specifically, we require a function $\alpha : \mathcal{P}^{+}(\textsc{Type}) \to \text{G}\textsc{Type}$ such that the following two properties hold:

- (soundness) If $\widehat{T}$ is not empty, then $\widehat{T} \subseteq \gamma(\alpha(\widehat{T}))$.
- (optimality) If $\widehat{T}$ is not empty and $\widehat{T} \subseteq \gamma(\widetilde{T})$ then $\gamma(\alpha(\widehat{T})) \subseteq \gamma(\widetilde{T})$.

It can be shown that this uniquely characterizes α:

$$\begin{aligned}
\alpha(\{\mathsf{Int}\}) &= \mathsf{Int} \\
\alpha(\{\mathsf{Bool}\}) &= \mathsf{Bool} \\
\alpha(\{\overline{T_{i1} \to T_{i2}}\}) &= \alpha(\{\overline{T_{i1}}\}) \to \alpha(\{\overline{T_{i2}}\}) \\
\alpha(\widehat{T}) &= ? \quad \text{otherwise}
\end{aligned}$$

The notation $\{\overline{T_i}\}$ here refers to a set of types indexed by i. Since this α satisfies the two properties listed above, it is said to form a *Galois connection* with γ.

Lifting Functions

Armed with this abstraction function, we can now lift functions on $\textsc{Type}$. We start by assuming a (partial) function $F : \textsc{Type} \times \textsc{Type} \rightharpoonup \textsc{Type}$; we define the collecting lifting of F to be

$$\begin{aligned}
\widehat{F} &: \mathcal{P}(\textsc{Type}) \times \mathcal{P}(\textsc{Type}) \to \mathcal{P}(\textsc{Type}) \\
\widehat{F}(\widehat{T}_1, \widehat{T}_2) &= \{F(T_1, T_2) : (T_1, T_2) \in \widehat{T}_1 \times \widehat{T}_2\}
\end{aligned}$$

and the gradual lifting to be

$$\widetilde{F} : \text{GT}\textsc{ype} \times \text{GT}\textsc{ype} \rightharpoonup \text{GT}\textsc{ype}$$

$$\widetilde{F}(\widetilde{T}_1, \widetilde{T}_2) = \alpha(\widehat{F}(\gamma(\widetilde{T}_1), \gamma(\widetilde{T}_2)))$$

and likewise for functions of arity greater than 2.

At this point we are ready to fully lift the typing rules from Figure 4 to work on our gradual language with GTERM and GTYPE. We have already lifted = to ∼. The only remaining tasks are to lift the functions dom, cod, and equate to $\widetilde{\text{dom}}$, $\widetilde{\text{cod}}$, and ⊓ respectively. It can be shown that the liftings of the first two functions are as follows:

$$\widetilde{\text{dom}} : \text{GT}\textsc{ype} \rightharpoonup \text{GT}\textsc{ype} \qquad \widetilde{\text{cod}} : \text{GT}\textsc{ype} \rightharpoonup \text{GT}\textsc{ype}$$

$$\widetilde{\text{dom}}(\widetilde{T}_1 \to \widetilde{T}_2) = \widetilde{T}_1 \qquad \widetilde{\text{cod}}(\widetilde{T}_1 \to \widetilde{T}_2) = \widetilde{T}_2$$

$$\widetilde{\text{dom}}(?) = ? \qquad \widetilde{\text{cod}}(?) = ?$$

Finally, the lifting of equate to ⊓ looks like this:

$$\mathsf{Int} \sqcap \mathsf{Int} = \mathsf{Int} \qquad \mathsf{Bool} \sqcap \mathsf{Bool} = \mathsf{Bool} \qquad \widetilde{T} \sqcap ? = ? \sqcap \widetilde{T} = \widetilde{T}$$

$$(\widetilde{T}_{11} \to \widetilde{T}_{12}) \sqcap (\widetilde{T}_{21} \to \widetilde{T}_{22}) = (\widetilde{T}_{11} \sqcap \widetilde{T}_{21}) \to (\widetilde{T}_{12} \sqcap \widetilde{T}_{22})$$

The complete static portion of our gradual type system is shown in Figure 5.

$$\widetilde{T}\text{X}\,\frac{x : \widetilde{T} \in \Gamma}{\Gamma \vdash x : \widetilde{T}} \qquad \widetilde{T}\text{N}\,\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \widetilde{T}\text{B}\,\frac{}{\Gamma \vdash b : \mathsf{Bool}}$$

$$\widetilde{T}\text{APP}\,\frac{\Gamma \vdash \widetilde{t}_1 : \widetilde{T}_1 \qquad \Gamma \vdash \widetilde{t}_2 : \widetilde{T}_2 \qquad \widetilde{T}_2 \sim \widetilde{\text{dom}}(\widetilde{T}_1)}{\Gamma \vdash \widetilde{t}_1\, \widetilde{t}_2 : \widetilde{\text{cod}}(\widetilde{T}_1)}$$

$$\widetilde{T}+\,\frac{\Gamma \vdash \widetilde{t}_1 : \widetilde{T}_1 \qquad \Gamma \vdash \widetilde{t}_2 : \widetilde{T}_2 \qquad \widetilde{T}_1 \sim \mathsf{Int} \qquad \widetilde{T}_2 \sim \mathsf{Int}}{\Gamma \vdash \widetilde{t}_1 + \widetilde{t}_2 : \mathsf{Int}}$$

$$\widetilde{T}\text{IF}\,\frac{\Gamma \vdash \widetilde{t}_1 : \widetilde{T}_1 \qquad \Gamma \vdash \widetilde{t}_2 : \widetilde{T}_2 \qquad \Gamma \vdash \widetilde{t}_3 : \widetilde{T}_3 \qquad \widetilde{T}_1 \sim \mathsf{Bool}}{\Gamma \vdash \mathsf{if}\ \widetilde{t}_1\ \mathsf{then}\ \widetilde{t}_2\ \mathsf{else}\ \widetilde{t}_3 : \widetilde{T}_2 \sqcap \widetilde{T}_3}$$

$$\widetilde{T}\lambda\,\frac{\Gamma, x : \widetilde{T}_1 \vdash \widetilde{t} : \widetilde{T}_2}{\Gamma \vdash (\lambda x : \widetilde{T}_1.\widetilde{t}) : \widetilde{T}_1 \rightarrow \widetilde{T}_2} \qquad \widetilde{T}{::}\,\frac{\Gamma \vdash \widetilde{t} : \widetilde{T} \qquad \widetilde{T} \sim \widetilde{T}_1}{\Gamma \vdash (\widetilde{t} :: \widetilde{T}_1) : \widetilde{T}_1}$$

Figure 5. Typing rules for our gradually-typed lambda calculus.

Discussion

The preceding sections lay out the background material for gradual program analysis. This final section discusses the research itself at a high level. Some discussion has already appeared in an extended abstract for the SPLASH 2019 undergraduate student research competition (Estep, 2019) and at the Workshop on Gradual Typing at the POPL 2020 conference (Estep et al., 2020).

Current Research

Reverse-engineered Infer. While the Facebook Infer tool advertises Eradicate as its primary null-safety checker, it also includes a few other checkers for null-pointer bugs. The most current checker is activated via the command-line parameter `--nullsafe`. Before starting on general gradual program analysis, we worked through the code of the `--nullsafe` checker to understand how it works, since no documentation previously existed for that checker. The checker's design informed the direction of the rest of the project.

Lifting semilattices. AGT can be applied fairly directly to a simple null-pointer analysis with a lattice of two elements by adjoining a single element ?. To construct a framework that could be applied to other static analyses, we designed a general way to lift a semilattice L to create a larger structure $\widetilde{L}$. Then using the AGT framework to lift predicates and functions, it can be shown that this larger structure $\widetilde{L}$ actually forms a semilattice structure under the lifted join operation $\widetilde{\sqcup}$, but of course not under the lifted order relation.

Lifting flow functions. In gradual typing, judgments are made from inference rules, which are simply transferred syntactically to the new system once all the sets, predicates, and functions have been lifted. In contrast, our dataflow analysis framework defines an explicit flow function. We initially tried lifting this flow function just using the AGT way to lift functions, which worked, but felt insufficiently elegant. Later, we reformalized the analysis so that the flow function is parametric in the set of lattice elements, so that it does not need to be lifted at all.

Static analysis warnings. By combining the lifted semilattice ordering relation with a safety function that is parametric in the set of semilattice elements just like the flow function, we provided a way to automatically decide when to give static warnings after computing the analysis fixpoint with the lifted semilattice.

Simple runtime checks. A fair amount of literature in gradual typing has been devoted to performing runtime checks for the types that have been only partially verified. This is nontrivial, because in general, it is undecidable to determine whether a value is of a particular type, such as if that type is a function type. Our framework avoids this complexity by assuming that it is efficient to check membership of any abstract lattice element, and thus provides a simple way to perform runtime checks after the static part of the gradual analysis is completed.

Null-pointer implementation. We implemented the null-pointer case of gradual analysis as a new checker in the Facebook Infer tool, then evaluated it by using it to analyze 15 of the 18 repositories analyzed in the evaluation of NULLAWAY (Banerjee et al., 2019). We compared the set of warnings against the warnings produced by Eradicate and `--nullsafe`, and our prototype seems to give fewer false positives than those existing tools. However, it is unclear exactly what this means, as none of the tools gave a significant number of true positives for any of those repositories, so it is possible that our tool simply produces fewer warnings in general, which would not strictly be a benefit.

Future Research

Proofs. From evidence we have so far, we believe that our framework satisfies certain properties, such as the so-called "gradual guarantee" (Siek et al., 2015, p. 1). However, we have yet to provide static proofs for these claims.

Better experiments. As mentioned above, the empirical evaluation we have performed so far is insufficient to demonstrate the practicality of our approach. It would be desirable to use our prototype to analyze repositories with actual errors in them, to see which it catches and which it does not.

Composable analysis component. Some recent work in abstract interpretation (Keidel & Erdweg, 2019) shows how to decompose a static analysis into a composition of reusable components. It would be interesting to see whether our framework can be formalized as a component in that system. This may also cut down on the proofs that we would need to write.

General runtime checks. To allow our approach to generalize to as many analyses as possible, it would be desirable to use the extensive literature from gradual typing to provide a way to perform runtime checks for arbitrary values, not just those that can be easily checked against our semilattice.

References

Aldrich, J. (2019). 17-355/17-665/17-819 program analysis. Retrieved from https://www.cs.cmu.edu/~aldrich/courses/17-355-19sp/

Ayewah, N. & Pugh, W. (2010). Null dereference analysis in practice. *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 65–72. doi:10.1145/1806672.1806686

Bader, J., Aldrich, J., & Tanter, É. (2018). Gradual program verification. *International Conference on Verification, Model Checking, and Abstract Interpretation*, 25–46. doi:10.1007/978-3-319-73721-8_2

Banerjee, S., Clapp, L., & Sridharan, M. (2019). NullAway: Practical type-based null safety for Java. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 740–750. doi:10.1145/3338906.3338919

Bierman, G. M., Parkinson, M., & Pitts, A. (2003). MJ: An imperative core calculus for Java and Java with effects. Retrieved from https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-563.html

Brotherston, D., Dietl, W., & Lhoták, O. (2017). Granullar: Gradual nullable types for Java. *Proceedings of the 26th International Conference on Compiler Construction*, 87–97. doi:10.1145/3033019.3033032

Castagna, G. & Lanvin, V. (2017). Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages*, *1*(41), 1–28. doi:10.1145/3110285

Cousot, P. & Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 238–252. doi:10.1145/512950.512973

Davey, B. A. & Priestley, H. A. (2002). *Introduction to lattices and order*. Cambridge, England: Cambridge University Press.

Estep, S. (2019). Gradual program analysis. *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 52–53. doi:10.1145/3359061.3361082

Estep, S., Wise, J., Aldrich, J., Tanter, É., Bader, J., & Sunshine, J. (2020). Gradual program analysis. Retrieved from https://popl20.sigplan.org/details/wgt-2020-papers/9/Gradual-Program-Analysis

Fähndrich, M. & Leino, K. R. M. (2003). Declaring and checking non-null types in an object-oriented language. *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, 302–312. doi:10.1145/949305.949332

Garcia, R., Clark, A. M., & Tanter, É. (2016). Abstracting gradual typing. *ACM SIGPLAN Notices*, *51*(1), 429–442. doi:10.1145/2914770.2837670

Girard, J.-Y. (1971). Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. *Proceedings of the Second Scandinavian Logic Symposium*, *63*, 63–92. doi:10.1016/S0049-237X(08)70843-7

Hoare, T. (2009). Null references: The billion dollar mistake. Retrieved from https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

Hovemeyer, D. & Pugh, W. (2007). Finding more null pointer bugs, but not too many. *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 9–14. doi:10.1145/1251535.1251537

Hovemeyer, D., Spacco, J., & Pugh, W. (2005). Evaluating and tuning a static analysis to find null pointer bugs. *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 13–19. doi:10.1145/1108792.1108798

Hubert, L., Jensen, T., & Pichardie, D. (2008). Semantic foundations and inference of non-null annotations. *International Conference on Formal Methods for Open Object-Based Distributed Systems*, 132–149. doi:10.1007/978-3-540-68863-1_9

Keidel, S. & Erdweg, S. (2019). Sound and reusable components for abstract interpretation. *Proceedings of the ACM on Programming Languages*, *3*(176), 1–28. doi:10.1145/3360602

Kildall, G. A. (1973). A unified approach to global program optimization. *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 194–206. doi:10.1145/512927.512945

Male, C., Pearce, D. J., Potanin, A., & Dymnikov, C. (2008). Java bytecode verification for @NonNull types. *International Conference on Compiler Construction*, 229–244. doi:10.1007/978-3-540-78791-4_16

Muehlboeck, F. & Tate, R. (2017). Sound gradual typing is nominally alive and well. *Proceedings of the ACM on Programming Languages*, *1*(56), 1–30. doi:10.1145/3133880

Pierce, B. C. (2002). *Types and programming languages*. Cambridge, MA: The MIT Press.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, *74*(2), 358–366. doi:10.2307/1990888

Siek, J. G. & Taha, W. (2006). Gradual typing for functional languages. *Scheme and Functional Programming Workshop*, *6*, 81–92. Retrieved from http://schemeworkshop.org/2006/13-siek.pdf

Siek, J. G., Vitousek, M. M., Cimini, M., & Boyland, J. T. (2015). Refined criteria for gradual typing. *1st Summit on Advances in Programming Languages (SNAPL 2015)*, *32*, 274–293. doi:10.4230/LIPIcs.SNAPL.2015.274

Spoto, F. (2011). Precise null-pointer analysis. *Software & Systems Modeling*, *10*(2), 219–252. doi:10.1007/s10270-009-0132-5

Spoto, F. (2016). The Julia static analyzer for Java. *International Static Analysis Symposium*, 39–57. doi:10.1007/978-3-662-53413-7_3

Takikawa, A., Feltey, D., Greenman, B., New, M. S., Vitek, J., & Felleisen, M. (2016). Is sound gradual typing dead? *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 456–468. doi:10.1145/2837614.2837630

Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, *5*(2), 285–309. doi:10.2140/pjm.1955.5.285

Toro, M. & Tanter, É. (2017). A gradual interpretation of union types. *International Static Analysis Symposium*, 382–404. doi:10.1007/978-3-319-66706-5_19

Trunov, A. (2016). Lambda calculus reduction examples. Retrieved from https://cs.stackexchange.com/a/52943

Van Horn, D. & Might, M. (2010). Abstracting abstract machines. *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, 51–62. doi:10.1145/1863543.1863553

Vitousek, M. M., Kent, A. M., Siek, J. G., & Baker, J. (2014). Design and evaluation of gradual typing for python. *Proceedings of the 10th ACM Symposium on Dynamic languages*, 45–56. doi:10.1145/2661088.2661101