Pratical Types for Python
Final Report

Daniel Randall

Contents

1 Introduction

1.1 Motivation

High level, dynamically typed languages such as Python have been gaining popularity in recent years. One reason for this is a high level of productivity involved in using such languages, partly due to the programmer being free from having to declare the types of the variables used. However the benefits gained from this design decision are at the expense of the security provided by a statically typed language. Type errors which are easily caught by a compiler for a statically typed language, such as Java or C++, can be be left unnoticed in Python source code into the release stage of a product where its execution may prove fatal. Specifying types is regarded as superfluous by many in the Python community. While there was has been significant talk of adding optional static typing to Python [20] [21] [22], the backlash from such discussions is strong enough such that it is very unlikely we shall see this addition to the Python language anytime soon.
Tools attempting to provide the benefits of a statically typed language to Python do exist, such as Pylint[1] and PyChecker[2], however their usability is hampered by the false positives returned and the amount of configuration needed.
This project aims to design and implement a type checker for Python which returns no false positives. The basis for our implementation is a field called success typings. We harness this technique in order to provide a over-approximation of how a function is intended to be used. This allows us to determine that a violation of a function's contract is guaranteed to be a legitimate programming error. We describe success typing as well as the general theory behind type systems in section 2. We then examine similar products and techniques and show the limitations in their application.

Using the methods and techniques we described, we begin to define our solution and evaluate the best way to solve the problems we shall face. Once we decide upon the best methods we then set about evaluating the best 3rd party software we can use to achieve these tasks and why we should not implement our own solutions.

In section 3 we describe what we have done and begin to evaluate it. We evaluate our implementation in three ways: speed, ease of use and the

[1] `http://www.pylint.org/`
[2] `http://pychecker.sourceforge.net/`

number of false positives/negatives it reports. We benchmark our solution against an existing file sharing platform before we allow a number of users to test drive our work. We see in which environments our work excels and in which it fails to make an impression.

1.2 Objectives

The objectives for this project are as follows:

- **No false positives** - Our solution should not report any errors which are then found not to be genuine bugs in the program. A 'genuine' bug can be defined as one which is guaranteed to cause a run-time crash when executed.
- **Static** - Our solution should not require the user to run the program with a test suite in order to use the tool.
- **Out-of-the-box** - The user should not need to specify the way the tool should run, such as command line arguments. The user also should not have to modify their program in any way in order to successfully use our solution.
- **Fast** - Our system will ideally run at a high speed, similar to alternative type checkers for Python.

2 Types in Programming Languages

Type systems are created by the writers of programming languages and are used to prevent improper use of program operations. To quote Benjamin Pierce [14]:

> *"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."*

In order for this to hold in programming languages, the phrases (e.g. variables, expressions, method calls) are labelled in order to express the classification. This labelling can be achieved in a number of ways and allows the language to prevent operations being used with terms with classification they

were not designed to be used with.
A type system can used for detecting errors, documentation, language safety, abstraction and efficiency [14]. This report focuses on error detection.

2.1 Static and Dynamic Type Systems

Static type systems require programs to be written such that all variables are labelled with a type. Static type systems are conservative, meaning that they can reject programs which may never cause an error at run-time. Consider the following example for the statically typed C++ language:

```
if (true) {
  int x = 5;
} else {
  int x = 5 + "Hello world";
}
```

The C++ compiler will not accept this code despite the fact it will, intuitively, behave well at run-time. This is because the type system is able to determine the *absence* of type errors but not the *presence*.
Dynamic type systems do not require a label for variables until run-time where "run-time type tags are used to distinguish different kinds of structures in the heap." [14] Dynamic type systems do not suffer from the same conservative nature as statically typed languages. For instance, the equivalent program to the previous C++ example in Python is:

```
if (True):
  x = 5;
else:
  x = 5 + "Hello world"
```

This program will never return an error, despite the fact 5 + "Hello world" is an illegal operation. Using a dynamic type system, only programs about to go wrong during run-time are rejected. The obvious downside to this is that errors which would be caught easily by a static type system may go undetected by a dynamic type system and wreck havoc later in the production cycle.

2.2 Control Flow

The control flow of an application is the route taken when the program is executed or the order in which individual statements are executed. Consider the following program:

```
1. x = 5
2. x = x + 1
3. print(x)
```

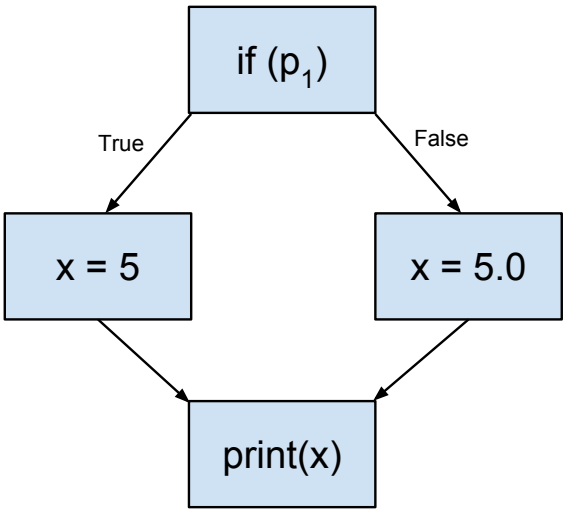
The route taken by this program is simply $1 \rightarrow 2 \rightarrow 3$. However with the introduction of conditional statements such as *if*, *for* and *while* the path taken in the program is dynamically decided by the evaluation of a condition. While the introduction of these statements undoubtedly provide more power to an engineer, it becomes far more difficult to statically predict the route taken. An example of this is as follows:

```
1. if (p1):
2.     x = 5
3. else:
4.     x = 5.0
5. print(x)
```

The flow of the above program can be described as $1 \rightarrow (2, 3 \rightarrow 4) \rightarrow 5$ where (x, y) represents a choice between x and y.

2.2.1 Control Flow Graph

A control flow graph (CFG) is a way of depicting the control flow of a program. A CFG provides a graphical way of representing the control flow as well as providing a meaningful way of storing the information in a program. CFGs are commonly used in compilers in order to reduce program size and increase performance by eliminating dead code.
Using the last example of *if* statement in the previous section we get:



2.2.2 Flow-sensitivity

A flow insensitive analysis takes no account of the order of execution [13]. That is, each occurrence of a variable typically is represented by the same set of possible types no matter where it appears in the program. Consider the following example:

```
if (y):
  x = 5    // S1
else:
x = 5.0  // S2
f(x)           // S3
```

A flow-sensitive algorithm would infer the type of x at S1 to be $\{int\}$ and $\{float\}$ at S2, while a flow-insensitive algorithm would infer the type of x to be $\{int, float\}$ at S1, S2 and S3.
Another case in which ignoring the flow of execution can change the types inferred is represented by the following trivial block of code:

```
x = 5
x = 5.0
```

A flow-insensitive algorithm would infer the type of x, for any future occurrences of x, to be the set $\{int, float\}$. A flow-sensitive algorithm is able to re-write the constraint graph to allow a more accurate representation of the flow, allowing it to infer more accurately the type of x as the set $\{float\}$.
A flow-sensitive algorithm is clearly the more accurate approach but requires a comparatively complex algorithm which consumes more information than a flow-insensitive approach.
Ryder [17] suggests that the difference between a flow-sensitive and flow-insensitive approach is minimal in regards to object-oriented programs due to the size of methods being small. While Python can be, and is, used for object-oriented programming, a significant number of Python programs are written as scripts and so the benefits of a flow-sensitive algorithm can not ruled out in our case.

2.3 SSA Naming

Primarily used in compilers, Single Static Assignment (SSA) is used to acquire "unique names for distinct entities." What this means is that each variable should only be assigned to once. There are number of variants of SSA, each designed for a specialised purpose. In our case we are only required to use the basic version of SSA, 'vanilla' SSA.
Consider the following example taken from...:

```
x = 5
y = x + 1
x = 7
z = x + 1
```

At this time the code violates the principles of SSA since the variable x is assigned to twice. Translating this block into SSA form would give us the following:

```
x1 = 5
y1 = x1 + 1
x2 = 7
z1 = x2 + 1
```

It can be observed that after each incarnation the x the latest is then to replace all instances of x where x isn't being assigned a new value. This

can be seen in the assignments to y and z. Note that the names of, both, x and y need not be changed in this case since they are only assigned to once. However it often makes the implementation of SSA easier to do so regardless.

2.3.1 The ϕ function

In a sequential program the manner used to number new variables is quite intuitive, we simply increment the number appended the end of the variable name. However as we introduce branching into our programs this method alone breaks down. Consider the following program:

```
x = 5
if (y):
  x = 5
else:
  x = 6
z = x + 1
```

The *if* statement creates two branch which both assign a new value to the the variable x. Assigning new names to these variables is required by the rules of SSA, however it is unclear which name we are to use after the *if* and *else* when x is used in then assignment to the variable z.
To resolve this problem, we introduce the concept of the ϕ *function*. The ϕ function is used "to merge values from different incoming paths, at control-flow merge points." Put simply, the result of the ϕ can be said to be *any* of the parameters given.

```
x1 = 5
if (y):
  x2 = 5.0
else:
  x3 = 6
x4 = φ(x2, x3)
z = x + 1
```

Since we only care about the possible types each variables, when applied in our application the ϕ function can viewed as a glorified union all possible types of all parameters. So in the above example we would have $x4 = \phi(x2, x3) = \{Float\} \cup \{Int\} = \{Float, Int\}$.
A similar, slightly more complex case arises for loops. In this case, we must

consider what the paths to the start of the loop as well as the end. Consider the following trivial example:

```
x = 5
while (y):
  x = x + 1.0
z = x + 1
```

At the entry of the loop, x can come from the statement immediately outside of the loop as well as the end of the loop. The case for immediately outside of the loop is the same as in case for *if*. Using this information gives us two cases for ϕ function. The result program in SSA form is:

```
x1 = 5
while (y):
  x2 = φ(x1, x3)
  x3 = x2 + 1.0
x4 = φ(x1, x3)
z = x4 + 1
```

The ϕ function can also be useful for a slightly different way in order to simplify inferring the possible return types of a function. Consider the following Python function:

```
def f():
        if (p1):
          return 5
        if (p2):
          return "Hi"
        return 5.0
```

Here we can see that the function f has a number of different *return* statements which can return a number of different types. In order to infer the possible types that a function can return we need to look at the values created by each return statement. We can use the ϕ function in order to simplify this task. To do this we modify the function like so:

```
def f():
        if (p1):
          r1 = 5
        if (p2):
```

```
  r2 = "Hi"
r3 = 5.0
r4 = φ(r1, r2, r3)
```

The usefulness of SSA naming is immediately obvious. We need not track how the types of a variable may change over time. We may treat each new naming as a completely distinct variable. Managing the control-flow also becomes a much simpler task as we are easily able to express the fact that a variable can possibly be a number of different types depending on the path taken through the program.

2.4 Dead Code Elimination

Dead Code Elimination, also known as Tree shaking, is the process of detecting that section of code is dead (unreachable) and completely eliminating it from the program. Code may be unreachable as it is placed after a *break*, *continue* or *return* statement or if the code has the effect of only modifying variables which will never be used again, 'dead' variables.The advantages of this is that we need not type check it, thus improving the speed of our program. An example of this is: (adapted from Wikipedia)

```
def foo ():
    a = 24
    b = 25   # Assignment to dead variable
    c = a + 2
    return c
    b = 24    # Unreachable code
    return 0
```

Whether flagging an error on unreachable code is considered a false-positive is one for the philosophers. Unfortunately there are no out-of-the-box solutions to this problem for Python programs.

2.5 Constraints and Constraint Solvers

A constraint satisfaction problem (CSP) can be broken down into one of three categories, boolean satisfiability problem (SAT), the Satisfiability Modulo Theories (SMT) and answer set programming (ASP). A CSP is typically

described $\langle X, D, C \rangle$ where X represents a set of variables, D is a set of corresponding domains and C is a set of constraints which must be followed by any solution. Every variable must have a *domain*. A domain is a specific area over which the variables can take a form. An example domain is the set of integers, $\mathbb{Z}$.
A solution to a CSP is a mapping from the given variables to a value in the domain which satisfies the constraints provided.
Constraints are frequently solve using satisfiability modulo theories (SMT) solvers
The most popular constraint solving software for Python is *python-constraint*.

3 Type Inference

The main problem which is required to be solved by this project is how to statically infer the types in Python without executing the program.
In this section we look at a number of the existing techniques to achieve this while keeping in mind the objectives described at the beginning of this report.

3.1 Hindley-Milner

The Hindley-Milner algorithm was developed by Robin Milner based on the work of J. Roger Hindley. Hindley described the *principal type schema* [15], "which is the most general polymorphic type of an expression, and showed that if a combinatorial term has a type, then it has a principal type." [3] Milner's contribution was an extension to Hindley's *principal type schema* which included the "notion of generic and non-generic type variables, which is essential for handling declarations of polymorphic functions." [3] Milner's work culminated in the type inference for the ML language [12].
The Hindley-Milner algorithm uses the operations performed on expressions to generate constraints on the types of those expressions. This is done by annotating the nodes of an expression tree with the constraints in a bottom up fashion[3].

[3]The Hindley-Milner algorithm can be implemented top-down, called the $\mathcal{M}$ algorithm, or bottom-up, called $\mathcal{W}$ [8]. However, the bottom-up version appears to much more common.

The constraint is solved to infer a type for a term using a *unification* algorithm. Bugs can be detected by determining whether there are inconsistencies in the set of constraints.

Limitations
The technique does not allow polymorphic argument to be of a different type in different locations. This is because unification requires there to be a single type for all appearances of a variable. For this reason Hindley-Milner is unable to infer a type for programs such as the following:

```
if (y):
  x = 5
else:
  x = 5.0
```

We have already seen examples such as this in previous sections, and so the Hindley-Milner approach is not appropriate in our case.

3.1.1 Pyty

Developed by Jeff Ruberg [16], Pyty is a bug checker which analyses annotated source code to detect errors related to the misuse of types. Pyty employs Hindley-Milner as its type inferencing algorithm.

Limitations
Pyty requires the need for annotations to the source code in a specific format. This is quite tedious and prevents from being an 'out-of-the-box' solution to our problem.

3.2 Cartesian Product Algorithm

The Cartesian Product Algorithm (CPA) was originally conceived by Agesen for the language Self [1] and was later adapted by Michael Salib for Python [18].
The Cartesian Product Algorithm works by modelling data flow as opposed to the constraint and unification method adopted by the Hindley-Milner algorithm. This is done by building a set of possible types an expression can take. Consider the following code:

```
  if (y):
x = 5
  else:
    x = 5.0
```

The inferred types for x would be the set $\{int, float\}$, where the possible *int* type is found in the *then* branch and the *float* type is found in the *else* branch. The types from each branch are added to the set of possible types for x. The CPA algorithm guarantees that the type of the variable is contained within the set.
In order to infer the types handed to a function by taking the Cartesian product of the set of possible types for all given arguments. Consider the following code:

```
if (y):
  x = 5
  z = "Hello world"
else:
  x = 5.0
  z = True
f(x, z)
```

The set of inferred types for x is, as before, $\{int, float\}$ and the set for z is $\{str, bool\}$. The possible types of the inputs to the function f is: $\{int, float\} \times \{str, bool\} = \{(int, str), (int, bool), (float, str), (float, bool)\}$, where each tuple represents the possible types of the variables given to f.

Limitations
The size of the Cartesian products grows exponentially with the number of arguments given to the function call. However the size is bounded by the number of types available.
The difficulty of implementation is greater than that of the Hindley-Milner algorithm which is known as being a fairly simple algorithm to code [10].

3.2.1 Starkiller

Starkiller was designed by Michael Salib as the type inference method of a Python-to-C++ compiler [18]. The algorithm used was based loosely on Age-

sen's Cartesian Product Algorithm and achieves a complete type inference of Python source code.

Limitations
Unable to infer types for the dynamic constructs such as *eval*, or *exec*. Exceptions are unsupported. The implementation is flow-insensitive and so is not as precise as it could be.

3.2.2 Shed Skin

Shed Skin is a Python-to-C++ compiler developed by Mark Dufour which boasts up to a 40-fold performance increase over Psyco [5]. Shedskin employs the Cartesian Product Algorithm alongside iterative class-splitting. Class-splitting is...

Limitations
The programs consumed by Shed Skin are required to be implicitly statically typed. Meaning the type of a variable can not dynamically change. Shed Skin also does not support a number of Python features such as *eval* and *isinstance*.

3.2.3 Localized Type Inference of Atomic Types in Python

Types for local, atomic variables are inferred in an attempt to improve the performance of Python.The type inference is done by intercepting bytecode from the compiler and modifying it by injecting additional bytecode relating the the types of variables. The idea was the processor could utilise this information to improve performance. Cannon's work differs to Psyco in that the work is done inside of Python's compiler, rather than the interpreter.
The algorithm described is capable of handling integrals, *float*, *complex*, *basestring*, *list*, *tuple* and *dict*.

Limitations
The limitations involve only in inferring types for local variables, meaning that the implementation is largely useless when used on programs designed with Object Oriented Programming (OOP) in mind, as noted by Cannon. While not a real limitation, Cannon's solution intercepts the compiler in order

to improve performance. This is not our aim and we need not complicate things by working with bytecode.

3.3 Success Typings

The phrase 'success typings' was coined by Lindhal and Sagonas in the 2006 paper *Practical Type Inference Based in Success Typings* [11]. The aim of a success typing is to fully describe all possible intended uses of a function. This description is given as type signature for a function f: $(\bar{\alpha}) \rightarrow \beta$, where $(\bar{\alpha})$ refers to the type of the function parameters, and $\bar{\alpha}$ is a shorthand for $\alpha_1, \alpha_2, ...\alpha_n$, and β is the type of the return value. Both types are the 'largest' possible types, i.e. subtypes are acceptable. For instance, consider the following function as described in the Lindhal et al. paper for the functional language *Erlang*:

```
and(true, true) → true;
and(false, _) → false;
and(_, false) → false;
```

where the symbol '_' represents a *don't care* option for pattern matching, meaning it will match any value for the corresponding parameter.
An acceptable success typing for this function, and, indeed, for any function with two arguments, would be: $(any(), any()) \rightarrow any()$ where $any()$ denotes the set of all Erlang types. Such a typing would raise no warnings about the any use of the function and so can be used when no typing information can be inferred about a function. However, a more useful typing for the function in question is $(any(), any()) \rightarrow bool()$ where the return type of the function is restricted to all subtypes of $bool()$. Since we have the *don't cares* any parameter paired with an instance of $false$, such as $(42, false)$, is a valid use of the function. This optimistic approach avoids any possible false positives from any warnings from reasoning about the typing and so will never reject a well-formed program. This typing does allow for warnings to be issued as a result of type clashes in matching a value which is not a subtype of $bool()$ with the result of the function.
A success typing is inferred by building constraints by traversing the code and then solving them.
Constraints are built by providing a list of derivation rule. Assume the following definitions: e is any expression which can be built in a language,

τ is a type, such as a boolean or integer,
A represents an environment with bindings of variables of the form $\{\ldots, x \mapsto \tau_x, \ldots\}$,
C represents nested conjunctions and disjunction of subtype constraints:

$$C ::= (T_1 \subseteq T_2) \mid (C_1 \wedge \ldots \wedge C_n) \mid (C_1 \vee \ldots \vee C_n)$$

$$T ::= none() \mid any() \mid V \mid c(T_1, \ldots, T_n) \mid (T_1, \ldots, T_n) \to T' \mid T_1 \cup T_2 \mid TwhenC \mid P$$

$$V ::= \alpha, \beta, \tau$$

$$P ::= integer() \mid float() \mid atom() \mid pid()|42|foo| \ldots$$

and the judgement $A \vdash e : \tau, C$ should be read as "given the environment A the expression e has type $Sol(\tau)$ whenever Sol is a solution to the constraints in C".
Then one such derivation rule is the rule for a struct:

$$\text{STRUCT}\frac{A \vdash e_1 : \tau_1, C_1 \ldots e_n : \tau_n, C_n}{A \vdash c(e_1, \ldots, e_n) : c(\tau_1, \ldots, \tau_n), C_1 \wedge \ldots \wedge C_n}$$

The struct rule states that given a number of elements, each with its own type, then they can be grouped into a tuple structure in the environment with each individual element retaining their type. The constraints for each element are added to the environment in a conjunction.
Sol is a mapping from type expressions and type variables to concrete types. Sol is a solution to a constraint set C if:

$$Sol \models T_1 \subseteq T_2 \iff none() \subset Sol(T_1) \subseteq Sol(T_2)$$

$$Sol \models C_1 \wedge C_2 \iff Sol \models C_1, Sol \models C_2$$

$$Sol \models C_1 \vee C_2 \iff \begin{cases} Sol_1 \models C_1, Sol_2 \models C_2, \\ Sol = Sol_1 \sqcup Sol_2 \end{cases}$$

where $Sol_1 \sqcup Sol_2$ denotes the point-wise least upper bound of the solutions. Each case represents a different type of constraint which we may encounter (subtype, conjunction or disjunction). The subtype case states that a solution satisfies a subtype constraint if the mapping satisfies the subtype constraint and neither of its constituents is *none()*. The conjunction case states that the solution must satisfy all conjunctive parts. The disjunction case that the solution is the point-wise least upper bound of all disjuncts.

3.4 Soft Typing

Cartwright and Fagan extended the Hindley-Milner $\mathcal{W}$ algorithm to introduce soft typing [4]. The aim of soft typing is not to reject a program for which static type checking fails but to "transform arbitrary programs to equivalent programs that type check." Soft typing works by inserting run-time checks, resulting from "narrowers," when a program fails to statically type a program, i.e. when unification in $\mathcal{W}$ fails. Narrowers are type casts which blindly convert from the current type to the destination type. Conversion errors are caught by the run-time checks.
Soft typing requires a program to be run in order to notify the programmer about errors. Our aim is to create a static debugger and so our aims are incompatible with soft typing.

3.5 Aggressive Type Inference

Aggressive type inference (ATI) is a technique developed by John Aycock [2] which follows the idea that:

> *"Giving people a dynamically-typed language does not mean that they write dynamically-typed programs."*

Aycock backs up this hypothesis by citing a study [23] which reveals that around 80% of operators in a set of *Icon* programs maintain the same type throughout their lifetime.
He exploits this by using a flow-insensitive method and does not use union types. Meaning the algorithm does not look through all routes to determine all possible types for a variable and only labels a variable with a single not type, not a union of multiple types. The algorithm works by iteratively analysing the code to infer the types of variables and by propagating the types on each iteration.

Limitations
The limitations of Aycock's approach are quite clear; the types involved in dynamic behaviour are not reliably extracted. How much of an issue this poses is up for debate; A study on this, more recent and relevant to our interests than the *Icon* analysis put forward by Aycock, has been conducted by Alex Holkner and James Harland [9]. This study details the evaluation of twenty four open source Python systems. Their results argue that dynamic

features are actually widely used. For instance, they find that all systems studied employ dynamic code execution. Holkner and Harland concede that there study is small in comparison to the amount of Python code available, however if their results are to be extrapolated then Aycock's assumption is not so reasonable.

3.6 RPython

RPython is an intermediate language, a subset of Python which is entirely statically typed, which acts as a link in the PyPy toolchain. PyPy's goal is to develop a Just-In-Time (JIT) compiler for Python. PyPy's interpreter is written in RPython which removes dynamic features in order to reduce of the complexity of type inference.

Limitations
The major limitation of PyPy is the use of RPython. One feature of RPython is that it does not allow variables to change their type.

3.7 Related Work

To the best of my knowledge, there is currently no debugger for Python which does not return any false positives. The most active debuggers at the time of writing are Pylint and and PyChecker.

3.7.1 Pylint

Pylint is an error checking tool for Python which statically analyses Python source code to look for errors, including type errors, and to assess the coding style. Pylint is a static analyser and so it looks for errors without importing/running the source code.

Limitations
Can often return a lot of false positives and offers a number of command-line options in an attempt to allow users to suppress them.

3.7.2 PyChecker

PyChecker is an bug checker for Python which checks for type errors among a host of other errors. PyChecker is similar to Pylint except it does not checker the coding style of the source program. PyChecker also differs in that it needs to runs the code in order to analyse it.

Limitations
PyChecker can return spurious errors and warnings. PyChecker imports the code it is analysing which can have undesirable side effects.

3.7.3 PySonar

PySonar was developed by Yin Wang while an intern at Google between 2009 and 2010 [24]. PySonar is a type inferencer and indexer, using abstract interpretation, intended for the internal use within Google's Grok project. PySonar claims to be able to resolve the names of 97% of the Python standard library.

Limitations
PySonar is focused on indexing the code it analyses and so it does not report any possible type errors. The program focuses mainly on a less dynamic subset of Python which is 'easier' to analyse (from Guido).

3.7.4 Pyntch

Created by Yusuke Shinyama, Pyntch uses type inference in Python in order to find bugs [19].
Project activity has ceased since 2009.

Limitations
Can return false positives.

3.7.5 Ruby

Ruby is a dynamically typed scripting language, similar to Python.
Furr et al. developed Diamondback Ruby (DRuby) [7] in order to discover type errors. DRuby works by using type annotations to library functions in order to generate constraints to infer the types for user defined functions.

DRuby is not sound as it accepts programs which are dynamically incorrect. DRuby also reports false positives.
An extension to Diamondback Ruby, $\mathcal{P}$Ruby, was created in order to handle difficult dynamic language constructs such as the *eval* method which converts a string into executable program code. [6] This is done by instrumenting the code such that, when the program is then run, all uses of the troublesome constructs are documented. This allows a *profile* to be built which fully describes how they were used. With this profile the dynamic code can be inserted into the program in place of the dynamic constructs. The modified program can then be statically analysed just like any other. Furr et al. use DRuby for the analysis.

4 Implementation

This project was built on top of the work by Edward K. Ream's *stc*. [4] The primary contribution which was left largely unchanged was the traversers which are used to navigate the the abstract syntax tree.

4.1 SSA Form

4.2 Type Inference

4.2.1 Inferring the Types of Simple Assignments

All changes to the types of variables involve assignment. For each assignment the variable on the left-hand side (LHS) assumes all of the types which the right-hand side (RHS) can possibly be at the time. The following shows a number of ways this can be done.

```
x = 5
x = "Hi"
x = f()
```

Assignments with a simple numeric or string value can be type inferred very simply and we can assign the variable a single possible type with complete

[4] `https://code.launchpad.net/~edreamleo/python-static-type-checking/trunk` Revision #448 was used for this project.

confidence. If the RHS is a function then we assign all of the types the function can possibly return.

4.2.2 Inferring the Types in List/Tuple Assignment

4.2.3 Inferring the Types in For/While Loops

A *while* loop in SSA form looks like:

```
x1 = 5
while (p):
  x2 = φ(x1, x3)
  y1 = x2
  x3 = "Hi"
x4 = φ(x1, x3)
y2 = φ(x1, None)
```

A problem raised by this is that the ϕ functions which exist at the beginning of the loop depend of types which can not be determined until the end of the loop body. In between then there may be other variables which use the result of the ϕ function in their own assignment, like $y1$ above. To overcome this we use a *Awaiting_Type* value which is assigned to any variable which is awaiting the type of another variable, the *waiter* along with the name of the variable they are waiting for, the *waitee*. In the case above we have $x2$ waiting for $x3$ and $y1$ waiting for $x2$. When a waitee is assigned its types then attempts are made to type all of that variable's waiters. If it is found they also need to wait for another variable then the same process occurs again otherwise they are successfully typed. Currently this approach collapses if there is a circular assignment involving a waiter and waitee, e.g. $x2 = x1 + 1$ where $x1$ which can cause the waiter of each to be each other. Currently if this circular assignment is detected then both variables are assigned the Any_Type.

4.2.4 Inferring the Possible Types of Function Arguments

There are number of ways we can go about inferring the types of function arguments. We could use either a *top-down* approach or a *bottom-up* approach. A top-down approach involves finding all of the calls to a function and determining the types the function accepts by the inferring the types of

values given. On the other hand a bottom-up approach involves determining the types of arguments solely by how they are used within the function body. Initially our approach was to use the bottom-up approach. It was felt that more to be gained using this approach since we would be able to detect errors caused by incorrect types being passed to a function. However it is important to note that useful information can be extracted from the calls. There is a possibility of using this information when we are unable to infer the types of the parameters using the function body in isolation.
To do this we will generate a number of constraints and attempt to solve them. In order to solve the constraints we will make use of a pre-existing constraint solver for Python, *python-constraint* [5]. When our generated constraints can not be solved then we resort to labelling the parameter has having any possible type (AnyType / Top). It is important to do this before we attempt to infer the return types of a function as it is likely that the possible return types will depend upon the types of the function arguments.

Constraint Generation
Consider the following two functions:

```
def f(x, y):
  return x + y

def g(x, y):
  if (p1):
    return x + y
```

Both functions apply the operations + operation to the parameters. However, due to the *if* statement the constraint this imposes on the types they can take is quite different. In function f the operation is unconditional and so we know that both parameters are required to be of a type (or a sub-type of a type) which can used in this operation. On the hand, in the function g this statement may not be executed in which case we can see that no type will fail. This leads us to the conclusion that we must take the context of the statement into account before we issue a constraint.

Constraint Solving
Our constraint solver returns all possible sets which satisfy the constraints

[5] `http://labix.org/python-constraint`

we feed into it. We are interested *all* of the possible types a parameter can legally take. Because of this we take the largest set returned for each parameter.

4.2.5 Inferring the Possible Return Types of a Function

To do this we rely on the ϕ function as described in the SSA section. We simply take the union of all of the variables passed. Consider the following, already transformed, example:

```
def f():
  if (p1):
    r1 = 5
  if (p2):
    r2 = "Hi"
  r3 = 5.0
  r4 = φ(r1, r2, r3)
```

The determined possible returns types are $r_1 \cup r_2 \cup r_3 = \{Int\} \cup \{String\} \cup \{Float\} = \{Int, Float, String\}$.
There is a the possibility of a relationship between the types of the parameters and the return types, e.g. a string is always returned if an integer is given. While this addition would improve the accuracy of our type inference, it would be complex to implement.

4.2.6 Inferring the Types Involving Recursive Functions

References

[1] Ole Agesen. Concrete type inference: Delivering object-oriented applications. Technical report, Stanford University, 1995.

[2] John Aycock. Aggressive type inference. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2000.

[3] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172, 1987.

[4] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI*, pages 278–292, 1991.

[5] Mark Defour. Shed skin: An optimizing python-to-c++ compiler, 2006.

[6] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. *SIGPLAN Not.*, 44(10):283–300, October 2009.

[7] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. *SAC 2009*, pages 1859–1866, 2009.

[8] Bastiaan Heeren, Bastiaan Heeren, Jurriaan Hage, Jurriaan Hage, Doaitse Swierstra, and Doaitse Swierstra. Generalizing hindley-milner type inference algorithms. Technical report, Institute of Information and Computing Sciences, Utrecht University, 2002.

[9] Alex Holkner and James Harland. Evaluating the dynamic behaviour of python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, pages 19–28, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[10] Mark P. Jones. Functional programming with overloading and higher-order polymorphism, 1995.

[11] Tobias Lindhal and Konstantinos Sagonas. Pratical type inference based on success typings. *Eighth ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 167–178, 2006.

[12] Robin Milner. A proposal for standard ml. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 184–197, New York, NY, USA, 1984. ACM.

[13] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[14] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[15] R.Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[16] Jeff Ruberg. Charming python with static typechecking, April 2012.

[17] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 126–137, Berlin, Heidelberg, 2003. Springer-Verlag.

[18] Michael Salib. Starkiller: A static type inferencer and compiler for python, 2004.

[19] Yusuke Shinyama. Pyntch - static code analyzer for python. `http://www.unixuser.org/~euske/python/pyntch/index.html`, 2009. Accessed: 29-04-2014.

[20] Guido van van Rossum. Adding optional static typing to python. `https://www.artima.com/weblogs/viewpost.jsp?thread=85551`, 2004. Accessed: 06-05-2014.

[21] Guido van van Rossum. Adding optional static typing to python – part ii. `https://www.artima.com/weblogs/viewpost.jsp?thread=86641`, 2005. Accessed: 06-05-2014.

[22] Guido van van Rossum. Adding optional static typing to python – stop the flames! `http://www.artima.com/weblogs/viewpost.jsp?thread=87182`, 2005. Accessed: 06-05-2014.

[23] Kenneth Walker and Ralph E. Griswold. Type inference in the icon programming language. Tr 93-32a, University of Arizona, Department of Computer Science, 1996.

[24] Yin Wang. Pysonar: a type inferencer and indexer for python. `http://yinwang0.wordpress.com/2010/09/12/pysonar/`, 2010. Accessed: 26-04-2014.