# Maximum Type Migration

ANONYMOUS AUTHOR(S)

Type migration is the task of adding as many types as possible to a program in a gradually typed language such as Flow, Hack, and Reticulated Python. Campora, Chen, Erwig, and Walkingshaw advanced the state of the art for this task in 2018 when they presented their new algorithm and tool. However, their algorithm has the major limitation that every parameter type must be either fully static or fully dynamic. This limitation has the upside that the tool scales linearly with the size of the program, but also the downside that the tool cannot add partially static types. For example, the tool cannot discover the type `List(Dyn)`, where `Dyn` is the fully dynamic type. We believe that type migration that can add partially static types would be of paramount importance for practical use. In this paper, we overcome this limitation and present a new approach to type migration. Our approach is the first to do type migration for the full gradually typed $\lambda$-calculus of Siek and Taha [Siek and Taha 2006] without restrictions. We define maximum type migration as a decision problem, prove that it is NP-complete, and present an implementation for a fragment of Python. Our NP-time algorithm represents the search space as a constraint system and decides whether any solution solves all hard constraints and more soft constraints than a given threshold. Our constraints can be solved by partially static types, which is the key to generalizing previous work. Our proof of NP-hardness uses partially static types, which suggests that lifting the limitation of previous work makes the problem significantly harder. Our implementation uses an approximation algorithm that combines heuristic search and constraint solving. Our experiments with four programs that total 166 lines of code show that our tool adds types via solving 84% of the soft constraints, on average. The resulting annotated programs all type check with our gradual type checker.

Additional Key Words and Phrases: keyword1, keyword2, keyword3

## 1 INTRODUCTION

### 1.1 Background

Static type checking has brought us more reliable software. Types make programs more readable, prevent entire classes of mistakes, and help compilers optimize data layout and data access. Types also make it easier to use libraries and to design interfaces, and they make it harder to misuse data and write bad programs. Languages such as Java and Haskell are statically typed and give those benefits, while languages such as JavaScript, PHP, and Python are dynamically typed. The dynamically typed languages have the strength that they give a programmer freedom to program in ways that fit no particular type system. The dichotomy between static and dynamic typing led researchers to produce a rich literature on how to combine them. In this paper we will focus on a well-known solution, namely the gradual typing of Siek and Taha [Siek and Taha 2006].

Gradual typing combines static typing with a dynamic type that we will write as `Dyn`. The idea is to use static types as much as possible and use `Dyn` otherwise. Gradual typing enables programmers to get the benefits of static typing when they use static types, and get the freedom of dynamic typing otherwise. While gradual typing is a weaker discipline than static typing, it gives well-understood benefits [Siek and Taha 2006; Siek et al. 2015a].

Recently, gradual typing has found practical use. For example, Hack is a gradually typed version of PHP, Flow is a gradually typed version of JavaScript, and Reticulated Python is a gradually typed version of Python. This has introduced the opportunity to migrate code from Python to Reticulated Python, and similarly for the other languages. Specifically, we can view a Python program as a Reticulated Python program in which every type annotation, implicitly, is `Dyn`. Now the migration path is to change some of those uses of `Dyn` to other types. The goal of *type migration* is to produce

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

a version of the program that type checks with a type checker for gradual types. The more static types we can add, the more benefits of static typing we can get.

Intuitively, type migration is akin to type inference with unusual types. While type inference has the goal to make the entire program be statically typed, type migration has the goal to do as much type inference as possible. As an analogy, we can say that type migration is to type inference what maximum satisfiability is to satisfiability. Specifically, while satisfiability tries to satisfy all the clauses in a Boolean formula, maximum satisfiability tries to satisfy as many clauses as possible. Notice that maximum satisfiability is harder: Max2SAT is NP-complete while 2SAT is solvable in polynomial time. Thus, we should expect that type migration is harder than type inference.

Can we automate type migration? Campora, Chen, Erwig, and Walkingshaw [Campora et al. 2018] showed how to use constraint solving to do type migration, proved correctness for a lambda-calculus, and presented a tool for Haskell. Their experiments show that their algorithm scales linearly with the size of the program. Our goal is to improve on their approach by lifting a major limitation.

The limitation of Campora et al.'s approach is that every parameter type must be either fully static or fully dynamic. For example, if a program has a parameter of type `Dyn`, then their algorithm may replace it with the static type `List(int)`. However, their algorithm is unable to replace `Dyn` with the partially static type `List(Dyn)`. We believe that this limitation is the key to the linear scaling of their tool. However, we also believe that type migration that can add partially static types would be of paramount importance for practical use. Thus, we are looking for a different design point in the quality-speed spectrum. In particular, we believe that the benefits of adding partially static types can be worth the increased time complexity, for three reasons. First, our approach can handle pieces of a program in isolation and make assumptions about other parts via a type environment. Second, our experiments show that a combination of constraint solving and heuristics can be a powerful for the migration problem. Third, type migration with partially static types is sufficiently important that even if a tool has to run overnight on a powerful computer, programmers will benefit.

### 1.2 Our results

We present a new approach to type migration that overcomes the limitation of previous work. Our approach is the first to do type migration for the full gradually typed $\lambda$-calculus of Siek and Taha [Siek and Taha 2006] without restrictions. We define maximum type migration as a decision problem, prove that it is NP-complete, and present an implementation for a fragment of Python. Our NP-time algorithm represents the search space as a constraint system and decides whether any solution solves more constraints than a given threshold. Thus we reduce maximum type migration to maximum satisfiability, as foreshadowed by the analogy above.

The approach of Campora et al. uses a different form of type constraints and asks how many argument positions can be given a static type. In comparison, we use more flexible constraints with a larger solution space and for which maximum satisfiability generalizes the focus on argument positions. In particular, our constraints can be solved by partially static types rather than only static types or `Dyn`.

Our proof of NP-hardness is a reduction from the maximum acyclic subgraph problem, which is NP-complete. Our reduction maps a graph to a program that uses partially static types. Thus, lifting the limitation from previous work appears to make the type migration problem significantly harder. Specifically, while Campora et al. focused on "a static type or `Dyn`" and achieved linear-time performance in practice, our NP-complete problem with partially static types is unlikely to be solvable in linear time.

Original:

```
@fields({'x':Dyn})
class Foo(object):
    def __init__(self:Dyn):
        pass

    def f(self:Dyn)->Dyn:
        self.x = self.g([1,2,3])
        return (self.g([True, False]))
                    .g([4,5])

    def g(self:Dyn,l:Dyn)->Dyn:
        return len(l)

print('%s' % Foo().f())
```

After migration by our tool:

```
@fields({'x':Dyn})
class Foo(object):
   def __init__(self:Foo):
      pass

   def f(self:Foo)->int:
      self.x = self.g([1,2,3])
      return (self.g([True,False]))
                  .g([4,5])

   def g(self:Foo,l:List(Dyn))->Dyn:
      return len(l)

print('%s' % Foo().f())
```

Fig. 1. A program before and after migration.

Our implementation uses an approximation algorithm that combines heuristic search and constraint solving. Our experiments with four programs that total 166 lines of code show that our tool adds types via solving 84% of the constraints, on average. The resulting annotated programs all type check with our gradual type checker.

The remainder of the paper has the following sections. In Section 2 we outline our approach via an example. In Section 3 we define the type migration problem for the $\lambda$-calculus of Cimini and Siek [Cimini and Siek 2016], and in Section 4 we prove that the type migration problem is NP-complete. In Section 5 we introduce our implementation for a fragment of Python and we describe the approximation algorithm that we use to solve constraints. In Section 6 we present our experimental results, in Section 7 we discuss related work, and in Section 8 we conclude.

## 2 EXAMPLE

In this section we give an example of maximum type migration. The goals are to highlight capabilities of our tool, compare with other tools, and give an impression of how our tool works. We will show an example in MiniPython, which intuitively is a subset of Python in which programs have type annotations that are gradual types. Those annotations are in the style of Reticulated Python. We will give additional details about MiniPython in Section 5 and in Appendix B.

*The example program.* Figure 1 shows both the original version of our example program and the version that our type-migrator tool produced. Both versions of the program type check with our gradual type checker, namely our gradual type checker for MiniPython. Note here that our gradual type checker is stricter than the gradual type checker for Reticulated Python. The reason is that the gradual type checker Reticulated Python supports only so-called *tag soundness*.

The original program consists of a class `Foo` with a single field `x` and three methods, and also a top-level print statement. Every type annotation in the original program is `Dyn`. The `f` method uses Python's built-in function `len`.

The migrated program has the same code but more static type annotations. Indeed, most of the types are static types, while the single partially static type is `List(Dyn)`. Let us explain how `List(Dyn)` meets a specific migration challenge.

*The challenge.* The original program contains two challenges for any type migration tool. We begin with the more obvious challenge. Notice the two calls `self.g([1,2,3])` and `self.g([True,False])` that pass a value of type `List(int)` and a value of type `List(bool)` to the method `g`. This requires some flexibility in the typing of `g`. Let us explore three ideas.

First, we might use *polymorphism*. A programmer would be able to migrate `g` as follows, here using Java-style syntax for a polymorphic method.

```
<X> def g(self:Foo,l:List(X))->int:
```

If `len` has a suitable polymorphic type, then the migrated program method `g` will type check with a type checker that supports polymorphism. However, now let us turn to the other challenge: the method `f` will fail to type check. The reason is that `f` does a method call on the result of its first call of `g`, which returns an integer. Thus, a polymorphic type checker would reject this program as untypable.

Second, we might use *subtyping*. A programmer would be able to migrate `g` as follows, here relying on the subtyping relations `List(int)` $\leq$ `List(object)` and `List(bool)` $\leq$ `List(object)`:

```
def g(self:Foo,l:List(object))->int:
```

Now we should be able to give `len` a suitable type that will make the program type check with a type checker that supports subtyping. However, like above, the method `f` remains untypable, and a type checker that supports subtyping would reject this program as untypable.

Third, we might use *gradual types*. A programmer would be able to migrate `g` as follows, here relying on that `List(int)` is so-called *consistent* with `List(Dyn)`, and that also `List(bool)` is consistent with `List(Dyn)`.

```
def g(self:Foo,l:List(Dyn))->int
```

Intuitively, consistency means that the types lead to no "contradictions." For example, `int` is inconsistent with `bool`, and `List(int)` is inconsistent with `List(bool)`, while while `int` is consistent with `Dyn`. As long as a program contains no inconsistencies, either immediate or derived, the program type checks with our gradual type checker.

Gradual typing can type check method `f` because of a mechanism called *matching*. For example, in the method call `self.g([True,False])`, the expression `self` gets the type `Dyn`. Now matching enables the result of the call to be `Dyn` as well, which enables the second call of `g` to type check in a similar manner.

Intuitively, matching means that while the preferred type of the receiver expression in a method call is a class name (in MiniPython), an alternative type is `Dyn`. This enables otherwise untypable method calls to type check with gradual types.

Together, consistency and matching are at the heart of gradual types and are the reason for the power of a gradual type system.

This paper presents a migration tool based on gradual types. The advantage of gradual types is that we can add types to parts of a program in such a way that the entire program type checks with a gradual type checker.

*Capabilities of our tool.* Our migration tool takes as input two items: a program like the ones in Figure 1 and a type environment that assigns a type to each predefined function, like `len` in Figure 1. If the user wants to do migration of a Python program, a natural first step is to annotate every field, argument, and return with `Dyn`, like in the original program in Figure 1, and thereby produce a MiniPython program. However, our tool can take as input a program with any type annotations so the user has the option to supply more informative annotations. Our tool will always make those type annotations more static (or leave them unchanged). This also models a situation where a programmer may have added some types while the program was built and where

Erased program:

```
class Foo(object):
   def __init__(self):
      pass

   def f(self):
      self.x = self.g([1,2,3])
      return self.g([True,False])

   def g(self,l):
      return len(l)

print('%s' % Foo().f())
```

Output from Typpete:

```
class Foo(object):
    def __init__(self:'Foo')->None:
        pass

    def f(self: 'Foo') -> int:
       self.x: complex = self.g([1,2,3])
        return self.g([True,False])
                   .g([4,5])

   def g(self:'Foo', l:Sequence)->int:
        return len(l)
print(('%s' % Foo().f()))
```

Fig. 2. Input and output from Typpete.

a, possibly different, programmer may have done some type migration. Then our tool can then do further migration, which we believe is an important use case. The input program has to type check with a gradual type checker, which tends to be the based case of programs in which most of the annotations are `Dyn`.

Additionally, the type environment reflects that we are doing type migration for an "open world" in which we rely on type assumptions about absent code. For example, our tool uses a type environment that assigns `len` the type `List(Dyn)` → `int`. If nothing is known about a piece of absent code, we can always give it type `Dyn`.

*Comparison with other tools.* Both programs in Figure 1 type check in Reticulated Python. However, Reticulated Python has no capability for type migration.

If we erase the annotations from the original program in Figure 1, we get a Python program that we can submit to the recent type inference tool Typpete [Hassan et al. 2018]. Note here that Typpete uses static types only, and it has no notion of gradual types and no notion of `Dyn`. Figure 2 shows the erased program and also the output from Typpete. The annotations produced by our tool and by Typpete are mostly the same, but differ for the type of `l`. In the output from Typpete, `l` has type `object`, which suggests that Typpete uses a subtyping approach. However, the output from Typpete fails to type check because `len` requires a list rather than a general object.

*How our tool works.* Our tool is based on a reformulation of the migration problem as a constraint problem. For our running example in Figure 1, our tool will first generate a set of constraints and then do a simplification step, leading to the *hard* constraints shown in Figure 3. We call those constraints hard because they must be satisfied for the output to type check with our gradual type checker.

We will give a high-level idea of the notation in Figure 3. Appendix B gives full details, while Section 4 gives a $\lambda$-calculus version.

In Figure 3, each number between 1 and 35 is a type variable. In the first and second column, we have type equations (written $\ldots = \ldots$) and consistency constraints (written $\ldots \sim \ldots$). In the third column, we have constraints derived from object creations, field accesses, and method calls. For example, the constraint `3 >g (14 15) -> 8 ;; 3 in [Foo]` says that we want to assign the type variable `3` either `Dyn` or a classname in the singleton set `{Foo}`. Additionally, if we assign it a non-`Dyn` classname, then we should match up the type `(14 15) -> 8` with the constraint for

```
 9 = List(10)
11 = 10
11 = int
12 = 10
12 = int
13 = 10
13 = int
19 = List(20)
21 = 20
21 = bool
22 = 20
22 = bool
25 = List(26)
27 = 26
27 = int
28 = 26
28 = int
31 = int
33 = Foo
34 = 33
```

```
14 ~ 3
15 ~ 9
16 ~ 8
17 ~ 4
23 ~ 3
24 ~ 19
29 ~ 18
30 ~ 25
31 ~ 7
32 ~ int
 6 ~ List(35)
```

```
Foo >f (3) -> 4
Foo >f (34) -> 32
Foo >g (5 6) -> 7
  3 >g (14 15) -> 8 ;; 3 in [Foo]
  3 >g (23 24) -> 18 ;; 3 in [Foo]
 18 >g (29 30) -> 17 ;; 18 in [Foo]
Foo >init (2)
Foo >init (33)
Foo >x 1
  3 >x 16 ;; 3 in [Foo]
```

Fig. 3. Hard constraints for the original program in Figure 1.

```
14 = 3
15 = 9
16 = 8
17 = 4
23 = 3
24 = 19
29 = 18
30 = 25
31 = 7
32 = int
 6 = List(35)

 3 in [Foo]
18 in [Foo]
```

Fig. 4. Soft constraints for the original program in Figure 1.

the method g in that class. In this case, Figure 3 lists the constraint `Foo >g (5 6) -> 7`. So, if we assign 3 the classname `Foo`, then we can go on to equate 14 and 5, we can equate 15 and 6, and we can equate 8 and 7.

In addition to the hard constraints in Figure 3, we also generate the *soft* constraints shown in Figure 4. Specifically, for each consistency constraint $s \sim s'$, we generate an equation $s = s'$. Additionally, for each constraint like `3 >g (14 15) -> 8 ;; 3 in [Foo]`, we generate `3 in [Foo]`. This last kind of constraint signals that we should assign 3 a classname and avoid `Dyn`. We call those constraints soft because we will try to satisfy a maximum number of them.

We find the notation `3 >g (14 15) -> 8 ;; 3 in [Foo]` convenient because it enables us to easily map a hard constraint to a soft constraint. The constraint generator derived the set `[Foo]` as the set of classes that each has a method g with the correct number of parameters.

Our tool solves all the hard constraints and as many of the soft constraints as it can, and then it outputs the migrated program in Figure 1. This migrated program is guaranteed to type check.

## 3 THE MAXIMUM TYPE MIGRATION PROBLEM

We will define the type migration problem for the gradually typed $\lambda$-calculus of Cimini and Siek [Cimini and Siek 2016]. Their calculus is a solid foundation for our work for three reasons. Specifically, it is a convenient reformulation of the calculus in the seminal paper by Siek and Taha on gradual typing [Siek and Taha 2006], it enables readers to check straightforwardly several established criteria for gradual typing [Siek et al. 2015a], and it is amenable to useful algorithms [Cimini and Siek 2016].

### 3.1 The gradually typed $\lambda$-calculus

Figure 5 shows the gradually typed $\lambda$-calculus [Cimini and Siek 2016]. Specifically, the figure shows the syntax, type rules, and definitions of consistency, matching, and precision. Let us highlight the key points of the definition.

The syntax of types includes the fully dynamic type `Dyn`. Examples of static types are `bool`, `int`, and `bool` $\rightarrow$ `int`. A partially static type is a type that has `Dyn` as leaf but isn't itself `Dyn`. Examples of partially static types are (`Dyn` $\rightarrow$ `Dyn`) and (`Dyn` $\rightarrow$ `int`).

The type judgments for numbers, booleans, variables, and $\lambda$-abstraction are the ones from simply typed $\lambda$-calculus. The single rule that departs from simply typed $\lambda$-calculus is the rule for application *(T-App)*. This rule uses matching and consistency to enable flexible type checking for an application $(E_1\ E_2)$, where $E_1$ has type $T_1$ and $E_2$ has type $T_2$. In case $T_1 = (T_{11} \rightarrow T_{12})$ and $T_2 = T_{11}$, the rule *(T-App)* behaves like the rule for simply typed $\lambda$-calculus and gives $(E_1\ E_2)$ the type $T_{12}$. However, two other cases are possible.

First, $T_2$ and $T_{11}$ can be different, as long as they are consistent. The definition of consistency says that `Dyn` is consistent with all types, that all types are consistent with themselves, and that for a function type, we define consistency recursively. Thus, `Dyn` $\sim$ `bool` and (`Dyn` $\rightarrow$ `Dyn`) $\sim$ (`Dyn` $\sim$ `bool`), etc. Consistency rules out some type mismatches because `bool` $\not\sim$ `int` and `bool` $\not\sim$ (`int` $\rightarrow$ `int`), etc. Consistency is *not* transitive, which was a careful design decision by the authors of the calculus. Thus, even though we do have `bool` $\sim$ `Dyn` and `Dyn` $\sim$ `int`, we cannot conclude `bool` $\sim$ `int`.

Second, $T_1$ may be `Dyn`, in which case the use of matching makes $T_{11} = T_{12} =$ `Dyn`. This has the effect of dropping all requirements on $T_2$ because since `Dyn` is consistent with all types, the requirement $T_2 \sim T_{11}$ is satisfied. Additionally, $T_{12} =$ `Dyn` means that the type of the application $(E_1\ E_2)$ will be `Dyn`.

One unique aspect of the gradually typed $\lambda$-calculus is its notion of precision that defines a partial order on types and a partial order on terms, both denoted $\sqsubseteq$. The idea is that if $T \sqsubseteq T'$, then $T'$ is more precise than $T$. We might also say that $T'$ is more *static* than $T$, in the sense that occurrences of `Dyn` are replaced with other types. Similarly, if $E \sqsubseteq E'$, then $E'$ has more precise type annotations than $E$. Precision is separate from the syntax and type system, and it enables easy statement of theorems.

The first three rules about precision define precision for types. Specifically, those rules say that that `Dyn` is the least element, that precision is reflexive, and that precision is defined recursively for function types. Notice that precision is covariant in both the domain and range of function types. The last three rules about precision define precision for terms. Specifically, those rules say that precision is reflexive, and defined recursively for $\lambda$-abstraction and application. Notice that the definition of precision for $\lambda$-abstraction relies on precision for types. We prove straightforwardly that both precision on types and precision on terms are partial orders.

The calculus has several desirable properties, including the following due to Cimini and Siek who called it *monotonicity with respect to precision.*

Notation:

We use $n$ to range over natural numbers and we use $x$ to range over term variables.

Syntax:

$$\begin{array}{rrcl} \text{(Types)} & T & ::= & \texttt{Dyn} \mid \texttt{bool} \mid \texttt{int} \mid T \to T \\ \text{(Terms)} & E & ::= & \mathit{True} \mid \mathit{False} \mid n \mid x \mid \lambda x : T.E \mid E\ E \\ \text{(Environments)} & \Gamma & ::= & \emptyset \mid \Gamma, x : T \end{array}$$

Type rules:

$$\Gamma \vdash n : \texttt{int}\ \textit{(T-Num)} \qquad \Gamma \vdash \mathit{True} : \texttt{bool}\ \textit{(T-True)} \qquad \Gamma \vdash \mathit{False} : \texttt{bool}\ \textit{(T-False)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textit{(T-Var)} \qquad \frac{\Gamma, x : T_1 \vdash E : T_2}{\Gamma \vdash (\lambda x : T_1.E) : T_1 \to T_2}\ \textit{(T-Abs)}$$

$$\frac{\Gamma \vdash E_1 : T_1 \quad T_1 \rhd (T_{11} \to T_{12}) \quad \Gamma \vdash E_2 : T_2 \quad T_2 \sim T_{11}}{\Gamma \vdash E_1\ E_2 : T_{12}}\ \textit{(T-App)}$$

Consistency:

$$T \sim \texttt{Dyn}\ \textit{(C-Dyn1)} \qquad \texttt{Dyn} \sim T\ \textit{(C-Dyn2)} \qquad \texttt{bool} \sim \texttt{bool}\ \textit{(C-Bool)} \qquad \texttt{int} \sim \texttt{int}\ \textit{(C-Int)}$$

$$\frac{T_1 \sim T_3 \quad T_2 \sim T_4}{(T_1 \to T_2) \sim (T_3 \to T_4)}\ \textit{(C-Arrow)}$$

Matching:

$$(T_1 \to T_2) \rhd (T_1 \to T_2)\ \textit{(M-Arrow)} \qquad \texttt{Dyn} \rhd (\texttt{Dyn} \to \texttt{Dyn})\ \textit{(M-Dyn)}$$

Precision:

$$\texttt{Dyn} \sqsubseteq T\ \textit{(P-Dyn)} \qquad T \sqsubseteq T\ \textit{(P-SameT)} \qquad \frac{T_1 \sqsubseteq T_3 \quad T_2 \sqsubseteq T_4}{T_1 \to T_2 \sqsubseteq T_3 \to T_4}\ \textit{(P-Arrow)}$$

$$E \sqsubseteq E\ \textit{(P-SameE)} \qquad \frac{T_1 \sqsubseteq T_2 \quad E_1 \sqsubseteq E_2}{\lambda x : T_1.E_1 \sqsubseteq \lambda x : T_2.E_2}\ \textit{(P-Abs)} \qquad \frac{E_1 \sqsubseteq E_3 \quad E_2 \sqsubseteq E_4}{(E_1\ E_2) \sqsubseteq (E_3\ E_4)}\ \textit{(P-App)}$$

Fig. 5. The gradually typed $\lambda$-calculus.

Theorem 3.1 (Monotonicity with respect to precision [Cimini and Siek 2016]). *$\forall \Gamma, E, E', T'$: if $\Gamma \vdash E' : T'$ and $E \sqsubseteq E'$, then for some $T$: $\Gamma \vdash E : T$ and $T \sqsubseteq T'$.*

Intuitively, Theorem 3.1 says that if we make the type annotations of a term $E'$ more dynamic, then the type of the entire term will become more dynamic, too. A key point here is that if $E'$ type checks, then the resulting term $E$ *always type checks*, irrespectively of the changes to the type annotations.

### 3.2 The maximum type migration problem

The goal of type migration is to move in the opposite direction of Theorem 3.1 and make the type annotations more static. Now we can wonder whether type migration satisfies a version of Theorem 3.1 that moves in the other direction. However, the answer is No. Specifically, consider the following false statement.

*This is false.* $\forall \Gamma, E, E', T'$: if $\Gamma \vdash E : T$ and $E \sqsubseteq E'$, then for some $T'$: $\Gamma \vdash E' : T'$ and $T \sqsubseteq T'$.

Intuitively, the false statement says that if we make the type annotations of a term $E$ more static, then the type of the entire term will become more static, too. The reason that this is false in some cases is that the term $E'$ with the more static types may *fail to type check*. For example, if $E = ((\lambda x : \texttt{Dyn}.x)\ 5)$, then the type of 5 is `int` and we have $\texttt{int} \sim \texttt{Dyn}$, so $E$ type checks. However, if we make the type annotation of $x$ more static by defining $E' = ((\lambda x : \texttt{bool}.x)\ 5)$, then we do have $E \sqsubseteq E'$ but we don't have $\texttt{bool} \sim \texttt{int}$, so $E'$ fails to type check. Thus, type migration faces the fundamental challenge to both make the type annotations more static *and* ensure that the resulting term type checks.

Let us turn to the question of how to define the meaning of *maximum* in a notion of maximum type migration. The previous work of Campora et al. counted the number of argument positions, namely one for each subterm $\lambda x : T.E$. This enabled a definition of maximum type migration in which *maximum* means the maximum number of argument positions with a type annotation that is static rather than Dyn. (Their work has the limitation that the only allowed type annotations are static types and Dyn.) Their definition has the good property that if *all* the type annotations are static types and the type environment uses static types exclusively, then the entire program has a type in simply typed $\lambda$-calculus.

Can we adapt Campora et al's notion of maximum type migration to partially static types? One idea could be to count the number of argument positions with a type annotation that is *not* Dyn. However, this has the problem that a partially static type such as $(\texttt{Dyn} \to \texttt{Dyn})$ becomes acceptable in a place where we were hoping for $(\texttt{int} \to \texttt{int})$. We can address this by counting the number of occurrences of Dyn in the type annotations and trying to minimize that count. However, this idea has the problem that if we migrate from Dyn to $\texttt{Dyn} \to \texttt{Dyn}$, then we actually *increase* the count. Related ideas have proved equally unsuccessful, however intuitive they may be. So, we leave to future work the task to define maximum migration for partially static types based on argument positions and counts of Dyn.

In this paper we define maximum type migration based on *counting uses of matching and consistency*. Specifically, we count *trivial uses* of matching and consistency in a type derivation. Our notion of *trivial* is that the two sides are *equal*. Specifically, $T_1 \rhd T_2$ is a trivial use of precision if $T_1 = T_2$, and $T_1 \sim T_2$ is a trivial use of consistency if $T_1 = T_2$. Now *maximum* means the maximum number of trivial uses of matching and consistency. If *all* the uses of matching and consistency are trivial, then each use of the rule *(T-App)* is like a use of the rule for application in simply typed $\lambda$-calculus. Notice though that the types are from the gradually typed $\lambda$-calculus and include Dyn and partially static types. While the calculus has no constants of type Dyn, the type environment may have entries that return type Dyn.

Let us now formalize our notion of counting trivial uses of matching and consistency. Our definition relies on the property that for $(E, \Gamma)$, if there exists $T$ such that $\Gamma \vdash E : T$, then both $T$ and the derivation of $\Gamma \vdash E : T$ are unique. This property is straightforward to prove.

Our definition uses the notion of *weight* of a type derivation, which is a count of the trivial uses of matching and consistency.

*Definition 3.2 (Weight of a type derivation).* For a type derivation $D$ of $\Gamma \vdash E : T$, we define that the weight of $D$, written $w(D)$, is a natural number, namely the sum of the trivial uses of matching in $D$ and the trivial uses of consistency in $D$. We define a trivial use of matching as a use of rule *(T-App)* where we derive $T_1 \rhd (T_{11} \to T_{12})$ because $T_1 = (T_{11} \to T_{12})$. Similarly, we define a trivial use of consistency as a use of rule *(T-App)* where we derive $T_2 \sim T_{11}$ because $T_2 = T_{11}$.

We will define a similar concept of weight on pairs $(E, \Gamma)$, which has the benefit that it leaves the type derivation implicit. However, the type derivation is never far away because if it exists, then it is unique (which is straightforward to prove).

*Definition 3.3 (Weight of an environment-term pair).* For a pair $(\Gamma, E)$ for which we have $D, T$ such that $D$ is the derivation of $\Gamma \vdash E : T$, the weight of $(\Gamma, E)$, written $w(E, \Gamma)$, is $w(E, \Gamma) = w(D)$.

As a convenient notation, we introduce an ordering $\leq_\Gamma$ on expressions that refines $\sqsubseteq$.

*Definition 3.4 (Typed precision).* $\forall E, \Gamma, E' : \ E \leq_\Gamma E' \iff (E \sqsubseteq E' \ \wedge \ \exists T' : \Gamma \vdash E' : T')$.

Intuitively, if we want to do type migration of $E$ in a context $\Gamma$, then we should consider the set of $E'$ such that $E \leq_\Gamma E'$. We will use $\leq_\Gamma$ to define the search space for type migrations.

Now we are ready to define maximum type migration. In order to define the problem as a decision problem, we use the standard technique of asking whether the count exceeds a threshold $k$.

*Definition 3.5 (Maximum type migration).* Given $(\Gamma, E, k)$, where $k$ is a natural number, decide $\exists E' : E \leq_\Gamma E' \wedge w(E', \Gamma) \geq k$.

Notice that the form of Definition 3.5 is generic. For example, we can change $w(E', \Gamma)$ to be a count of the number of argument positions with a type annotation that is *not* `Dyn`. However, in this paper we focus on a count of trivial uses of matching and consistency.

## 4 MAXIMUM TYPE MIGRATION IS NP-COMPLETE

In this section we will prove our main theoretical result, namely Theorem 4.1.

Theorem 4.1. *The maximum type migration problem is NP-complete.*

This theorem has two implications. First, the NP upper bound means that we should look for an approximation algorithm to solve practical instances of the problem. Second, the NP-hardness means that the problem for partially static types appears to be significantly harder than when limited to static types and `Dyn`. We leave as an open problem to settle the complexity for the limited case.

We find the NP-time algorithm more surprising than the NP-hardness because of the vast space of possible type migrations for a given program. Indeed, the NP-time algorithm gives hope that maximum type migration for partially static types can become practical.

In the following subsections we will prove several lemmas that eventually we will combine to prove Theorem 4.1. In Section 4.1 we introduce a notion of *simple constraints* and present a constraint solver that runs in cubic time. In Section 4.2 we introduce a notion of *general constraints* and prove that a notion of maximum satisfiability of general constraints is in NP. In Section 4.3 we show how to map the maximum type migration problem to the maximum satisfiability problem for general constraints. In Section 4.4 we prove that maximum type migration is in NP, and in Section 4.5 we prove that maximum type migration is NP-hard.

### 4.1 Simple constraints and satisfiability

A simple constraint is of one of the following three forms, where $v, v', v''$ range over type variables, and $s, s'$ range over $\{\texttt{Dyn}, \texttt{bool}, \texttt{int}, v, v \to v'\}$:

$$s = s' \qquad\qquad v \rhd v' \to v'' \qquad\qquad s \sim s'$$

We use $C$ to range over sets of simple constraints.

A *solution* for a set of simple constraints is a map:

$$\varphi \quad : \quad Var \to Type$$

that satisfies every constraint in the set, that is, for $s = s'$, we have $\varphi(s) = \varphi(s')$, for $v \rhd v' \to v''$, we have $\varphi(v) \rhd \varphi(v') \to \varphi(v'')$, and for $s \sim s'$, we have $\varphi(s) \sim \varphi(s')$. If $\varphi$ is a solution of $C$, then we write $\varphi \models C$.

*Definition 4.2.* A set of simple constraints is *closed* if it satisfies the following rules. Each rule says that if zero, one, or two constraints are in the set, then one or two other constraints are also in the set.

1st rule: the equality relation = is reflexive, symmetric, and transitive.

2nd rule: the relation $\sim$ is reflexive and symmetric.

3rd rule: if $s_1 \to s_2 = s_1' \to s_2'$, then $s_1 = s_1'$ and $s_2 = s_2'$.

4th rule: if $s_1 \to s_2 \sim s_1' \to s_2'$, then $s_1 \sim s_1'$ and $s_2 \sim s_2'$.

5th rule: if $s = s'$ and $s' \sim s''$ and $s \sim s''$.

6th rule: if $v \rhd v' \to v''$ and $v = \texttt{Dyn}$, then $v' = \texttt{Dyn}$ and $v'' = \texttt{Dyn}$.

7th rule: if $v \rhd v' \to v''$ and ($v = s$ or $v' = s$ or $v'' = s$), where $s \in \{\texttt{bool}, \texttt{int}, s_1 \to s_2\}$, then $v = v' \to v''$.

The *closure* of a constraint $C$ is the smallest closed constraint that contains $C$ as a subset.

Lemma 4.3. *A set of simple constraints and its closure have the same solutions.*

Proof. Observe that any solution of the closure of $C$ is also a solution of $C$, since $C$ has fewer constraints. Conversely, the closure of $C$ can be constructed from $C$ by iterating the closure rules,

and it follows inductively that any solution of $C$ satisfies the additional constraints added by this process. □

*Definition 4.4.* A set of simple constraints is *well-formed* if none of $\texttt{bool} \sim \texttt{int}$ and $\texttt{bool} \sim (s \to s')$ and $\texttt{int} \sim (s \to s')$ are in the set.

Lemma 4.5. *If $\varphi \models C$, then $C$ is well-formed.*

Proof. Straightforward. □

Following Bloom, Elgot, and Wright [Bloom et al. 1980] and also Courcelle [Courcelle 1983], we say that a variable $v$ is *singular* in a closed constraint set $C$, if for every constraint in $C$ of the form $v = s$, we have that $s$ is a variable.

For a substitution $\varphi$, define

$$\varphi' \quad = \quad \varphi \circ \{\ v := \texttt{Dyn} \mid v \text{ is singular}\ \}$$

For a set $C$ of simple constraints, let $Eq(C) \subseteq C$ consist of the constraints of the form $s = s'$. Let *size* be a count of the number of nodes in the syntax tree for a type.

Theorem 4.6. *For a closed and well-formed constraint set $C$, we have $C$ is satisfiable if and only $Eq(C)$ is satisfiable.*

Proof. In the forwards direction, suppose $\varphi \models C$. Immediately, we have $\varphi \models Eq(C)$.

In the backwards direction, suppose $Eq(C)$ is satisfiable and let $\varphi$ be the most general unifier of $Eq(C)$. We will show $\varphi' \models C$ by considering the constraints in $C$ in turn.

First, consider $Eq(C)$. We have that $\varphi$ maps singular variables to themselves, so $\varphi' \models Eq(C)$.

Second, consider a constraint $v \triangleright v' \to v''$. We have five cases.

- Suppose $C$ contains a constraint $v = \texttt{Dyn}$. The 6th closure rule gives that $C$ contains the constraints $v' = \texttt{Dyn}$ and $v'' = \texttt{Dyn}$, so $\varphi' \models v \triangleright v' \to v''$.
- Suppose $C$ contains a constraint $v = \texttt{bool}$. The 7th closure rule gives that $C$ contains $v = v' \to v''$, which contradicts that $C$ is well-formed.
- Suppose $C$ contains a constraint $v = \texttt{int}$. The 7th closure rule gives that $C$ contains $v = v' \to v''$, which contradicts that $C$ is well-formed.
- Suppose $C$ contains a constraint $v = v_1 \to v_2$. The 7th closure rule gives that $C$ contains $v = v' \to v''$, so $\varphi \models v = v' \to v''$, hence $\varphi' \models v = v' \to v''$, hence $\varphi' \models v \triangleright v' \to v''$.
- Otherwise, $v$ is singular, so $\varphi'(v) = \texttt{Dyn}$. The 7th closure rules gives that if $v' = s$ or $v'' = s$, where $s \in \{\texttt{bool}, \texttt{int}, s_1 \to s_2\}$, then $v = v' \to v''$, which contradicts that $v$ is singular. So, we must have that either $v' = \texttt{Dyn}$ or $v'$ is singular, and similarly that either $v'' = \texttt{Dyn}$ or $v''$ is singular. In all combinations of those cases, we have $\varphi'(v') = \texttt{Dyn}$ and $\varphi'(v'') = \texttt{Dyn}$, so $\varphi' \models v \triangleright v' \to v''$.

Third, consider the constraints of the form $s \sim s'$. Suppose $\varphi'$ does not satisfy all those constraints. We will show that this leads to a contradiction. Let us first show that we can choose a constraint $s \sim s'$ such that 1) $\varphi'$ does not satisfy $s \sim s'$ and 2) $size(\varphi'(s)) + size(\varphi'(s'))$ is minimal among all such constraints and 3) neither of $s, s'$ is a nonsingular variable. Let us begin by choosing a constraint $s \sim s'$ that satisfies (1) and (2). If $s \sim s'$ doesn't satisfy (3) because $s$ is a nonsingular variable, then we can find a constraint $s = s_1$, where $s_1$ is a nonvariable. Similarly, if $s \sim s'$ doesn't satisfy (3) because $s'$ is a nonsingular variable, then we can find a constraint $s' = s'_1$, where $s'_1$ is a nonvariable. Thus, in all cases where $s \sim s'$ doesn't satisfy (3), we can use that $C$ is closed and the 5th closure rule, together with that $\varphi' \models Eq(C)$, to conclude that $C$ contains a constraint that satisfies all of (1), (2), (3). For simplicity, let us call that constraint $s \sim s'$. Now we have two cases.

- If $s$ is Dyn or a singular variable, then $\varphi'(s) = \texttt{Dyn}$, so $\varphi' \models s = s'$, which contradicts (1). Similar reasoning applies when $s'$ is Dyn or a singular variable.
- Otherwise, $s, s'$ are both in $\{\texttt{bool}, \texttt{int}, v \to v'\}$, so from that $C$ is well-formed, we have that only three cases are possible. First, $s = \texttt{bool}$ and $s' = \texttt{bool}$, in which case $\varphi' \models s = s'$, which contradicts (1). Second, $s = \texttt{int}$ and $s' = \texttt{int}$, in which case $\varphi' \models s = s'$, which contradicts (1). Third, $s = v_1 \to v_2$ and $s' = v_1' \to v_2'$, in which case we use that $C$ is closed and the 4th closure rule to conclude that $C$ contains $v_1 \sim v_1'$ and $v_2 \sim v_2'$. Here we have $size(\varphi'(v_1)) + size(\varphi'(v_1')) < size(\varphi'(s)) + size(\varphi'(s'))$, which contradicts (2).

In summary, we reach a contradiction in every case, so $\varphi'$ satisfies all constraints of the form $s \sim s'$. □

*Definition 4.7 (Constraint solver).* An algorithm for solving a set of simple constraints.
Input: a set $C$ of simple constraints.
Output: true, if $C$ is satisfiable, and false otherwise.
Method:
1. Close $C$ to produce $C'$.
2. If $C'$ is well-formed, then go to step 3, else output false.
3. Output whether $Eq(C')$ is satisfiable.

Theorem 4.8. *The constraint solver B.1 decides in $O(n^3)$ time whether a set of simple constraints is satisfiable.*

Proof. Correctness: Lemma 4.3 gives that $C$ and $C'$ have the same solutions. Lemma 4.5 gives that if step 2 outputs false, then $C'$ has no solution. But if the algorithm goes to step 3, then $C'$ is closed and well-formed, so Theorem 4.6 gives that step 3 decides whether $C'$ is satisfiable.

Time complexity: step 1 takes $O(n^3)$ time, step 2 takes $O(n)$ time, and step 3 takes $O(n)$ time. So, in total the algorithm takes $O(n^3)$ time. □

### 4.2 General constraints and maximum satisfiability

A general constraint is of one of the following six forms, where $v, v', v''$ range over type variables, and $t$ ranges over types:

$$v = t \qquad v = v' \qquad v = v' \to v'' \qquad t \sqsubseteq v \qquad v \rhd v' \to v'' \qquad v \sim v'$$

We use $B$ to range over sets of general constraints.

A *solution* for a set of general constraints is a map:

$$\varphi \quad : \quad Var \to Type$$

that satisfies every constraint in the set, that is, for for $v = t$, we have $\varphi(v) = t$, for $v = v'$, we have $\varphi(v) = \varphi(v')$, for $v = v' \to v''$, we have $\varphi(v) = \varphi(v') \to \varphi(v'')$, for $t \sqsubseteq v$, we have $t \sqsubseteq \varphi(v)$, for $v \rhd v' \to v''$, we have $\varphi(v) \rhd \varphi(v') \to \varphi(v'')$, and for $v \sim v'$, we have $\varphi(v) \sim \varphi(v')$. If $\varphi$ is a solution of $B$, then we write $\varphi \models B$.

For a set $B$ of general constraints, define a set $B^{soft}$ of general constraints as follows. For each constraint of the form $v \rhd v' \to v''$ in $B$, we have a constraint $v = v' \to v''$ in $B^{soft}$. Additionally, for each constraint of the form $v \sim v'$ in $B$, we have a constraint $v = v'$ in $B^{soft}$. We can think of $B$ as *hard* constraints that must be satisfied and $B^{soft}$ as *soft* constraints that may be satisfied.

The definition of $B^{soft}$ enables us to define maximum satisfiability of general constraints in an unusual way that focuses on solving all the hard constraints and solving a maximum number of soft constraints. As usual, we define a version of the problem as a decision problem by introducing a threshold $k$.

*Definition 4.9 (Maximum satisfiability of general constraints).*
Given $(B, k)$, decide $\exists B', \varphi : B' \subseteq B^{soft} \wedge |B'| \geq k \wedge \varphi \models (B \cup B')$.

Theorem 4.10. *Satisfiability of general constraints is decidable in $O(n^3)$ time.*

Proof. Given a set $B$ of general constraints, first we transform $B$ to a set $C$ of simple constraints. We do this by repeating the following transformation until it no longer has an effect.

| From: | to: |
|---|---|
| $v = t' \to t''$ | $v = v' \to v'' \ \wedge \ v' = t' \ \wedge \ v'' = t''$ where $v', v''$ are fresh type variables |
| $\texttt{Dyn} \sqsubseteq v$ | (no constraint) |
| $\texttt{bool} \sqsubseteq v$ | $v = \texttt{bool}$ |
| $\texttt{int} \sqsubseteq v$ | $v = \texttt{int}$ |
| $t' \to t'' \sqsubseteq v$ | $v = v' \to v'' \ \wedge \ t' \sqsubseteq v' \ \wedge \ t'' \sqsubseteq v''$ |

Notice that the simplification procedure preserves the set of solutions. Then we use Algorithm B.1 to check satisfiability of $C$ in $O(n^3)$ time (Theorem 4.8). □

Theorem 4.11. *Maximum satisfiability of general constraints is in NP.*

Proof. Let $(B, k)$ be an instance of the maximum satisfiability of the general constraints problem. We can guess a subset $B'$ of $B^{soft}$ such that $B'$ is of size at least $k$. Now use Theorem 4.10 to check satisfiability of $(B \cup B')$ in polynomial time. □

### 4.3 From maximum type migration to maximum satisfiability

In this section, we reduce the maximum type migration problem to the maximum satisfiability of general constraints problem. Specifically, we can rephrase an instance $(\Gamma, E, k)$ of the maximum type migration problem in terms of solving a system of general constraints that is derived from $(\Gamma, E, k)$.

Assume that $E$ has been $\alpha$-converted so that all bound variables are distinct from each other and distinct from the free variables. Let $X$ be the set of $\lambda$-variables $x$ occurring in $E$, and let $Y$ be a set of variables disjoint from $X$ consisting of a variable $[\![F]\!]$ for every occurrence of the subterm $F$ in $E$. The notation $[\![F]\!]$ is ambiguous because there may be more than one occurrence of some subterm $F$ in $E$. However, it will always be clear from context which occurrence is meant. Let $Z$ be a set of variables disjoint for $X$ and $Y$ consisting of a variable $\langle G \rangle$ for every occurrence of the subterm $(F\ G)$ in $E$. Also in this case, the notation $\langle G \rangle$ may be ambiguous but will be clear from the context which occurrence is meant.

From $E$ and $\Gamma$, where $FV(E) \subseteq Dom(\Gamma)$, we generate a set $B(E, \Gamma)$ of general constraints as follows.

| For every occurrence in $E$ of a subterm of this form: | generate this constraint: |
|---|---|
| $n$ | $[\![n]\!] = Int$ |
| $True$ | $[\![True]\!] = Bool$ |
| $False$ | $[\![False]\!] = Bool$ |
| (free variable) $x$ | $[\![x]\!] = \Gamma(x)$ |
| (bound variable) $x$ | $[\![x]\!] = x$ |
| $\lambda x : S.F$ | $[\![\lambda x : S.F]\!] = x \to [\![F]\!] \ \wedge \ S \sqsubseteq x$ |
| $F\ G$ | $[\![F]\!] \rhd \langle G \rangle \to [\![FG]\!] \ \wedge \ \langle G \rangle \sim [\![G]\!]$ |

Before we state that the above reduction is correct, we introduce some helper notation. We let $\wp(X)$ denote the power set of $X$, that is, the set of all subsets of $X$. We define let $Dom(\Gamma)$ denote

the domain of $\Gamma$: $Dom(\emptyset) = \emptyset$ and $Dom(\Gamma, x : T) = Dom(\Gamma) \cup \{x\}$. We let $FV(E)$ denote the set of free variables of $E$: $FV(n) = \emptyset$ and $FV(True) = \emptyset$ and $FV(False) = \emptyset$ and $FV(x) = \{x\}$ and $FV(\lambda x : T.F) = FV(F) \setminus \{x\}$ and $FV(E_1\ E_2) = FV(E_1) \cup FV(E_2)$. We say that $\varphi$ *extends* $\Gamma$ and write $\varphi \geq \Gamma$ if and only if $\forall x \in Dom(\Gamma) : \varphi(x) = \Gamma(x)$, that is, $\varphi$ agrees with $\Gamma$ on the domain of $\Gamma$. Finally, if $\varphi$ is a function from type variables to types, then we define the function $G_\varphi$ from terms to terms:

$$\begin{array}{rcl} G_\varphi(n) & = & n \\ G_\varphi(True) & = & True \\ G_\varphi(False) & = & False \\ G_\varphi(x) & = & x \\ G_\varphi(\lambda x : T.F) & = & \lambda x : \varphi(x).G_\varphi(F) \\ G_\varphi(E_1\ E_2) & = & G_\varphi(E_1)\ G_\varphi(E_2) \end{array}$$

Theorem 4.12 (Correctness of the reduction). *$\forall(\Gamma, E, k)$: if $FV(E) \subseteq Dom(\Gamma)$, then*

$$\exists E' : E \leq_\Gamma E' \wedge w(E', \Gamma) \geq k \iff \exists B', \varphi : B' \subseteq B(E, \Gamma)^{soft} \wedge |B'| \geq k \wedge \varphi \models (B(E, \Gamma) \cup B')$$

Proof. Theorem 4.12 is immediate from Lemma 4.17 below. □

Before we state and prove Lemma 4.17, we state four lemmas that each has a straightforward proof. We give the proofs of Lemmas 4.13, 4.14, 4.15, 4.16 in Appendix A.

Lemma 4.13. *$\forall\varphi, \Gamma, E, T$: if $\varphi \geq \Gamma$ and $FV(E) \subseteq Dom(\Gamma)$ and $\varphi \vdash E : T$, then $\Gamma \vdash E : T$.*

Lemma 4.14. *$\forall\varphi, \Gamma$:*
1. *If $\varphi \models B(\lambda x : S.F, \Gamma)$, then $\varphi \models B(F, \{(x : \varphi(x))\} \cup \Gamma)$.*
2. *If $\varphi \models B(E_1E_2, \Gamma)$, then $\varphi \models B(E_1, \Gamma)$ and $\varphi \models B(E_2, \Gamma)$.*

Lemma 4.15. *$\forall\varphi, E, \Gamma$: if $\varphi \models B(E, \Gamma)$ and $\varphi \geq \Gamma$, then $\varphi \vdash G_\varphi(E) : \varphi([\![E]\!])$.*

Lemma 4.16. *$\forall E, \Gamma, E'$: if $E \leq_\Gamma E'$, then exactly one of the following holds:*
1. *$E = n$ and $E' = n$.*
2. *$E = True$ and $E' = True$.*
3. *$E = False$ and $E' = False$.*
4. *$E = x$ and $E' = x$.*
5. *$E = \lambda x : S.F$ and $E' = \lambda x : S'.F'$ and $F \leq_{\{(x:S)\}\cup\Gamma} F'$*
6. *$E = E_1E_2$ and $E' = E_1'E_2'$ and $E_1 \leq_\Gamma E_2'$ and $E_1 \leq_\Gamma E_2'$.*

Now we are ready to state and prove Lemma 4.17. The proof uses a technique from a paper by Kozen, Palsberg, and Schwartzbach [Kozen et al. 1994].

Lemma 4.17. *$\forall E, \Gamma, E', k'$ : if $FV(E) \subseteq Dom(\Gamma)$, then*

$$\begin{array}{l} E \leq_\Gamma E' \ \wedge\ w(E', \Gamma) = k' \iff \\ \exists B', \varphi : \varphi \geq \Gamma \ \wedge\ E' = G_\varphi(E) \ \wedge\ B' \subseteq B(E, \Gamma)^{soft} \ \wedge\ |B'| = k' \ \wedge\ \varphi \models (B(E, \Gamma) \cup B') \end{array}$$

Proof. We will prove each of the two directions in turn.

*Forwards direction.* By expanding Definition 3.4, we have $E \sqsubseteq G_\varphi(E)$, $\exists T' : \Gamma \models G_\varphi(E) : T'$, and $w(E', \Gamma) = k'$. We show that $\exists B', \varphi : \varphi \geq \Gamma \ \wedge\ E' = G_\varphi(E) \ \wedge\ B' \subseteq B(E, \Gamma)^{soft} \ \wedge\ |B'| = k' \ \wedge\ \varphi \models B(E, \Gamma) \cup B'$.

We will construct $\varphi$ and $B'$ with the desired properties.

For an occurrence of $n$ let $\varphi([\![n]\!]) = Int$. For an occurrence of $True$ let $\varphi([\![True]\!]) = Bool$. For an occurrence of $False$ let $\varphi([\![False]\!]) = Bool$. For every $\lambda$-variable $x$ occurring freely in $E$, let $\varphi(x) = \Gamma(x)$.

For every bound $\lambda$-variable $x$, let $\lambda x : S.F$ be the subterm of $E$ in which it is bound. Consider the derivation $\Gamma \vdash E' : T'$ and find the occurrence $\lambda x : S'.F'$ in $E'$. Find the last judgment in the derivation of the form $\Gamma', (x : S') \vdash \lambda x : S'.F' : S' \to U'$ for some $U'$ and let $\varphi(x) = S'$.

For every occurrence of a subterm of the form $\lambda x : S.F$ in $E$, consider the derivation $\Gamma \vdash E' : T'$ and find the occurrence $\lambda x : S'.F'$ in $E'$. Find the last judgment in the derivation of the form $\Gamma', (x : S') \vdash \lambda x : S'.F' : S' \to U'$ for some $U'$ and let $\varphi([\![F]\!]) = U'$ and $\varphi([\![\lambda x : S.F]\!]) = \varphi(x) \to \varphi([\![F]\!])$.

Finally, for every occurrence of a subterm of the form $E_1E_2$ in $E$, consider the derivation $\Gamma \vdash E' : T'$. The rule *T-App* must have been used. Find the last judgment in the derivation is of the form $\Gamma' \vdash E'_1 : U'_1$ involving that occurrence of $E'_1$ and the last judgment involving the occurrence of $E'_2$ where $\Gamma' \vdash E'_2 : U'_2$. Then $U'_1 \rhd U'_{11} \to U'_{12}$ and $U'_2 \sim U'_{11}$. Let $\varphi(\langle E_2 \rangle) = U'_{11}$ and $\varphi([\![E_1E_2]\!]) = U'_{12}$. If $U'_1 = U'_{11} \to U'_{12}$ then let $[\![E_1]\!] = \langle E_2 \rangle \to [\![E_1E_2]\!] \in B'$ and if $U'_2 = U'_{11}$ then let $[\![E_2]\!] = \langle E_2 \rangle \in B'$.

We have $\varphi \geq \Gamma$ because $\varphi(x) = \Gamma(x)$ where $x$ is a variable occurring freely in $E$.

We will now show that $\varphi \models B(E, \Gamma) \cup B'$.

For occurrences $E$ of the form $n$, $True$ and $False$ and free-variables $x$, clearly $\varphi \models B(E, \Gamma) \cup \emptyset$.

For an occurrence of a bound $\lambda$-variable $x$, find the unique application of the *T-Abs* deriving the judgment $\Gamma' \vdash \lambda x : S.F : S \to U$ from the premise $\Gamma', (x : S) \vdash F : U$. Then $S \sqsubseteq \varphi(x)$. The rule *T-Var* must have been applied to obtain a judgment of the form $\Gamma'' \vdash x : \varphi(x)$. Thus $\varphi(x) = \varphi([\![x]\!])$. Then $\varphi \models B(x, \Gamma) \cup \emptyset$.

For an occurrence of a subterm of the form $\lambda x : S.F$ , find the unique application of the *T-Abs* deriving the judgment $\Gamma' \vdash \lambda x : S'.F' : S' \to U'$ from the premise $\Gamma', x : S' \vdash F' : U'$. Then $\varphi(x) = S'$, $\varphi([\![F]\!]) = U'$, and $\varphi(x) \to \varphi([\![F]\!]) = S' \to U' = \varphi([\![\lambda x : S.F]\!])$. Since $\varphi(x) = S'$ we have $S \sqsubseteq \varphi(x)$ then $\varphi \models B(\lambda x : S.F, \Gamma) \cup \emptyset$.

For an occurrence of a subterm of the form $E_1E_2$, find the unique application of the *T-App* deriving the judgment $\Gamma' \vdash E_1E_2 : U_{12}$ from the premises $\Gamma' \vdash E_1 : U_1$ and $U_1 \rhd U_{11} \to U_{12}$ and $\Gamma' \vdash E_2 : U_2$ and $U_2 \sim U_{11}$. Then $\varphi([\![E_1]\!]) = U_1$ which implies that $\varphi([\![E_1]\!]) \rhd U_{11} \to U_{12}$ by *M-Arrow*. Furthermore, $\varphi(\langle E_2 \rangle) = U_{11}$ and $\varphi([\![E_2]\!]) = U_2$. We have $\varphi([\![E_1]\!]) \rhd \varphi(\langle E_2 \rangle) \to \varphi([\![E_1E_2]\!])$ and $\varphi(\langle E_2 \rangle) \sim \varphi([\![E_2]\!])$. Then we have that if $\varphi([\![E_1]\!]) = \varphi(\langle E_2 \rangle) \to \varphi([\![E_1E_2]\!])$ then $\varphi \models [\![E_1]\!] = \langle E_2 \rangle \to [\![E_1E_2]\!]$ and if $\varphi(\langle E_2 \rangle) = \varphi([\![E_2]\!])$ then $\varphi \models \langle E_2 \rangle = [\![E_2]\!]$. So $\varphi \models B(E_1E_2, \Gamma) \cup B'$.

We show that $E' = G_\varphi(E)$. Since $E \leq_\Gamma E'$ we have $\Gamma \vdash E' : T$. We proceed by induction on $E'$.

Case: $n$. Then $n = G_\varphi(n)$.

Case: $True$, $False$, $x$. Similar to case $n$.

Case: $\lambda x : S'.F'$. Then we have $\lambda x : S.F \leq_\Gamma \lambda x : S'.F'$. We can apply clause 5 of Lemma 4.16 to get $F \leq_{\{(x:S')\} \cup \Gamma} F'$. Since $\varphi \geq \{(x : S') \cup \Gamma\}$ we can apply our induction hypothesis to get $F' = G_\varphi(F)$. Then we have $\lambda x : S'.F' = \lambda x : \varphi(x).G_\varphi(F) = G_\varphi(\lambda x : S.F)$.

Case: $E'_1E'_2$. Then we have $E_1E_2 \leq_\Gamma E'_1E'_2$, then by clause 6 of Lemma 4.16, we have $E_1 \leq_\Gamma E'_1$ and $E_2 \leq_\Gamma E'_2$. Then we can apply the induction hypothesis, we have $E'_1 = G_\varphi(E_1)$ and $E'_2 = G_\varphi(E_2)$. Then by properties of $G_\varphi$, we have $E'_1E'_2 = G_\varphi(E_1)G_\varphi(E_2) = G_\varphi(E_1E_2)$.

Finally, we know that $w(E', \Gamma) = k' = |B'|$ by proceeding by induction on the derivation $\Gamma \vdash E' : T$ in a similar manner to what we did earlier in the proof.

*Backwards direction.* Suppose $\exists B', \varphi : \varphi \geq \Gamma \ \wedge \ E' = G_\varphi(E) \ \wedge \ B' \subseteq B(E, \Gamma)^{soft} \ \wedge \ |B'| = k' \ \wedge \ \varphi \models B(E, \Gamma) \cup B'$.

By expanding the Definition 3.4 we show that: $E \sqsubseteq G_\varphi(E)$, $\exists T' : \Gamma \models G_\varphi(E) : T'$, and $w(E', \Gamma) = k'$.

To show that $E \sqsubseteq G_\varphi(E)$ we will proceed by induction on $E$.

Case: $n$. $n \sqsubseteq n$ by *P-SameE*.

Case: $True$, $False$, $x$. Similar to case $n$.

Case: $\lambda x : S.F$. Since $\varphi \models B(\lambda x : S.F, \Gamma) \cup B'$ then $\varphi \models S \sqsubseteq \varphi(x)$. So we have $S \sqsubseteq \varphi(x)$. We also have that $\varphi \models B(F, x : \varphi(x), \Gamma)$ by Lemma 4.14. Given that $(x : \varphi(x)) \in \varphi$ and $\varphi \geq \Gamma$, then we have $\varphi \geq \{(x : \varphi(x))\} \cup \Gamma$. From our proposition $E' = G_\varphi(\lambda x : S.F) = \lambda x : \varphi(x).G_\varphi(F) = \lambda x : S'.F'$ for some $S' = \varphi(x)$ and $F' = G_\varphi(F)$. We can now apply our induction hypothesis on $F' = G_\varphi(F)$ to get that $F \sqsubseteq G_\varphi(F)$. And we have previously shown that $S \sqsubseteq \varphi(x)$. Then by *P-Abs*, we have $\lambda x : S.F \sqsubseteq \lambda x : \varphi(x).G_\varphi(F)$. By the properties of $G_\varphi$, $\lambda x : \varphi(x).G_\varphi(F) = G_\varphi(\lambda x : S.F)$. So $\lambda x : S.F \sqsubseteq G_\varphi(\lambda x : S.F)$.

Case: $E_1E_2$. Since $\varphi \models B(E_1E_2, \Gamma) \cup B'$ and $L \geq \Gamma$, then by Lemma 4.14 and without loss of generality, $L \models B(E_1, \Gamma)$. From $FV(E_1E_2) \subseteq Dom(\Gamma)$ we have that $FV(E_1) \subseteq Dom(\Gamma)$. From $E' = G_\varphi(E_1E_2) = G_\varphi(E_1)G_\varphi(E_2) = E_1'E_2'$ for some $E_1' = G_\varphi(E_1)$ and $E_2' = G_\varphi(E_2)$. Then we can apply our induction hypothesis to get that $E_1 \sqsubseteq G_\varphi(E_1)$ and $E_2 \sqsubseteq G_\varphi(E_2)$. By *P-App*, we have that $E_1E_2 \sqsubseteq G_\varphi(E_1)G_\varphi(E_2)$. By properties of $G_\varphi$, we have that $G_\varphi(E_1)G_\varphi(E_2) = G_\varphi(E_1E_2)$. Then $E_1E_2 \sqsubseteq G_\varphi(E_1E_2)$.

Next, we show that $\exists T' : \Gamma \vdash G_\varphi(E) : T'$. First, we apply Lemma 4.15 since $\varphi \models B(E, \Gamma)$ and $\varphi \geq \Gamma$. So we have that $\varphi \vdash G_\varphi(E) : \varphi([\![E]\!])$. By properties of $G_\varphi$, we have that $FV(E) = FV(G_\varphi(E))$. By the assumption that $FV(E) \subseteq Dom(\Gamma)$ we get that $FV(G_\varphi(E)) \subseteq Dom(\Gamma)$. So we can apply Lemma 4.13 to get that $\Gamma \vdash G_\varphi(E) : \varphi([\![E]\!])$.

Finally, we will show that $w(E', \Gamma) = k'$ assuming that $|B'| = k'$. We proceed by induction on the derivation of $\Gamma \vdash E' : T$ for some $T$.

Suppose that the last rule in the derivation of $\Gamma \vdash E' : T$ was *T-Num*. Then $E = n$ and by Lemma 4.16, $E' = n$. The derivation not have any rules involving $\sim$ or $\rhd$ therefore we get that $w(\Gamma, E') = 0$. We have that $B^{soft} = \emptyset$ so $B' \subseteq B^{soft} = \emptyset$ and $|B'| = 0 = k'$. Therefore, $w(E', \Gamma) = k'$ where $k' = 0$.

A similar argument applies to the cases: *T-True*, *T-False*, *T-Var* and *T-Abs*.

Suppose that the last rule in the derivation of $\Gamma \vdash E' : T$ was *T-App*. Then $E' = E_1'E_2'$ and by Lemma 4.16, $E = E_1E_2$. Consider the uses $T_1 \rhd (T_{11} \to T_{12})$ and $T_2 \sim T_{11}$ in the derivation. We have $[\![E_1]\!] \rhd \langle E_2 \rangle \to [\![E_1E_2]\!] \in B(E_1E_2, \Gamma)$. Suppose that $T_1 = (T_{11} \to T_{12}) \in B'$. Then since $\varphi \models B(E_1E_2, \Gamma) \cup B'$ then for the constraint $[\![E_1]\!] \rhd \langle E_2 \rangle \to [\![E_1E_2]\!]$ we will have $\varphi \models \varphi([\![E_1]\!]) = \varphi(\langle E_2 \rangle) \to \varphi([\![E_1E_2]\!])$. Then in the derivation of $\Gamma \vdash E_1'E_2' : T$, $T_1 = (T_{11} \to T_{12})$ must have been used to derive $T_1 \rhd (T_{11} \to T_{12})$ so if $|B'| = k'$ then $w(E', \Gamma) = k'$. A similar argument holds for the case of $T_2 \sim T_{11}$. □

### 4.4 The maximum type migration problem is in NP

Lemma 4.18. *The maximum type migration problem is in NP.*

Proof. The maximum type migration problem is in NP because we can reduce the maximum type migration problem to maximum satisfiability of general constraints (Theorem 4.12), which is in NP (Theorem 4.11). □

### 4.5 The maximum type migration problem is NP-hard

Recall that a directed degree-3 graph is a directed graph in which the total degree (in-degree plus out-degree) of every vertex is at most 3.

*Definition 4.19 (Maximum acyclic subgraph).* Given $(G, k)$ where $G$ is a directed degree-3 graph and $k$ is an integer, decide $\exists A : A$ is an acyclic subset of the edges of $G \ \wedge |A| \geq k$.

The maximum acyclic subgraph problem is NP-complete [Karp 1972; Newman 2001]. The maximum type migration problem is also NP-hard because we can reduce the maximum acyclic subgraph problem to the maximum type migration problem. The idea is to map a graph to a $\lambda$-term

in such a way that an acyclic subgraph corresponds to a way to annotate the $\lambda$-term with types. We begin with an example of this mapping and then give the proof.

Consider the following graph, which clearly is a directed degree-3 graph:


We proceed in six steps, as follows.

Our first step is to define a term-variable for each node: $source$, $middle$, $sink$.

Our second step is to define a $\lambda$-calculus expression for each node:

$$\begin{aligned} a_{source} &= source\ middle \\ a_{middle} &= middle\ sink \\ a_{sink} &= sink \end{aligned}$$

Our third step is to define a term-variable for each edge in the graph; we will call those term-variables $z_{source}$ and $z_{middle}$. The idea is that $z_{source}$ relates to the edge $(source, middle)$, while $z_{middle}$ relates to the edge $(middle, sink)$.

Our fourth step is to define a program. Specifically, suppose we are given a function from edges to types, that is, $t : \{z_{source}, z_{middle}\} \to Type$, we define the program

$$\begin{aligned} Exp(t) \quad = \quad & (\ \lambda source : t(z_{source}) \to \texttt{int}. \\ & \quad \lambda middle : t(z_{middle}) \to \texttt{int}. \\ & \quad \lambda sink : \texttt{int}. \\ & \qquad (\lambda x_{source} : \texttt{int}.\ \lambda x_{middle} : \texttt{int}.\ \lambda x_{sink} : \texttt{int}.\ 0)\ a_{source}\ a_{middle}\ a_{sink} \\ & ) \\ & (\lambda z_{source} : t(z_{source}).\ 0) \\ & (\lambda z_{middle} : t(z_{middle}).\ 0) \\ & 0 \end{aligned}$$

We use the notational convention that function application associates to the left; thus, $E_1\ E_2\ E_3$ means $(E_1\ E_2)\ E_3$.

The program $Exp(t)$ has a total of eight applications, namely three at the outer level, three at the inner level in the body of $\lambda sink$, and two in the definitions of $a_{source}, a_{middle}, a_{sink}$. Thus, a type derivation for $Exp(t)$ has eight uses of matching and eight uses of consistency. Let us consider all of those uses in turn.

Notice that for the three outer-level applications, we can use matching trivially. The reason is that each of the three outer-level applications is a beta-redex, and the type rule for $\lambda$-abstraction ensures that operator part has a function type.

Notice also that for the three outer-level applications, we can always use consistency trivially. To see that, let us examine $\lambda source : t(z_{source}) \to \texttt{int}$ and see that the argument is $(\lambda z_{source} : t(z_{source}).\ 0)$, which has type $(t(z_{source}) \to \texttt{int})$. So, we can have this trivial use of consistency: $(t(z_{source}) \to \texttt{int}) \sim (t(z_{source}) \to \texttt{int})$.

Now consider the three inner-level applications. For those, notice that each of them is a beta-redex, so also here the type rule for $\lambda$-abstraction ensures that operator part has a function type and we can use matching trivially.

Notice also that for the three inner-level applications, we can always use consistency trivially. To see that, let us examine the first of those applications, namely $((\lambda x_{source} : \texttt{int}.\ \ldots 0)\ a_{source})$. we have $(a_{source} = source\ middle)$, and $source$ has type $t(z_{source}) \to \texttt{int}$, so if $a_{source}$ type

checks, it will have type int, hence for the application $((\lambda x_{source} : \texttt{int}.\ \ldots\ 0)\ a_{source})$ we can use consistency trivially.

Now consider the two applications in the definitions of $a_{source}, a_{middle}, a_{sink}$. For example, let us consider $a_{source} = source\ middle$. Notice that *source* will be bound to the $\lambda$-abstraction $(\lambda z_{source} : t(z_{source}).\ 0)$, so for $a_{source}$ we can use matching trivially. Similarly, *middle* will be bound to the $\lambda$-abstraction $(\lambda z_{middle} : t(z_{middle}).\ 0)$, so also for $a_{middle}$ we can use matching trivially.

Thus we arrive at the center piece of the construction of $Exp(t)$, which is the question:

> How many trivial uses of consistency can we use for the two applications in the definitions of $a_{source}, a_{middle}, a_{sink}$?

The lower bound is 0 and the upper bound is 2, which is the number of edges in the graph. Indeed, recall that the graph has $n = 3$ nodes and $m = 2$ edges, and notice that across the $2 \times 8 = 16$ cases that we started with above, we have shown so far that in $4 * n + m = 14$ cases we can use matching or consistency trivially, while 2 cases need more work. To make progress on those two cases, which is also the highlighted question, we need $t$.

Our fifth step is to examine a fundamental possibility for $t$. Specifically, let $t^*$ be the function that defines $t^*(z_{source}) = \texttt{Dyn}$ and $t^*(z_{middle}) = \texttt{Dyn}$. Thus, we get:

$$\begin{array}{rcl} source & : & \texttt{Dyn} \to \texttt{int} \\ middle & : & \texttt{Dyn} \to \texttt{int} \\ sink & : & \texttt{int} \end{array}$$

Now we can see that $a_{source} = source\ middle$ will type check with this nontrivial use of consistency: $\texttt{Dyn} \sim (\texttt{Dyn} \to \texttt{int})$. Similarly, we can see that $a_{middle} = middle\ sink$ will type check with this nontrivial use of consistency: $\texttt{Dyn} \sim \texttt{int}$. Now we can rephrase the highlighted question above as follows: *can we define t such that we can use consistency trivially in one or both cases?* This question is the essence of the type migration problem for $Exp(t)$.

Our sixth step is to consider this particular function $t^A$, where

$$\begin{array}{rcl} t^A(z_{source}) & = & \texttt{int} \to \texttt{int} \\ t^A(z_{middle}) & = & \texttt{int} \end{array}$$

Thus, we get:

$$\begin{array}{rcl} source & : & (\texttt{int} \to \texttt{int}) \to \texttt{int} \\ middle & : & \texttt{int} \to \texttt{int} \\ sink & : & \texttt{int} \end{array}$$

We can see that those types make both (*source middle*) and (*middle sink*) type check and that with two trivial uses of consistency. Does it always work out that way? No, only for acyclic graphs, like our example graph. Once we consider graphs with cycles, type checking of $Exp(t)$ will require partially static types and nontrivial uses of consistency for applications that correspond to $a_{source}$ and $a_{middle}$. The reason is that our type system has only finite types and no recursive types. Thus, the construction of $Exp(t)$ creates a close correspondence between the notion of an acyclic subgraph and the notion of a finite type. Ultimately, the key to the construction of $Exp(t)$ is that the number of edges in a maximum acyclic subgraph is the same as the number of trivial uses of consistency in the type checking of the applications that correspond to $a_{source}$ and $a_{middle}$. In the example, this maximum acyclic subgraph is $A = \{(source, middle), (middle, sink)\}$, that is, the full set of edges in the original graph.

This completes the example of the mapping that we will use in the following proof of NP-hardness.

Lemma 4.20. *The maximum type migration problem is NP-hard.*

Proof. Let $(G, k)$ be an instance of the maximum acyclic subgraph problem. Let $\{v_1, \ldots, v_n\}$ be the nodes of $G$ and let $m$ be the number of edges in $G$. For each node $v_i$, fix an ordering of the outgoing edges and call the targets, in order, $w_{i1}, w_{i2}, \ldots, w_{im_i}$, where $m_i \leq 3$. Thus, each $w_{ij}$ is an element of $\{v_1, \ldots, v_n\}$. For each node $v_i$, define a term-variable $v_i$. (We are overloading the notation but the reader will always be able to tell from the context whether $v_i$ denotes a node or a term-variable.) For each node $v_i$, define the $\lambda$-calculus expression:

$$a_i \quad = \quad v_i \; w_{i1} \; \ldots \; w_{im_i}$$

This expression applies $v_i$ to the target of each of its outgoing edges. Given a function

$$t \quad : \quad \{z_{ij} \mid (v_i, w_{ij}) \text{ is an edge}\} \to \mathit{Type}$$

define the program

$$\begin{aligned} \mathit{Exp}(t) \quad = \quad & (\; \lambda v_1 : t(z_{11}) \to \ldots \to t(z_{1m_1}) \to \texttt{int}. \\ & \quad \ldots \\ & \quad \lambda v_n : t(z_{n1}) \to t(z_{n2}) \to t(z_{nm_n}) \to \texttt{int}. \\ & \qquad (\lambda x_1 : \texttt{int}. \; \ldots \lambda x_n : \texttt{int}. \; 0) \; a_1 \; \ldots \; a_n \\ & \;) \\ & (\lambda z_{11} : t(z_{11}). \; \ldots \; \lambda z_{1m_1} : t(z_{1m_1}). \; 0) \\ & \ldots \\ & (\lambda z_{n1} : t(z_{n1}). \; \ldots \; \lambda z_{nm_n} : t(z_{nm_n}). \; 0) \end{aligned}$$

Let $t^*$ be the function that defines $t^*(z_{ij}) = \texttt{Dyn}$, where $(v_i, w_j)$ is an edge.

Notice that $(\emptyset, \mathit{Exp}(t^*), 4n + m + k)$ is an instance of the maximum type migration problem (Definition 3.5). We will show that $(G, k)$ is solvable iff $(\emptyset, \mathit{Exp}(t^*), 4n + m + k)$ is solvable.

First, suppose $(G, k)$ has solution $A$. We will recursively define a function $t^A$ such that $\mathit{Exp}(t^A)$ solves $(\emptyset, \mathit{Exp}(t^*), 4n + m + k)$:

$$t^A(z_{ij}) \quad = \quad \begin{cases} t^A(z_{p1}) \to \ldots \to t^A(z_{pm_p}) \to \texttt{int} & \text{if } (v_i, w_{ij}) \in A \wedge w_{ij} = v_p \\ \texttt{Dyn} & \text{otherwise} \end{cases}$$

Intuitively, given a node $v_i$, this definition gives a type to an outgoing edge. In other words, this definition gives a type to one of arguments of the function for a node $v_i$. Given that $A$ forms an acyclic subgraph, $t^A$ maps every variable to a finite tree, and we can check easily that $\mathit{Exp}(t^*) \leq_\emptyset \mathit{Exp}(t^A) \wedge w(\mathit{Exp}(t^A), \emptyset) \geq 4n + m + k$.

Second, suppose $(\emptyset, \mathit{Exp}(t^*), 4n+m+k)$ has solution $\mathit{Exp}(t)$. Let $A$ be the subset of edges $(v_i, w_{ij})$, where $w_{ij} = v_p$, such that $t(z_{ij}) = t(z_{p1}) \to \ldots \to t(z_{pm_p})$. We have $|A| = w(\mathit{Exp}(t^A), \emptyset) - (4n + m) \geq k$. Given that $t$ maps every variable to a finite tree, $A$ is acyclic. □

## 4.6 Putting it all together

Now we can prove Theorem 4.1. First, Lemma 4.18 shows that the maximum type migration problem is in NP. Second, Lemma 4.20 shows that the maximum type migration problem is NP-hard. This completes the proof.

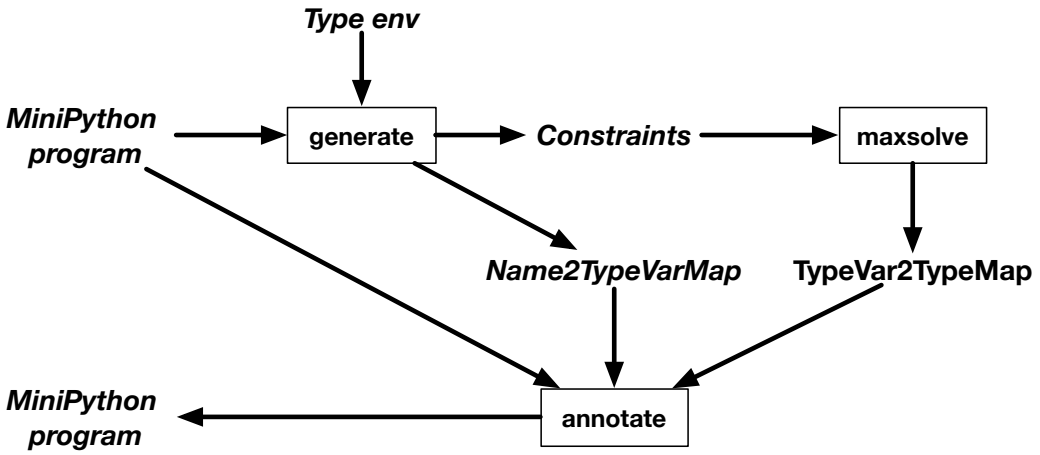


Fig. 6. Our type migrator for MiniPython.

## 5 IMPLEMENTATION

We have implemented our approach for a fragment of Python called MiniPython. We will submit our implementation to POPL's Artifact Evaluation.

*MiniPython.* MiniPython data includes Booleans, integers, floats, lists, sets, objects, and the value `None`. Lists and sets have types that can be partially static, such as `List(Dyn)`.

We based the design of MiniPython on three considerations. First, MiniPython accommodates four benchmark programs from the literature, after minor modification, none of which appear to type check with static types. Second, MiniPython programs are type checked in an initial environment with types for 26 built-in functions, imported functions, and standard constants and operators. Third, MiniPython uses syntax for type annotations that is valid both in Python 3 and in Reticulated Python. Specifically, a programmer can specify field types, parameter types, and return types. Reticulated Python and its implementation were immensely helpful during the design of MiniPython.

Appendix B of the supplementary material gives full details of MiniPython, including the grammar, the initial type environment, and the type system. Appendix B also explains the constraints, the constraint solver, and the constraint generator that we use for maximum type migration. Appendix B has largely the same structure as Section 4 and uses the same kinds of techniques. The main difference lies in the treatment of objects, particularly field access and method calls, as we explain below.

*Our implementation.* Figure 6 shows the design of our type migrator for MiniPython. The starting point consists of a MiniPython program and a type environment. The MiniPython program can have any type annotations; our type migrator will either make them more static or, in the worst case, leave them unchanged. The first step is to *generate* constraints, as described in Appendix B, and in the spirit of Section 4. Thus, we generate both hard constraints and soft constraints. Our approach associates type variables with names and expressions in the program, and we store that association as a Name2TypeVarMap. The second step is to solve all the hard constraints and a maximal number of the soft constraints, which we depict as *maxsolve*. We make no attempt to

solve every subset of the soft constraints. Instead, we use an approximation algorithm that combines heuristic search and constraint solving. Specifically, we pick a pseudo-random sequence of subsets of the soft constraints, and we use constraint solving to solve the hard constraints together with each of those subsets of soft constraints. We store the solution for the largest solvable subset as a TypeVar2TypeMap. The third step is to *annotate* the program, based on the the input program, the Name2TypeVarMap, and the TypeVar2TypeMap. The result is a program that type checks and has more static type annotations than the input program (or the same type annotations).

*The challenge of objects.* The objects in MiniPython present a particular challenge that is absent in the gradually typed $\lambda$-calculus. We will explain this challenge via comparison of method calls in MiniPython and function calls in the gradually typed $\lambda$-calculus.

First, consider a function call $(F\ G)$ in the gradually typed $\lambda$-calculus. One of the constraints that we will generate from $(F\ G)$ is the *hard* constraint $[\![F]\!] \rhd \langle G \rangle \to [\![FG]\!]$. From this constraint, we generate the *soft* constraint $[\![F]\!] = \langle G \rangle \to [\![FG]\!]$. Intuitively, this soft constraint says that if $[\![F]\!]$ must be assigned a type other than Dyn, then that type must be a function type. The key here is the unique choice embodied in the implication that "not Dyn implies function type".

Second, consider a method call $e.m()$ in MiniPython. One of the constraints that we will generate from $e.m()$ is the *hard* constraint $[\![e]\!] \rhd_m () \to [\![e.m()]\!]$. From this constraint, we generate the *soft* constraint $[\![e]\!] \in [C_1, \ldots, C_n]$, where $C_1, \ldots, C_n$ are the classes that each has a method $m$ with the correct number of parameters. Intuitively, this soft constraint says that if $[\![e]\!]$ must be assigned a type other than Dyn, then that type must *one of* $C_1, \ldots, C_n$. The key here is the *lack* of a unique choice embodied in the implication that "not Dyn implies one of $C_1, \ldots, C_n$". In some cases, this creates no new challenge because constraint solving will determine a particular choice of type for $[\![e]\!]$. However, in other cases, it doesn't. For some of our benchmark programs, this lack of choice happens sufficiently often that trying all possibilities is infeasible. So, our implementation for MiniPython has *two* choices to make. We must both chose which subset of soft constraints to satisfy and chose which type to assign to receiver expressions (in some cases).

*Lines of code.* Our implementation is about 9,000 lines of Java plus some generated code. Our implementation is a proof of concept and likely it can be optimized substantially. We will now show an evaluation on four benchmark programs.

## 6 EXPERIMENTAL RESULTS

The goal of this section is to present an empirical evaluation of our tool that supports the following three claims.

*Claims.*

(1) Our tool produces worthwhile type migration for our microbenchmarks.
(2) The migrated programs all type check with our gradual type checker.
(3) The migration time for each microbenchmark is at most five minutes.

*Benchmarks.* Our benchmark suite consists of four MiniPython programs with a total of 166 lines of code. We extracted those microbenchmarks from larger original versions that were used for experiments reported in the papers [Greenman and Migeed 2018; Vitousek et al. 2017]. Specifically, we extracted micro_chaos and micro_chaos2 from chaos, we extracted micro_go from go, and we extracted micro_richards from richards. Note that micro_chaos and micro_chaos2 are quite different programs with almost nothing in common; they just happen to both stem from chaos.

A key property of each program is that it has eluded our best efforts to make it type check with static types. In particular, we ran the Typpete type inferencer on each program and found that each program fails to type check. Thus, our programs are fundamentally different from the ones used in two recent papers on performance aspects of gradual typing [Bauman et al. 2017; Takikawa et al. 2016]; they use benchmark programs that type check with *static* types.

Another key property is that all four of our programs use lists or sets. The following table presents static measurements of each program.

| Program | LOC | #classes | #methods | #params | #fields | ***Total*** |
|---|---|---|---|---|---|---|
| micro_chaos | 34 | 3 | 3 | 10 | 2 | 15 |
| micro_chaos2 | 43 | 4 | 4 | 10 | 2 | 16 |
| micro_go | 52 | 3 | 4 | 9 | 5 | 18 |
| micro_richards | 37 | 3 | 3 | 8 | 2 | 13 |
| ***Total*** | 166 | 13 | 14 | 37 | 11 | 62 |

For each program, the total in the right-most column (***Total***) is number of program points that need a type annotation. This total is the sum of the columns #methods, #params, and #fields. Notice that #methods is part of the sum because each method has a return type.

We believe that our microbenchmarks present a big challenge for type migration.

*Platform.* We ran our experiments on an Intel Core i7-6700K at 4.00Ghz with 16 GB of RAM.

*Measurements.* The following table shows the number of type variables and the number of hard and soft constraints after simplification.

| | | #constraints | |
|---|---|---|---|
| Program | #type var. | hard | soft |
| micro_chaos | 61 | 65 | 25 |
| micro_chaos2 | 71 | 66 | 27 |
| micro_go | 97 | 115 | 70 |
| micro_richards | 58 | 61 | 30 |

The following table shows results from our migration tool.

| Program | soft constraints solved | | annot. not Dyn | | #partially static type annotations | migration time |
|---|---|---|---|---|---|---|
| micro_chaos | 18 | (72%) | 9 | (60%) | 1 | 5 min |
| micro_chaos2 | 25 | (93%) | 11 | (69%) | 1 | 5 min |
| micro_go | 56 | (80%) | 14 | (78%) | 1 | 5 min |
| micro_richards | 28 | (93%) | 10 | (77%) | 2 | 5 min |
| *Weighted average* | | (84%) | | (71%) | | 5 min |

The resulting programs all type check with our gradual type checker. Indeed, we used our migration tool as a gradual type checker in the following way: given a program, we solve only the *hard* constraints.

*Assessment.* Our approach to maximum type migration is based on the idea of solving a maximum number of *soft* constraints. Our tool solved an average of 84% of the soft constraints, which we believe is the best or close to the best that can be achieved for those programs.

Our tool can produce both static and partially static annotations, and in the migrated programs from our tool, 71% of the annotations are not `Dyn`. We are unable to compare with the tool of Campora et al. [Campora et al. 2018] because their tool works with Haskell.

A key capability of our tool is that it can produce partially static type annotations. For the three benchmark programs that use lists and/or sets, our tool produced a total of 5 partially static type annotations. Specifically, our tool produced 4 annotations of the form `List(Dyn)` and 1 annotation of the form `Set(Dyn)`. Note that in MiniPython, local variables have no type annotations so some partially static types may "hide beneath the surface."

We gave our tool five minutes to do each migration. Our tool gives good results already after 2 seconds and it gives great results after five minutes. Our tool is unoptimized and has room for performance improvement. For comparison, the type inference tool Typpete [Hassan et al. 2018] uses an off-the-shelf MaxSMT-solver to solve constraints; this could be an avenue for future work on type migration. Our hope is that a highly optimized MaxSMT-solver can help make our current implementation more scalable.

## 7 RELATED WORK

We will discuss three categories of related work, namely type migration, type inference, and gradual typing design.

*Type migration.* Siek and Vachharajani [Siek and Vachharajani 2008] presented the first algorithm for type migration with gradual types. Their starting point was the type system by Siek and Taha [Siek and Taha 2006], which they restricted significantly to enable a unification-based algorithm. Their paper proved correctness but had no report on experiments. In contrast, our paper shows how to do type migration for the full type system by Siek and Taha [Siek and Taha 2006] (as reformulated by Cimini and Siek [Cimini and Siek 2016]). Later, Garcia and Cimini [Garcia and Cimini 2015] presented a new migration algorithm a type system that type checks the same programs as the type system by Siek and Vachharajani [Siek and Vachharajani 2008]. Their approach limits the algorithm to add only static types. In contrast, our algorithm can add partially static types.

Rastogi, Chaudhuri, and Hosmer [Rastogi et al. 2012] presented a migration algorithm for an object-oriented language with subtyping. They proved a correctness theorem that says that the added types cannot cause new run-time failures. In contrast, we prove the stronger theorem that a migrated program type checks with a gradual type checker.

Recently, Campora, Chen, Erwig, and Walkingshaw [Campora et al. 2018] presented a migration algorithm for the type system by Siek and Taha [Siek and Taha 2006], restricted such that every parameter type must be either fully static or fully dynamic (like Siek and Vachharajani [Siek and Vachharajani 2008]). Our migration algorithm handles the type system by Siek and Taha [Siek and Taha 2006], without restrictions. Both Campora et al.'s approach and our approach are based on constraint solving. A major difference lies in the nature of the constraints: their constraints are for a combination of variational types and gradual types, while our approach uses a combination of hard and soft constraints.

*Type inference.* Type inference has a stricter goal than type migration. Specifically, type inference aims to infer static types for all variables, while type migration aims to add as many types as possible. A recent example is the inference tool for Python by Hassan, Urban, Eilers, and Muller [Hassan et al. 2018]. We showed results from an experiment with this tool in Section 2. Another recent example is the inference tool for Dart by Heinze, Møller, and Strocco [Heinze et al. 2016].

Some approaches to type inference add types for the purpose of program understanding but without a guarantee that the resulting program type checks. A recent example is the inference tool for Python by Xu, Zhang, Chen, Pei, and Xu [Xu et al. 2016].

*Gradual typing design.* Researchers have explored many extensions and variations of gradual types. Those approaches can inspire new algorithms for type migration. For example, future work could pursue type migration that supports subtyping [Garcia et al. 2016; Siek and Taha 2007; Vitousek et al. 2014], refinement types [Lehmann and Tanter 2017], and monotonic references [Siek et al. 2015b]. Another direction is to implement new migration tools for successful gradually typed languages such as Hack and Flow.

## 8 CONCLUSION

We have shown that a natural notion of maximum type migration is NP-complete and that our implementation for a subset of Python gives good results. Our paper is the first to do type migration for the full calculus of Siek and Taha [Siek and Taha 2006] without restrictions. Our techniques are general and apply to other gradually typed languages.

We will submit our implementation to POPL's Artifact Evaluation and we hope that our four Python benchmark programs can be a challenge for the community. Specifically, can new approaches to maximum type migration produce more static types than our tool? For example, new approaches may use better approximation algorithms or more powerful type systems with such ideas as polymorphism, subtyping, and overloading.

## REFERENCES


Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: Only Mostly Dead. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 54:1–54:24.

Stephen L. Bloom, Calvin C. Elgot, and Jesse B. Wright. 1980. Solutions of the Iteration Equation and Extensions of the Scalar Iteration Operation. *SIAM J. Comput* 9, 1 (1980), 25–45.

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018. Migrating Gradual Types. *Proc. ACM Program. Lang.* 2, POPL (2018), 15:1–15:29.

Matteo Cimini and Jeremy Siek. 2016. The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems. In *Proceedings of POPL'16, ACM Symposium on Principles of Programming Languages*.

Bruno Courcelle. 1983. Fundamental Properties of Infinite Trees. *Theoretical Computer Science* 25, 1 (1983), 95–169.

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 303–315.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 429–442.

Ben Greenman and Zeina Migeed. 2018. On the cost of Type-Tag Soundness. In *PEPM*.

Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-Based Type Inference for Python 3. In *Proceedings of CAV'18, Computer-Aided Verification*.

Thomas S. Heinze, Anders Møller, and Fabio Strocco. 2016. Type Safety Analysis for Dart. In *DLS*.

Richard M. Karp. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, R. Miller and J. Thatcher (Eds.). Plenum Press, 85–103.

Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. 1994. Efficient Inference of Partial Types. 49, 2 (1994), 306–324. Preliminary version in Proceedings of FOCS'92, 33rd IEEE Symposium on Foundations of Computer Science, pages 363–371, Pittsburgh, Pennsylvania, October 1992.

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 775–788.

Alantha Newman. 2001. The Maximum Acyclic Subgraph Problem and Degree-3 Graphs. In *RANDOM-APPROX*. 147–158.

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. *SIGPLAN Not.* 47, 1 (2012), 481–494.

Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*. 2–27.

Jeremy G. Siek and Ronald Garcia. 2012. Interpretations of the Gradually-typed Lambda Calculus. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*. 68–80.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*. 81–92.

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*. 7:1–7:12.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015a. Refined Criteria for Gradual Typing. In *SNAPL*. 274–293.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic References for Efficient Gradual Typing. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*. 432–456.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 456–468.

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. *SIGPLAN Not.* 50, 2 (2014), 45–56.

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems. *SIGPLAN Not.* 52, 1 (2017), 762–774.

Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 607–618.

## A PROOFS OF FOUR LEMMAS

Lemma 4.13. $\forall \varphi, \Gamma, E, T$: if $\varphi \geq \Gamma$ and $FV(E) \subseteq Dom(\Gamma)$ and $\varphi \vdash E : T$, then $\Gamma \vdash E : T$.

Proof. Straightforward by induction on the derivation of $\varphi \vdash E : T$. □

Lemma 4.14. $\forall \varphi, \Gamma$:

(1) *If* $\varphi \models B(\lambda x : S.F, \Gamma)$, *then* $\varphi \models B(F, \{(x : \varphi(x))\} \cup \Gamma)$.
(2) *If* $\varphi \models B(E_1 E_2, \Gamma)$, *then* $\varphi \models B(E_1, \Gamma)$ *and* $\varphi \models B(E_2, \Gamma)$.

Proof. For *1*, we have $\varphi \models \varphi(\lambda x : S.F, \Gamma)$ which implies that $\varphi \models [\![\lambda x : S.F]\!] = x \rightarrow [\![F]\!], S \sqsubseteq x$. Then for some $\Gamma'$, and for the occurrence $F$ in $\lambda x : S.F$, $\varphi$ must also be a solution of $B(F, \Gamma')$. If $F$ contains an occurrence of $x$ then we will have $x$ as a free variable in $F$ so $\varphi$ must also be a solution for $x$. So we have $\Gamma' = \{(x : \varphi(x))\} \cup \Gamma$. So $\varphi \models B(F, \{(x : \varphi(x))\} \cup \Gamma)$.
Next, *2* Follows directly from our constraint system. □

Lemma 4.15. $\forall \varphi, E, \Gamma$: *if* $\varphi \models B(E, \Gamma)$ *and* $\varphi \geq \Gamma$, *then* $\varphi \vdash G_\varphi(E) : \varphi([\![E]\!])$.

Proof. We proceed by induction on the structure of $E$.
Case: $n$. From *T-Int*, $\varphi \vdash n : Int$. We have $\varphi \models [\![n]\!] = Int$. Therefore, $\varphi([\![n]\!]) = Int$. Then $\varphi \vdash n : \varphi([\![n]\!])$. From properties of $G_\varphi$, we have $G_\varphi(n) = n$ so $\varphi \vdash G_\varphi(n) : \varphi([\![n]\!])$.
Case: $True$, $False$. Similar to case $n$.
Case: $x$. We consider two subcases. Suppose $x$ is a bound variable. $\varphi \models [\![x]\!] = x$. Then $\varphi(x) = \varphi([\![x]\!])$. Therefore $(x : \varphi(x)) \in \varphi$ So we can invoke *T-Var* as follows:

$$\frac{x : \varphi(x) \in \varphi}{\varphi \vdash x : \varphi(x)} \ (T\text{-}Var)$$

So we have $\varphi \vdash x : \varphi(x)$. Then $\varphi \vdash x : \varphi([\![x]\!])$. From properties of $G_\varphi$, we have $G_\varphi(x) = x$ so $\varphi \vdash G_\varphi(x) : \varphi([\![x]\!])$. Next, suppose $x$ is a free variable. Then $\varphi \models [\![x]\!] = \Gamma(x)$. Then $\Gamma(x) = \varphi([\![x]\!])$. Since $\varphi \geq \Gamma$ then $\Gamma(x) = \varphi(x)$. Then $\varphi(x) = \varphi([\![x]\!])$. So we have $(x : \varphi(x)) \in \varphi$ and we can invoke *T-Var* as follows:

$$\frac{x : \varphi(x) \in \varphi}{\varphi \vdash x : \varphi(x)} \ (T\text{-}Var)$$

So we have $\varphi \vdash x : \varphi(x)$. Then $\varphi \vdash x : \varphi([\![x]\!])$. By properties of $G_\varphi$, we have $G_\varphi(x) = x$ so $\varphi \vdash G_\varphi(x) : \varphi([\![x]\!])$.
Case: $\lambda x : S.F$. Given that $\varphi \models B(\lambda x : S.F, \Gamma)$ then by Lemma 4.14, $\varphi \models B(F, (x : \varphi(x)), \Gamma)$. Since $\varphi \geq \Gamma$ and $\varphi = \{(x : \varphi(x))\} \cup \varphi$ because $(x : \varphi(x)) \in \varphi$, then $\varphi \geq \{(x : \varphi(x)) \cup \Gamma\}$. Therefore, we can apply our induction hypothesis to get: $\varphi \vdash G_\varphi(F) : \varphi([\![F]\!])$. By properties of $G_\varphi$, we have $G_\varphi(\lambda x : S.F) = \lambda x : \varphi(x).G_\varphi(F)$ and from $\varphi \models B(\lambda x : S.F, \Gamma)$ we have $\varphi([\![\lambda x : S.F]\!]) = \varphi(x) \rightarrow \varphi([\![F]\!])$. Since, we have that $\varphi \vdash G_\varphi(F) : \varphi([\![F]\!])$ and $\varphi = \{(x : \varphi(x))\} \cup \varphi$, we can invoke *T-Abs* as follows:

$$\frac{\varphi, x : \varphi(x) \vdash G_\varphi(F) : \varphi([\![F]\!])}{\varphi \vdash \lambda x : \varphi(x).G_\varphi(F) : \varphi(x) \rightarrow \varphi([\![F]\!])} \ (T\text{-}Abs)$$

Thus, we have concluded that $\varphi \vdash \lambda x : \varphi(x).G_\varphi(F) : \varphi(x) \rightarrow \varphi([\![F]\!])$. Recall that $\varphi \models B(\lambda x : S.F, \Gamma)$. Then $\varphi \models [\![\lambda x : S.F]\!] = x \rightarrow [\![F]\!]$. So we have $\varphi \vdash G_\varphi(\lambda x : S.F) : \varphi([\![\lambda x : S.F]\!])$ as required.
Case: $E_1 E_2$. Given that $\varphi \models B(E_1 E_2, \Gamma)$. Then by Lemma 4.14, $\varphi \models B(E_1, \Gamma)$ and $\varphi \models B(E_2, \Gamma)$. Furthermore, $\varphi \models [\![E_1]\!] \triangleright \langle E_2 \rangle \rightarrow [\![E_1\ E_2]\!], \langle E_2 \rangle \sim [\![E_2]\!]$. So we can invoke our induction hypothesis

and conclude that:
(*i*) $\varphi \vdash G_\varphi(E_1) : \varphi([\![E_1]\!])$
(*ii*) $\varphi \vdash G_\varphi(E_2) : \varphi([\![E_2]\!])$

Now can construct:

$$\frac{\begin{array}{cc}\varphi \vdash G_\varphi(E_1) : \varphi([\![E_1]\!]) & \varphi \vdash G_\varphi(E_2) : \varphi([\![E_2]\!]) \\ \varphi([\![E_1]\!]) \triangleright \varphi(\langle E_2 \rangle) \to \varphi([\![E_1E_2]\!])) & \varphi(\langle E_2 \rangle) \sim \varphi([\![E_2]\!])\end{array}}{\varphi \vdash G_\varphi(E_1)G_\varphi(E_2) : \varphi([\![E_1E_2]\!])} \text{(T-App)}$$

Therefore, $\varphi \vdash G_\varphi(E_1E_2) : \varphi([\![E_1E_2]\!])$. □

Lemma 4.16. *$\forall E, \Gamma, E'$: if $E \leq_\Gamma E'$, then exactly one of the following holds:*

(1) *$E = n$ and $E' = n$.*
(2) *$E = True$ and $E' = True$.*
(3) *$E = False$ and $E' = False$.*
(4) *$E = x$ and $E' = x$.*
(5) *$E = \lambda x : S.F$ and $E' = \lambda x : S'.F'$ and $F \leq_{\{(x:S)\}\cup\Gamma} F'$*
(6) *$E = E_1E_2$ and $E' = E_1'E_2'$ and $E_1 \leq_\Gamma E_2'$ and $E_1 \leq_\Gamma E_2'$.*

Proof. Case 1-4. Straightforward.

Case: 5. Suppose $E = \lambda x : S.F$. Then from the fact that $E \sqsubseteq E'$ and *P-Abs*, $E'$ must be of the form $\lambda x : S'.F'$. From Definition 3.4 and $E \leq_\Gamma E'$, we have $FV(\lambda x : S.F) \subseteq Dom(\Gamma)$, $\Gamma \vdash \lambda x : S'.F' : T'$ and $\lambda x : S.F \sqsubseteq \lambda x : S'.F'$. From $FV(\lambda x : S.F) \subseteq Dom(\Gamma)$ we have $FV(F) \subseteq Dom((x, S), \Gamma)$.

Since $\Gamma \vdash \lambda x : S'.F' : T'$, the last rule used in the derivation must have been *T-Abs* so we have:

$$\frac{\Gamma, x : S' \vdash F' : U'}{\Gamma \vdash \lambda x : S'.F' : S' \to U'} \textit{ (T-Abs)}.$$

We know that $S \sqsubseteq S'$, because $\lambda x : S.F \sqsubseteq \lambda x : S'.F'$. From $\lambda x : S.F \sqsubseteq \lambda x : S'.F'$ we also know that $F \sqsubseteq F'$. In summary, we have shown the following:
1- $FV(F) \subseteq Dom((x, S), \Gamma)$
2- $\Gamma', (x : S') \vdash F' : U'$
3- $F \sqsubseteq F'$
Then from Definition 3.4, we have $F \leq_{\{(x:S)\}\cup\Gamma} F'$.

Case: 6. Suppose $E = E_1E_2$. From the fact that $E \sqsubseteq E'$ and *P-App*, we have $E' = E_1'E_2'$. Without loss of generality, consider $E_1$. We have $E_1 \sqsubseteq E_1'$. From our proposition, $FV(E_1E_2) \subseteq Dom(\Gamma)$ and $\Gamma \vdash E_1'E_2' : T'$. The last rule in the derivation of $\Gamma \vdash E_1'E_2' : T'$ must have been *T-App*. So we have:

$$\frac{\begin{array}{cc}\Gamma \vdash E_1' : T_1' & \Gamma \vdash E_2' : T_2' \\ T_1' \triangleright T_2'' \to T' & T_2' \sim T_2''\end{array}}{\Gamma \vdash E_1'E_2' : T'} \textit{ (T-App)}$$

Then we have $\Gamma \vdash E_1' : T_1'$. We also know that $E_1 \sqsubseteq E_1'$. From $FV(E_1E_2) \subseteq Dom(\Gamma)$ we have that $FV(E_1) \subseteq Dom(\Gamma)$. Therefore $E_1 \leq_\Gamma E_1'$. In summary, we have shown the following:
1-$FV(E_1) \subseteq Dom(\Gamma)$
2-$\Gamma \vdash E_1' : T_1'$
3-$E_1 \sqsubseteq E_1'$
Then from Definition 3.4, we have $E_1 \leq_\Gamma E_1'$ and $E_2 \leq_\Gamma E_2'$. □

# B MINIPYTHON

In this appendix we give full details of MiniPython, including its syntax, initial type environment, and type system. We also explain the constraints, the constraint solver, and the constraint generator that we use for maximum type migration.

## B.1 Syntax

We use the following metavariables that range over disjoint subsets of identifiers:

$$\begin{array}{rcl} F & : & \text{Predefined or imported } \mathit{FunctionName} \\ M & : & \mathit{ModuleName} \\ C & : & \mathit{ClassName} \\ f & : & \mathit{FieldName} \\ m, p & : & \mathit{MethodName} \\ x, y & : & \mathit{ParameterName} \text{ or } \mathit{LocalName} \end{array}$$

We write $\overline{I}$ as a shorthand for a possible empty sequence $I_1 \ldots I_n$ (and similarly for $\overline{CD}$, $\overline{D}$, $\overline{S}$, etc) and write $\overline{e}$ as a shorthand for $e_1, \ldots, e_n$ (with commas). Additionally, we write $\overline{x : t}$ as a shorthand for for $x_1 : t_1, \ldots, x_n : t_n$, where $n$ is the length of $\overline{x : t}$ (a similarly for $\overline{f : t}$).

Here is the MiniPython abstract syntax:

$$\begin{array}{rrcl} \text{(Program)} & P & ::= & \overline{I}\ \overline{CD}\ \texttt{print}(C().m(\overline{e})) \\ \text{(Import)} & I & ::= & \texttt{from}\ M\ \texttt{import}\ F \\ \text{(Class Def.)} & CD & ::= & \texttt{@fields}\ (\{\overline{f : t}\})\ \texttt{class}\ C(\texttt{object}) : \mathit{Init}\ \overline{D} \\ & \mathit{Init} & ::= & \texttt{def}\ \texttt{\_\_init\_\_}(\overline{x : t}) : \overline{S} \\ \text{(Def)} & D & ::= & \texttt{def}\ m(\overline{x : t})\ \texttt{->}\ t : \overline{S} \\ \text{(Type)} & t, u & ::= & \texttt{Dyn} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{float} \mid \texttt{List}(t) \mid \texttt{Set}(t) \mid C \\ \text{(Statement)} & S & ::= & \texttt{pass} \mid x = e \mid x.f = e \mid x[e] = e \mid \\ & & & \texttt{if}\ e : \overline{S}\ \texttt{else}: S \mid \texttt{while}\ e : \overline{S}\ \texttt{else}: S \mid \texttt{for}\ x\ \texttt{in}\ e : \overline{S}\ \texttt{else}: S \mid \\ & & & \texttt{raise}\ C() \mid \texttt{break} \mid \texttt{print}(e) \mid \texttt{return}\ e \\ & & & e.\texttt{append}(e) \mid e.\texttt{pop}(\,) \mid e.\texttt{add}(e) \mid e.\texttt{clear}(\,) \\ \text{(Expression)} & e & ::= & F(\overline{e}) \mid x \mid \texttt{list}(\,) \mid \texttt{set}(\,) \mid C(\overline{e}) \mid e.f \mid e.m(\overline{e}) \mid \\ & & & \texttt{isinstance}(e, C) \mid [\overline{e}] \mid e[e] \mid \texttt{None} \mid \texttt{True} \mid \texttt{False} \mid \\ & & & \mathit{IntConst} \mid \mathit{HexConst} \mid \mathit{FloatConst} \end{array}$$

Notice the three statements that contain `else`: $S$. Those `else`-clauses contain a single statement to make parsing easy.

We assume that sequences of field declarations, method declarations, and parameter names contain no duplicate names. We disallow assignment to formal parameters.

### B.2 Initial type environment $\Gamma$

Built-in functions:

$$
\begin{aligned}
\texttt{divmod} &: (\texttt{int} \times \texttt{int}) \to \texttt{Dyn} \\
\texttt{int} &: \texttt{float} \to \texttt{int} \\
\texttt{float} &: \texttt{int} \to \texttt{float} \\
\texttt{len} &: \texttt{List(Dyn)} \to \texttt{int} \\
\texttt{list} &: \texttt{Dyn} \to \texttt{List(Dyn)} \\
\texttt{max} &: \texttt{List(float)} \to \texttt{float} \\
\texttt{min} &: \texttt{List(float)} \to \texttt{float} \\
\texttt{range} &: \texttt{int} \to \texttt{Dyn} \\
\texttt{set} &: \texttt{Dyn} \to \texttt{Set(Dyn)}
\end{aligned}
$$

Imported functions:

$$
\begin{aligned}
\texttt{random} &: () \to \texttt{float} \\
\texttt{randrange} &: \texttt{int} \to \texttt{int}
\end{aligned}
$$

Standard constants and operators:

$$
\begin{aligned}
\texttt{not} &: \texttt{bool} \to \texttt{bool} \\
\\
\texttt{and} &: (\texttt{bool} \times \texttt{bool}) \to \texttt{bool} \\
\texttt{or} &: (\texttt{bool} \times \texttt{bool}) \to \texttt{bool} \\
\texttt{==} &: (\texttt{int} \times \texttt{int}) \to \texttt{bool} \\
\texttt{is} &: (\texttt{Dyn} \times \texttt{Dyn}) \to \texttt{bool} \\
< &: (\texttt{int} \times \texttt{int}) \to \texttt{bool} \\
+ &: (\texttt{int} \times \texttt{int}) \to \texttt{int} \\
- &: (\texttt{int} \times \texttt{int}) \to \texttt{int} \\
// &: (\texttt{int} \times \texttt{int}) \to \texttt{int} \\
* &: (\texttt{float} \times \texttt{float}) \to \texttt{float} \\
/ &: (\texttt{float} \times \texttt{float}) \to \texttt{float} \\
** &: (\texttt{float} \times \texttt{float}) \to \texttt{float} \\
\& &: (\texttt{int} \times \texttt{int}) \to \texttt{int} \\
\uparrow &: (\texttt{int} \times \texttt{int}) \to \texttt{int} \\
\texttt{in} &: (\texttt{Dyn} \times \texttt{Set(Dyn)}) \to \texttt{bool}
\end{aligned}
$$

## B.3 Type System

*B.3.1 Helper functions.* We use a function *abstract* to abstract a sequence $\overline{CD}$ of class definitions to a map $\Psi$ of the following functionality.

$$\begin{array}{rcl} \Psi & : & (ClassName \to (\overline{Type}) \quad \wedge \\ & & (ClassName \times FieldName) \to Type \quad \wedge \\ & & (ClassName \times MethodMame) \to ((\overline{Type}) \to Type) \end{array}$$

The notation $\wedge$ means that the function $\Psi$ is overloaded. Specifically, if $C$ is a class name, then $\Psi(C)$ is the type of the initializer, $\Psi(C, f)$ is the type of the field $f$, and $\Psi(C, m)$ is the type of the method $m$. We define *abstract* as follows. For a single class definition $CD$, where

$$\begin{array}{rcl} CD & = & (\texttt{@fields}\ (\{\overline{f : t}\})\ \texttt{class}\ C(\texttt{object}) : Init\ \overline{D}) \\ Init & = & \texttt{def}\ \texttt{\_\_init\_\_}(\overline{x : t}) : \overline{S} \end{array}$$

and method $m_i$ has definition

$$\texttt{def}\ m_i(\overline{x : t})\ \texttt{->}\ \ t : \overline{S}$$

we have:

$$\begin{array}{rcl} abstract(CD) & = & \Psi \text{ where} \\ & & \qquad \begin{array}{rcl} \Psi(C) & = & (\overline{x : t}) \\ \Psi(C, f_i) & = & t_i \\ \Psi(C, m_i) & = & ((\overline{t}) \to t) \end{array} \end{array}$$

For a sequence $\overline{CD}$ of class definitions we define:

$$abstract(\overline{CD}) \quad = \quad \bigcup_i abstract(CD_i)$$

where we use the notation $\cup$ to denote the union of functions with disjoint domains.

Notice that $\Psi$ says nothing about local variables; we handle those differently.

Next we define a helper function that from a statement collects the local variables that it assigns. We assume that those names differ from the parameters of the initializer or method definition in which the statement occurs. Additionally, we assume that those names differ from the names defined in for-statements of the method.

| $S$ | : | $locals(S)$ |
|---|---|---|
| `pass` | : | $\emptyset$ |
| $x = e$ | : | $\{x\}$ |
| $x.f = e$ | : | $\emptyset$ |
| $x[e_2] = e_3$ | : | $\emptyset$ |
| `if` $e$`:` $\overline{S}$ `else:` $S$ | : | $locals(\overline{S}) \cup locals(S)$ |
| `while` $e$`:` $\overline{S}$ `else:` $S$ | : | $locals(\overline{S}) \cup locals(S)$ |
| `for` $x$ `in` $e$`:` $\overline{S}$ `else:` $S$ | : | $locals(\overline{S}) \cup locals(S)$ |
| `raise` $C()$ | : | $\emptyset$ |
| `break` | : | $\emptyset$ |
| `print`$(e)$ | : | $\emptyset$ |
| `return` $e$ | : | $\emptyset$ |
| $e_1$`.append(`$e_2$`)` | : | $\emptyset$ |
| $e$`.pop( )` | : | $\emptyset$ |
| $e_1$`.add(`$e_2$`)` | : | $\emptyset$ |
| $e$`.clear( )` | : | $\emptyset$ |

For a sequence $\overline{S}$, we define

$$locals(\overline{S}) \quad = \quad \bigcup_i locals(S_i)$$

*B.3.2 Consistency.*

$$\begin{array}{rcl} \texttt{bool} & \sim & \texttt{bool} \\ \texttt{int} & \sim & \texttt{int} \\ \texttt{float} & \sim & \texttt{float} \\ C & \sim & C \\ t & \sim & \texttt{Dyn} \\ \texttt{Dyn} & \sim & t \end{array}$$

$$\frac{t \sim t'}{\texttt{List}(t) \sim \texttt{List}(t')}$$

$$\frac{t \sim t'}{\texttt{Set}(t) \sim \texttt{Set}(t')}$$

*B.3.3 Matching.* The following rules for matching play key roles in later type rules for statements and expressions that involve lists, sets, and objects. The idea of $\triangleright list$ is to enable an expression to have either the type $\texttt{List}(t)$ or the type Dyn. If the type is Dyn, then the matching rule below enables us to treat the type of the expression as if it was $\texttt{List}(\texttt{Dyn})$.

Similarly, the idea of $\triangleright_{Set}$ is to enable an expression to have either the type $\texttt{Set}(t)$ or the type Dyn. If the type is Dyn, then the matching rule below enables us to treat the type of the expression as if it was $\texttt{Set}(\texttt{Dyn})$.

The idea of $\triangleright^{\Psi}_{init}$ is to map from a class name to the type of the initializer of that class.

Finally, the idea of $\triangleright^{\Psi}_{f}$ and $\triangleright^{\Psi}_{m}$ is to enable an expression to have either the type $C$ or the type Dyn. The matching rules for those cases work differently than the ones for lists and sets. Here the idea of $\triangleright^{\Psi}_{f}$ is to find the type of a field, while the idea of $\triangleright^{\Psi}_{m}$ is to find the type of a method. If the type is Dyn, then the matching rule for $\triangleright^{\Psi}_{f}$ below enables us to treat an expression as if it produces an object with a field $f$ of type Dyn. Similarly, the matching rule for $\triangleright^{\Psi}_{m}$ below enables us to treat an expression as if it produces an object with a method $m$ of type $((\overline{\texttt{Dyn}}) \to \texttt{Dyn})$.

$$
\begin{array}{rcl}
\texttt{List}(t) & \triangleright_{List} & \texttt{List}(t) \\
\texttt{Dyn} & \triangleright_{List} & \texttt{List}(\texttt{Dyn}) \\
\texttt{Set}(t) & \triangleright_{Set} & \texttt{Set}(t) \\
\texttt{Dyn} & \triangleright_{Set} & \texttt{Set}(\texttt{Dyn}) \\
C & \triangleright^{\Psi}_{init} & \Psi(C) \\
C & \triangleright^{\Psi}_{f} & \Psi(C, f) \\
\texttt{Dyn} & \triangleright^{\Psi}_{f} & \texttt{Dyn} \\
C & \triangleright^{\Psi}_{m} & \Psi(C, m) \\
\texttt{Dyn} & \triangleright^{\Psi}_{m} & ((\overline{\texttt{Dyn}}) \to \texttt{Dyn})
\end{array}
$$

*B.3.4 Precision.*

$$
\begin{array}{rcl}
\texttt{Dyn} & \sqsubseteq & t \\
t & \sqsubseteq & t
\end{array}
$$

$$
\frac{t \sqsubseteq t'}{\texttt{List}(t) \sqsubseteq \texttt{List}(t')}
$$

$$
\frac{t \sqsubseteq t'}{\texttt{Set}(t) \sqsubseteq \texttt{Set}(t')}
$$

### B.3.5 *Programs, statements, and expressions.*

(Program, Init, and Def)

A program $P$ type checks in the context of an initial type environment $\Gamma$ (which we defined earlier) if we can derive $\Gamma \vdash P$ using the rule below. Notice that hypothesis of the rule type checks the top-level print statement using rules for statements that we will give below. In the hypothesis of the rule for programs, the occurrence of `Dyn` is a dummy that has no role to play here.

An init function *Init* type checks in the context of a type environment $\Gamma$ and a list of class definitions $\Psi$ if we can derive $\Gamma, \Psi \vdash \mathit{Init}$ using the rule below. In the hypothesis of the rule, the occurrence of `Dyn` is a dummy that has no role to play here. The rule for methods is similar, except here we see the type $t$, which is the return type of the method.

$$\frac{\Gamma, \mathit{abstract}(\overline{CD}) \vdash \overline{CD} \qquad \Gamma, \mathit{abstract}(\overline{CD}), \texttt{Dyn} \vdash \texttt{print}(C().m(\overline{e}))}{\Gamma \vdash \overline{I}\ \overline{CD}\ \texttt{print}(C().m(\overline{e}))}$$

$$\frac{\Gamma[\overline{f : t}], \Psi \vdash \mathit{Init} \qquad \Gamma[\overline{f : t}], \Psi \vdash \overline{D}}{\Gamma, \Psi \vdash \texttt{@fields}\ (\{\overline{f : t}\})\ \texttt{class}\ C(\texttt{object}) : \mathit{Init}\ \overline{D}}$$

$$\frac{\Gamma[\overline{x : t}][\overline{y : u}], \Psi, \texttt{Dyn} \vdash \overline{S} \qquad \mathit{locals}(\overline{S}) = \overline{y}}{\Gamma, \Psi \vdash \texttt{def}\ \texttt{\_\_init\_\_}(\overline{x : t}) : \overline{S}}$$

$$\frac{\Gamma[\overline{x : t}][\overline{y : u}], \Psi, t \vdash \overline{S} \qquad \mathit{locals}(\overline{S}) = \overline{y}}{\Gamma, \Psi \vdash \texttt{def}\ m(\overline{x : t})\ \texttt{->}\ \ t : \overline{S}}$$

(Statement)

A statement $S$ type checks in the context $\Gamma, \Psi, t_{ret}$ if we can derive $\Gamma, \Psi, t_{ret} \vdash S$ using the rules below. Here $\Gamma$ is a type environment, $\Psi$ is a list of class definitions, and $t_{ret}$ is the return type of the method in which $S$ occurs. In more detail, the rule above for $\texttt{def}\ m(\overline{x : t})\ \texttt{->}\ t$ picks up the type $t$, which in $\Gamma, \Psi, t_{ret} \vdash S$ is called $t_{ret}$.

$$\frac{}{\Gamma, \Psi, t_{ret} \vdash \texttt{pass}}$$

$$\frac{\Gamma(x) = t \qquad \Gamma, \Psi \vdash e : u \qquad t \sim u}{\Gamma, \Psi, t_{ret} \vdash x = e}$$

$$\frac{\Gamma(x) \rhd_f^{\Psi} t \qquad \Gamma, \Psi \vdash e : u \qquad t \sim u}{\Gamma, \Psi, t_{ret} \vdash x.f = e}$$

$$\frac{\Gamma(x) \rhd_{List} \texttt{List}(t) \qquad \Gamma, \Psi \vdash e_2 : u_2 \qquad \Gamma, \Psi \vdash e_3 : u_3 \qquad u_2 \sim \texttt{int} \qquad t \sim u_3}{\Gamma, \Psi, t_{ret} \vdash x[e_2] = e_3}$$

$$\frac{\Gamma, \Psi \vdash e : u \qquad \Gamma, \Psi, t_{ret} \vdash S_i \qquad \Gamma, \Psi, t_{ret} \vdash S \qquad u \sim \texttt{bool}}{\Gamma, \Psi, t_{ret} \vdash \texttt{if}\ e\texttt{:}\ \overline{S}\ \texttt{else:}\ S}$$

$$\frac{\Gamma, \Psi \vdash e : u \qquad \Gamma, \Psi, t_{ret} \vdash S_i \qquad \Gamma, \Psi, t_{ret} \vdash S \qquad u \sim \texttt{bool}}{\Gamma, \Psi, t_{ret} \vdash \texttt{while}\ e\texttt{:}\ \overline{S}\ \texttt{else:}\ S}$$

$$\frac{\Gamma, \Psi \vdash e : u \qquad \Gamma[x : t], \Psi, t_{ret} \vdash S_i \qquad \Gamma[x : t], \Psi, t_{ret} \vdash S \qquad u \sim \texttt{List}(t)}{\Gamma, \Psi, t_{ret} \vdash \texttt{for}\ x\ \texttt{in}\ e\texttt{:}\ \overline{S}\ \texttt{else:}\ S}$$

$$\frac{}{\Gamma, \Psi, t_{ret} \vdash \texttt{raise}\ C()}$$

$$\frac{}{\Gamma, \Psi, t_{ret} \vdash \texttt{break}}$$

$$\frac{\Gamma, \Psi \vdash e : u \qquad u \sim \texttt{int}}{\Gamma, \Psi, t_{ret} \vdash \texttt{print}(e)}$$

$$\frac{\Gamma, \Psi \vdash e : u \qquad u \sim t_{ret}}{\Gamma, \Psi, t_{ret} \vdash \texttt{return}\ e}$$

$$\frac{\Gamma, \Psi \vdash e_1 : u_1 \qquad u_1 \rhd_{List} \texttt{List}(t) \qquad \Gamma, \Psi \vdash e_2 : u_2 \qquad t \sim u_2}{\Gamma, \Psi \vdash e_1.\texttt{append}(e_2)}$$

$$\frac{\Gamma, \Psi \vdash e : u \qquad u \rhd_{List} \texttt{List}(t)}{\Gamma, \Psi \vdash e.\texttt{pop}()}$$

$$\frac{\Gamma, \Psi \vdash e_1 : u_1 \qquad u_1 \rhd_{Set} \texttt{Set}(t) \qquad \Gamma, \Psi \vdash e_2 : u_2 \qquad t \sim u_2}{\Gamma, \Psi \vdash e_1.\texttt{add}(e_2)}$$

$$\frac{\Gamma, \Psi \vdash e : u \qquad u \rhd_{Set} \texttt{Set}(t)}{\Gamma, \Psi \vdash e.\texttt{clear}()}$$

(Expression)

An expression $e$ has type $t$ in the context $\Gamma$, $\Psi$ if we can derive $\Gamma, \Psi \vdash e : t$ using the rules below. Notice that we use $\Gamma$ in both of the first two rules. In the first rule we use $\Gamma$ to get the type of a predefined or imported *FunctionName*. In the second rule we use $\Gamma$ to get the type of a *ParameterName* or a *LocalName*.

$$\frac{\Gamma(F) : (\overline{t}) \to t \qquad \Gamma, \Psi \vdash e_i : u_i \qquad t_i \sim u_i}{\Gamma, \Psi \vdash F(\overline{e}) : t}$$

$$\frac{(x : t) \in \Gamma}{\Gamma, \Psi \vdash x : t}$$

$$\frac{}{\Gamma, \Psi \vdash \texttt{list}(\,) : \texttt{List}(t)}$$

$$\frac{}{\Gamma, \Psi \vdash \texttt{set}(\,) : \texttt{Set}(t)}$$

$$\frac{C \rhd^{\Psi}_{init} (\overline{t}) \qquad \Gamma, \Psi \vdash e_i : u_i \qquad t_i \sim u_i}{\Gamma, \Psi \vdash C(\overline{e}) : C}$$

$$\frac{\Gamma, \Psi \vdash e : u \qquad u \rhd^{\Psi}_{f} t}{\Gamma, \Psi \vdash e.f : t}$$

$$\frac{\Gamma, \Psi \vdash e : u \qquad u \rhd^{\Psi}_{m} ((\overline{t}) \to t) \qquad \Gamma, \Psi \vdash e_i : u_i \qquad t_1 = u \qquad t_{i+1} \sim u_i}{\Gamma, \Psi \vdash e.m(\overline{e}) : t}$$

$$\frac{\Gamma, \Psi \vdash e : t}{\Gamma, \Psi \vdash \texttt{isinstance}(e, C) : \texttt{bool}}$$

$$\frac{\Gamma, \Psi \vdash e_i : t}{\Gamma, \Psi \vdash [\overline{e}] : \texttt{List}(t)}$$

$$\frac{\Gamma, \Psi \vdash e_1 : u_1 \qquad u_1 \rhd_{List} \texttt{List}(t) \qquad \Gamma, \Psi \vdash e_2 : u_2 \qquad u_2 \sim \texttt{int}}{\Gamma, \Psi \vdash e_1[e_2] : t}$$

$$\frac{}{\Gamma, \Psi \vdash \mathit{IntConst} : \texttt{int}}$$

$$\frac{}{\Gamma, \Psi \vdash \mathit{HexConst} : \texttt{int}}$$

$$\frac{}{\Gamma, \Psi \vdash \mathit{FloatConst} : \texttt{float}}$$

$$\frac{}{\Gamma, \Psi \vdash \texttt{None} : \texttt{Dyn}}$$

$$\frac{}{\Gamma, \Psi \vdash \texttt{True} : \texttt{bool}}$$

$$\frac{}{\Gamma, \Psi \vdash \texttt{False} : \texttt{bool}}$$

### B.4 Constraints

A constraint system consists of a finite set of variables $V$ and a finite set of constraints. We use $v, w$ to range over $V$, and we use $t$ to range over types. Constraints are of the forms shown in the first column of the table below.

We use $\varphi$ to range over maps from $V$ to types, and we use $\Psi$ to range over maps with the functionality described earlier. A pair $(\varphi, \Psi)$ satisfies a set of constraints if $(\varphi, \Psi)$ satisfies each constraint according to the definition in the second column of the table below.

| Constraints | | A solution $(\varphi, \Psi)$ |
|---|---|---|
| $v = v'$ | : | $\varphi(v) = \varphi(v')$ |
| $v = t$ | : | $\varphi(v) = t$ |
| $v = \texttt{List}(v')$ | : | $\varphi(v) = \texttt{List}(\varphi(v'))$ |
| $v = \texttt{Set}(v')$ | : | $\varphi(v) = \texttt{Set}(\varphi(v'))$ |
| $t \sqsubseteq v$ | : | $t \sqsubseteq \varphi(v)$ |
| $v \sim v'$ | : | $\varphi(v) \sim \varphi(v')$ |
| $v \sim t$ | : | $\varphi(v) \sim t$ |
| $v \sim \texttt{List}(v')$ | : | $\varphi(v) \sim \texttt{List}(\varphi(v'))$ |
| $v \sim \texttt{Set}(v')$ | : | $\varphi(v) \sim \texttt{Set}(\varphi(v'))$ |
| $C \rhd_{init} \overline{v}$ | : | $C \rhd^{\Psi}_{init} \overline{\varphi(v)}$ |
| $v \rhd_{List} \texttt{List}(v')$ | : | $\varphi(v) \rhd_{List} \texttt{List}(\varphi(v'))$ |
| $v \rhd_{Set} \texttt{Set}(v')$ | : | $\varphi(v) \rhd_{Set} \texttt{Set}(\varphi(v'))$ |
| $C \rhd_f w$ | : | $C \rhd^{\Psi}_f \varphi(w)$ |
| $v \rhd_f w \;;;\; v \in [C_1, \ldots, C_n]$ | : | $\varphi(v) \rhd^{\Psi}_f \varphi(w)$ and $\varphi(v) \in \{\texttt{Dyn}\} \cup \{C_1, \ldots, C_n\}$ |
| $C \rhd_m (\overline{v}) \to w$ | : | $C \rhd^{\Psi}_m (\overline{\varphi(v)}) \to \varphi(v')$ |
| $v \rhd_m (\overline{v}) \to w \;;;\; v \in [C_1, \ldots, C_n]$ | : | $\varphi(v) \rhd^{\Psi}_m (\overline{\varphi(v)}) \to \varphi(v')$ and $\varphi(v) \in \{\texttt{Dyn}\} \cup \{C_1, \ldots, C_n\}$ |
| $v \in \{C_1, \ldots, C_n\}$ | : | $\varphi(v) \in \{C_1, \ldots, C_n\}$ |

### B.5 Simplifying constraints

A simple constraint system is a constraint system that satisfies the following three conditions.

(1) No constraint is of the forms $v = \texttt{List}(t)$ and $v = \texttt{Set}(t)$ and $v \sim \texttt{List}(t)$ and $v \sim \texttt{Set}(t)$ and $t \sqsubseteq v$.
(2) For every constraint of the form $v \triangleright_f w \;;;\; v \in [C_1, \ldots, C_n]$, we also have a single constraint of the form $C_i \triangleright_f w$.
(3) For every constraint of the form $v \triangleright_m (\overline{v}) \to w$, we also have a single constraint of the form $C_i \triangleright_m (\overline{v}) \to w$.

We can transform a constraint system into a constraint system that satisfies the first condition by repeating the following transformation until it no longer has an effect.

| From | | To: |
|---|---|---|
| $v = \texttt{List}(t)$ | : | $v = \texttt{List}(v') \;\wedge\; v' = t$ where $v'$ is a fresh type variable |
| $v = \texttt{Set}(t)$ | : | $v = \texttt{Set}(v') \;\wedge\; v' = t$ where $v'$ is a fresh type variable |
| | | |
| $v \sim \texttt{Dyn}$ | : | no constraint |
| $v \sim \texttt{List}(t)$ | : | $v \sim \texttt{List}(v') \;\wedge\; v' \sim t$ where $v'$ is a fresh type variable |
| $v \sim \texttt{Set}(t)$ | : | $v \sim \texttt{Set}(v') \;\wedge\; v' \sim t$ where $v'$ is a fresh type variable |
| | | |
| $\texttt{Dyn} \sqsubseteq v$ | : | no constraint |
| $\texttt{bool} \sqsubseteq v$ | : | $v = \texttt{bool}$ |
| $\texttt{int} \sqsubseteq v$ | : | $v = \texttt{int}$ |
| $\texttt{float} \sqsubseteq v$ | : | $v = \texttt{float}$ |
| $\texttt{List}(t) \sqsubseteq v$ | : | $v = \texttt{List}(v') \;\wedge\; t \sqsubseteq v'$ where $v'$ is a fresh type variable |
| $\texttt{Set}(t) \sqsubseteq v$ | : | $v = \texttt{Set}(v') \;\wedge\; t \sqsubseteq v'$ where $v'$ is a fresh type variable |
| $C \sqsubseteq v$ | : | $v = C$ |

In Section 5, we will present a mapping from programs to constraints that produces constraints that satisfy the second and third condition above. Thus, we can map such a set of constraints to a set of simple constraints by applying the transformation above.

### B.6 Solving constraints

A set of simple constraints is *closed* if it satisfies the following rules. Each rule says that if zero, one, or two constraints are in the set, then some other constraints are also in the set. We use $s$ to range over $\{\texttt{bool}, \texttt{int}, \texttt{float}, \texttt{List}(v''), \texttt{Set}(v''), C\}$.

1st rule: the equality relation = is reflexive, symmetric, and transitive.

2rd rule: if $\texttt{List}(v_1) = \texttt{List}(v_2)$, then $v_1 = v_2$.

3rd rule: if $\texttt{Set}(v_1) = \texttt{Set}(v_2)$, then $v_1 = v_2$.

4th rule: if $v \rhd_{List} \texttt{List}(v')$ and $v = \texttt{Dyn}$, then $v' = \texttt{Dyn}$.

5th rule: if $v \rhd_{Set} \texttt{Set}(v')$ and $v = \texttt{Dyn}$, then $v' = \texttt{Dyn}$.

6th rule: if $v \rhd_f w \;\;;\;\; v \in [C_1, \ldots, C_n]$ and $v = \texttt{Dyn}$, then $w = \texttt{Dyn}$.

7th rule: if $v \rhd_m (\overline{v}) \to w \;\;;\;\; v \in [C_1, \ldots, C_n]$ and $v = \texttt{Dyn}$, then $v_i = \texttt{Dyn}$ and $w = \texttt{Dyn}$.

8th rule: if $v \rhd_{List} \texttt{List}(v')$ and $(v = s$ or $v' = s)$, then $v = \texttt{List}(v')$.

9th rule: if $v \rhd_{Set} \texttt{Set}(v')$ and $(v = s$ or $v' = s)$, then $v = \texttt{Set}(v')$.

10th rule: if $C \rhd_{init} (\overline{v})$ and $C \rhd_{init} (\overline{v'})$, then $v_i = v'_i$.

11th rule: if $v \rhd_f w \;\;;\;\; v \in [C_1, \ldots, C_n]$ and $v = C_i$ and $C_i \rhd_f w'$, then $w = w'$.

12th rule: if $v \rhd_m (\overline{v}) \to w \;\;;\;\; v \in [C_1, \ldots, C_n]$ and $v = C_i$ and $C_i \rhd_m (\overline{v'}) \to w'$, then $v_i = v'_i$ and $w = w'$.

13th rule: if $C \rhd_m (\overline{v}) \to w$ and $C \rhd_m (\overline{v'}) \to w'$, then $v_i = v'_i$ and $w = w'$.

14th rule: the relation $\sim$ is reflexive and symmetric.

15th rule: if $\texttt{List}(v_1) \sim \texttt{List}(v_2)$, then $v_1 \sim v_2$.

16th rule: if $\texttt{Set}(v_1) \sim \texttt{Set}(v_2)$, then $v_1 \sim v_2$.

17th rule: if $s = s'$ and $s' \sim s''$, then $s \sim s''$.

18th rule: if $v \rhd_f w \;\;;\;\; v \in [C_1, \ldots, C_n]$ and $(v = s$ or $w = s)$, then $v \in \{C_1, \ldots, C_n\}$.

19th rule: if $v \rhd_m (\overline{v}) \to w \;\;;\;\; v \in [C_1, \ldots, C_n]$ and $(v = s$ or $v_i = s$ or $w = s)$, then $v \in \{C_1, \ldots, C_n\}$.

The *closure* of a constraint $C$ is the smallest closed constraint that contains $C$ as a subset.

Notice that Rules 1–12 produce constraints of the form $s = s'$, Rules 13–16 produce constraints of the form $s \sim s'$, and Rules 17–18 produce constraints of the form $v \in \{C_1, \ldots, C_n\}$.

Notice that the 8th rule includes the case that if $\texttt{Set}(t) \rhd_{List} \texttt{List}(t')$, then $\texttt{Set}(t) = \texttt{List}(t')$. The point here is that $\texttt{Set}(t) \rhd_{List} \texttt{List}(t')$ is unsatisfiable and we want to make that explicit. In response, the idea of the 8th rule is to create an ill-formed constraint that the rules for well-formedness (see below) will catch.

Define *top* to be the function that maps an $s$ to a symbol as follows:

$$\begin{aligned} top(\texttt{Dyn}) &= \texttt{Dyn} \\ top(\texttt{bool}) &= \texttt{bool} \\ top(\texttt{int}) &= \texttt{int} \\ top(\texttt{float}) &= \texttt{float} \\ top(\texttt{List}(v)) &= \texttt{List} \\ top(\texttt{Set}(v)) &= \texttt{Set} \\ top(C) &= C \end{aligned}$$

A set of simple constraints is *well-formed* if none of the following three patterns are in the set, and the set satisfies the condition below.

$$\begin{aligned} s \sim s' &\quad \text{where } top(s) \neq top(s') \\ &\quad \text{and } s \neq \texttt{Dyn} \text{ and } s' \neq \texttt{Dyn} \\ v = s \text{ and } v \in \{C_1, \ldots, C_n\} &\quad \text{where } s \notin \{C_1, \ldots, C_n\} \\ v_1 = LS_1(v_2) \text{ and } v_2 = LS_2(v_3) \text{ and } \ldots v_n = LS_n(v_1) &\quad \text{where } LS_i \in \{\texttt{List}, \texttt{Set}\}. \end{aligned}$$

Notice that the third pattern is the classical *occurs check*.

Condition: if $v_1, \ldots, v_n$ is a maximal set of type variables such that

- for each $i, j \in 1..n$, the set of constraints contains $v_i = v_j$,
- the set of constraints doesn't contain any $v_i = s$, and
- the set of constraints contains at least one constraint of the form $v_i \in \{C_1, \ldots, C_n\}$ for some $i$,

then across all constraints of the form $v_i \in \{C_1, \ldots, C_n\}$, there exists $C$ such that $C$ is a member of all those sets.

*Definition B.1.* An algorithm for solving a set of simple constraints.
Input: a set $C$ of simple constraints.
Output: true, if $C$ is satisfiable, and false otherwise.
Method:

(1) For each constraint of the form $v \rhd_f w \;;;\; v \in [C_1, \ldots, C_n]$,
guess one of $v = \texttt{Dyn}$ or $v = C_i$, for some $i$.
For each constraint of the form $v \rhd_m (\overline{v}) \to w \;;;\; v \in [C_1, \ldots, C_n]$,
guess one of $v = \texttt{Dyn}$ or $v = C_i$, for some $i$.
Add all the guessed constraints to $C$ to produce $C'$.
(2) Close $C'$ to produce $C''$.
(3) Output whether $C''$ is well-formed.

## B.7 From programs to constraints

*B.7.1 Type variables.* For each expression $e$ (which includes method arguments and local variables), we have a type variable $[\![e]\!]$. For each class $C$ and for each field $f$ in $C$, we have a type variable $[\![C.f]\!]$. For every class $C$ and for each method $m$ in $C$, we have a type variable $[\![C.m.ret]\!]$.

In the rule for a program, $v$ is a fresh type variable. In each of the rules for the statements `append`, `pop`, `add`, and `clear`, $v$ is a fresh type variable. In each of the rules for the expressions `list( )`, `set( )`, and $[\overline{e}]$, $v$ is a fresh type variable. In each of the rules for the expressions $C(\overline{e})$ and $e.m(\overline{e})$, $\overline{v}$ are fresh type variables.

*B.7.2 Constraints.*

(Program) For a program $\overline{I}\ \overline{CD}$ `print`$(C().m(\overline{e}))$ we have the constraints

$$\begin{array}{rcl} C & \triangleright_{init} & (v_1) \\ v_1 & = & C \\ C & \triangleright_{m} & (\overline{v}) \to v \\ v_{i+1} & \sim & [\![e_i]\!] \\ v & \sim & \texttt{int} \end{array}$$

where $v$ is a fresh type variable.

(Class Def.) For a class definition `@fields` $(\{\overline{f : t}\})$ `class` $C$`(object)` : *Init* $\overline{D}$, we have the constraints

$$\begin{array}{rcl} t_i & \sqsubseteq & [\![C.f_i]\!] \\ C & \triangleright_{f_i} & [\![C.f_i]\!] \end{array}$$

(Init) For an initializer `def __init__`$(\overline{x : t}) : \overline{S}$ in a class $C$, we have the constraints

$$\begin{array}{rcl} t_i & \sqsubseteq & [\![x_i]\!] \\ C & \triangleright_{init} & \overline{[\![x_i]\!]} \end{array}$$

(Def) For a method definition in class $C$, `def` $m(\overline{x : t})$ `->` $t : \overline{S}$, we have the constraints

$$\begin{array}{rcl} t_i & \sqsubseteq & [\![x_i]\!] \\ t & \sqsubseteq & [\![C.m.ret]\!] \\ C & \triangleright_{m} & ((\overline{[\![x]\!]}) \to [\![C.m.ret]\!]) \end{array}$$

(Statement) For a type environment $\Gamma$ and for a statement in a class $C$ and in a method $m$, we generate constraints as follows.

$$
\begin{array}{rcl}
\texttt{pass} & : & \text{no constraints} \\
x = e & : & [\![x]\!] \sim [\![e]\!] \\
x.f = e & : & [\![x]\!] \rhd_f v \;\; ;; \;\; [\![x]\!] \in [C_1, \ldots, C_n] \;\; \wedge \;\; v \sim [\![e]\!] \\
 & & \text{where } C_1, \ldots, C_n \text{ are the classes that each has a field } f \\
x[e_2] = e_3 & : & [\![x]\!] \rhd_{List} \texttt{List}(v) \;\; \wedge \;\; [\![e_2]\!] \sim \texttt{int} \;\; \wedge \;\; v \sim [\![e_3]\!] \\
\texttt{if } e\texttt{: } \overline{S} \texttt{ else: } S & : & [\![e]\!] \sim \texttt{bool} \\
\texttt{while } e\texttt{: } \overline{S} \texttt{ else: } S & : & [\![e]\!] \sim \texttt{bool} \\
\texttt{for } x \texttt{ in } e\texttt{: } \overline{S} \texttt{ else: } S & : & [\![e]\!] \sim \texttt{List}([\![x]\!]) \\
\texttt{raise } C() & : & \text{no constraints} \\
\texttt{break} & : & \text{no constraints} \\
\texttt{print}(e) & : & [\![e]\!] \sim \texttt{int} \\
\texttt{return } e & : & [\![e]\!] \sim [\![C.m.ret]\!] \\
e_1\texttt{.append}(e_2) & : & [\![e_1]\!] \rhd_{List} \texttt{List}(v) \;\; \wedge \;\; v \sim [\![e_2]\!] \\
e\texttt{.pop}(\,) & : & [\![e]\!] \rhd_{List} \texttt{List}(v) \\
e_1\texttt{.add}(e_2) & : & [\![e_1]\!] \rhd_{Set} \texttt{Set}(v) \;\; \wedge \;\; v \sim [\![e_2]\!] \\
e\texttt{.clear}(\,) & : & [\![e]\!] \rhd_{Set} \texttt{Set}(v)
\end{array}
$$

(Expression) For a type environment $\Gamma$ and for an expression, we generate constraints as follows.

$$
\begin{array}{rcl}
F(\overline{e}) & : & [\![e_i]\!] \sim t_i \quad \wedge \quad [\![F(\overline{e})]\!] = t \\
 & & \text{where } \Gamma(F) = (\overline{(\bar{t})} \rightarrow t) \\
x & : & \text{no constraints} \\
\texttt{list(\,)} & : & [\![\texttt{list(\,)}]\!] = \texttt{List}(v) \\
\texttt{set(\,)} & : & [\![\texttt{set(\,)}]\!] = \texttt{Set}(v) \\
C(\overline{e}) & : & C \rhd_{init} \overline{v} \quad \wedge \quad v_1 = C \quad \wedge \quad v_{i+1} \sim [\![e_i]\!] \quad \wedge \quad [\![C(\overline{e})]\!] = C \\
e.f & : & [\![e]\!] \rhd_f [\![e.f]\!] \;\; ;; \;\; [\![e]\!] \in [C_1, \ldots, C_n] \\
 & & \text{where } C_1, \ldots, C_n \text{ are the classes that each has a field } f \\
e.m(\overline{e}) & : & [\![e]\!] \rhd_m ((\overline{v}) \rightarrow [\![e.m(\overline{e})]\!]) \;\; ;; \;\; [\![e]\!] \in [C_1, \ldots, C_n] \quad \wedge \\
 & & \qquad v_1 = [\![e]\!] \quad \wedge \quad v_{i+1} \sim [\![e_i]\!] \\
 & & \text{where } C_1, \ldots, C_n \text{ are the classes that each has a method } m \\
 & & \text{with the correct number of parameters} \\
\texttt{isinstance}(e, C) & : & [\![\texttt{isinstance}(e, C)]\!] = \texttt{bool} \\
[\overline{e}] & : & [\![e_i]\!] = v \quad \wedge \quad [\![[\overline{e}]]\!] = \texttt{List}(v) \\
e_1[e_2] & : & [\![e_1]\!] \rhd_{List} \texttt{List}([\![e_1[e_2]]\!]) \quad \wedge \quad [\![e_2]\!] \sim \texttt{int} \\
\mathit{IntConst} & : & [\![\mathit{IntConst}]\!] = \texttt{int} \\
\mathit{HexConst} & : & [\![\mathit{HexConst}]\!] = \texttt{int} \\
\mathit{FloatConst} & : & [\![\mathit{FloatConst}]\!] = \texttt{float} \\
\texttt{None} & : & [\![\texttt{None}]\!] = \texttt{Dyn} \\
\texttt{True} & : & [\![\texttt{True}]\!] = \texttt{bool} \\
\texttt{False} & : & [\![\texttt{False}]\!] = \texttt{bool}
\end{array}
$$