Type Inference with Run-time Logs (Work in Progress)

Ravi Chugh, Ranjit Jhala, and Sorin Lerner

University of California, San Diego

Abstract. Gradual type systems offer the possibility of migrating programs in dynamically-typed languages to more statically-typed ones. There is little evidence yet that large, real-world dynamically-typed programs can be migrated with a large degree of automation. Unfortunately, since these systems typically lack principal types, fully automatic type inference is beyond reach. To combat this challenge, we propose using logs from run-time executions to assist inference. As a first step, in this paper we study how to use run-time logs to improve the efficiency of a type inference algorithm for a small language with first-order functions, records, parametric polymorphism, subtyping, and bounded quantification. Handling more expressive features in order to scale up to gradual type systems for dynamic languages is left to future work.

1 Introduction

Dynamic languages have become increasingly popular in recent years, stimulating renewed interest in the long-studied question of how to mix the guarantees of static type systems with the flexibility of dynamic languages. The *gradual typing* approach [10, 11, 2] extends a static type system with a type `dynamic` that all values inhabit. Unlike values of a `Top` type, which cannot be used to do any computation, values of type `dynamic` can be implicitly downcast to any type. The benefit of this approach is that portions of a program can be annotated with non-trivial (non-`dynamic`) types that are statically checked in standard ways, while other portions annotated with `dynamic` fall back on run-time checks. The continuous spectrum offered by this approach is appealing for migrating existing programs written in dynamic languages to more statically-typed languages.

There is little evidence yet that such programs can easily be migrated, however. The first barrier is defining a type system (even with the `dynamic` type) that can assign types to non-trivial programs in practice. Many attempts [13, 14, 6, 5] are unable to type all of the features in a full dynamic language and so fall back on manual annotation or refactoring. Even if this challenge is overcome, the problem of how to add type annotations to existing unannotated programs remains. Requiring programmers to provide type annotations can hinder adoption, so a successful approach will likely require a large degree of automation.

Unfortunately, there are barriers to static type reconstruction for statically-typed object systems, let alone gradual ones. Both partial and full type inference

for System F are undecidable [8, 15], and many object systems [3] are based on yet more expressive type theories like $F_{\leq}$, which extends F with subtyping and bounded quantification [4, 9]. The lack of principal types in these systems stand in the way of effective type inference. Constraint-based systems have been used to recover principal types despite the presence of subtyping [12], but types in these systems can become more complex than those in syntactic systems like $F_{\leq}$. Since our ideal gradual type system should types that are easy to understand, in this work we focus on systems with syntactically-limited forms of subtyping constraints. When extending these systems with `dynamic`, to create gradual object systems, the problem becomes harder still because `dynamic` annotations can be assigned in multiple incomparable ways. One approach is to insert the minimal number of casts required [7]. Another approach, however, might be to insert casts to minimize how frequently they are executed during typical executions.

To combat these challenges, we propose an approach that uses traces from run-time executions to assist type inference for languages without principal types. The idea is that although a program may be well-typed in many ways, a particular set of executions may eliminate some of the incomparable possible types. The test suites that often accompany dynamically-typed programs could provide our approach with the run-time observations it needs. As a first step towards this goal, in this paper we study a language with first-order functions and records, defined in Section 2. We present several type systems for this language, fully-static type inference algorithms, and improvements that can be made with the help of run-time information. In Sections 3 and 4, we start with a system that has parametric polymorphism and subtyping; we call it System E due to its similarity to System F besides the lack of higher-order functions. In Section 5, we extend the system with a simple form of bounded quantification. Although we consider the type inference algorithms in a fair amount of detail, we will not prove desired formal properties in this work. In Section 6, we outline the future challenges for our approach, including adding support for recursion, higher-order functions, and `dynamic`.

Recent work on type inference for Ruby [1] leverages run-time executions to assign types to a program. They wrap run-time values with type variables and then generate subtype constraints when wrapped values are used at primitive operations, field operations, and method invocations. Constraints are then solved after execution to assign static types. Our approach, which instead starts with completely static type inference and then uses run-time information to improve its efficiency, deviates from their fully-dynamic approach for two reasons. Since dynamic code generation, a major obstacle to static type inference in Ruby, is not present in our language, we would like to do as much static inference as possible. Furthermore, even though not all expressions in our language have principal types, some do. We would like to statically infer types in these situations and only rely on run-time information when necessary. If the challenges for our approach are overcome, then the combination of these two approaches might be fruitful, because most dynamic languages have both dynamic code generation and features that prevent the existence of principal types.

2 Programs

Each type system in this paper classifies programs from the following grammar. A program is a sequence of function declarations.

$$
\begin{array}{rcl}
d & ::= & \texttt{def}\ z\texttt{(}x\texttt{)\{}e\texttt{\}} \mid \texttt{def}\ z\texttt{[}\overline{X}\texttt{](}x\texttt{:}\pi\texttt{)\{}e\texttt{\}} \\
e & ::= & x \mid c \mid e_1\ \texttt{op}\ e_2 \mid \texttt{\{}\overline{f}\texttt{=}\overline{e}\texttt{\}} \mid e\texttt{.}f \mid z\texttt{(}e\texttt{)} \mid z\texttt{[}\overline{\tau}\texttt{](}e\texttt{)} \\
 & \mid & \texttt{if}_k\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2
\end{array}
$$

We use the metavariables d for function declarations, e for expressions, x and y for arguments and let-bound variables, z for function identifiers, c for base value constants, f and g for field names, B for base types, X for type variables, and τ, π, and σ for arbitrary types. Expressions are variables, base values (naturals, booleans, etc.), primitive operations, record literals, field projections, function calls, if-expressions labeled with unique integer identifiers k, and let-bindings. We use vector notation for sequences and use integer subscripts to index them. Note that function definitions and calls both have untyped and typed versions. Strictly speaking, terms of the external language contain no type annotations and leave the burden to the type inference algorithm, and terms of the internal language always provide type annotations to be checked by the type checker. To keep the presentation simple, we informally represent both forms with the same grammar, and we rely on context to disambiguate between typed and untyped programs.

3 Type Inference for System E$^-$

We start with E$^-$ ("E-Minus"), a system without a typing rule for if-expressions. The type language, subtyping relation, two expression typing rules, and a top-level function typing rule for E$^-$ are shown below.

$$\tau \quad ::= \quad B \mid X_{x.l} \mid \texttt{\{}\overline{f}\texttt{:}\overline{\tau}\texttt{\}}_\bullet \mid \texttt{\{}\overline{f}\texttt{:}\overline{\tau}\texttt{\}}_{x.l}$$

$$\frac{}{\tau \le \tau}\ \textsc{S-Refl} \qquad \frac{\forall i.\ \exists ji.\ f_i = g_{ji} \wedge \pi_{ji} \le \tau_i}{\texttt{\{}\overline{g}\texttt{:}\overline{\pi}\texttt{\}} \le \texttt{\{}\overline{f}\texttt{:}\overline{\tau}\texttt{\}}}\ \textsc{S-Rcd}$$

$$\frac{S; \Gamma \vdash e : \pi \qquad \pi \le \texttt{\{}f\texttt{:}\tau\texttt{\}}}{S; \Gamma \vdash e\texttt{.}f : \tau}\ \textsc{T-Proj} \qquad \frac{S(z) = \forall \overline{X}.\ \pi \to \pi' \qquad S; \Gamma \vdash e : \tau \qquad |\overline{X}| = |\overline{\sigma}| \qquad \tau \le \pi[\overline{\sigma}/\overline{X}]}{S; \Gamma \vdash z\texttt{[}\overline{\sigma}\texttt{](}e\texttt{)} : \pi'[\overline{\sigma}/\overline{X}]}\ \textsc{T-App}$$

$$\frac{S; x\texttt{:}\pi \vdash e : \pi'}{S \vdash \texttt{def}\ z\texttt{[}\overline{X}\texttt{](}x\texttt{:}\pi\texttt{)\{}e\texttt{\}} : \pi'}\ \textsc{T-Fun}$$

A signature S maps functions z to function types $\forall \overline{X}.\ \pi \to \pi'$ and a context Γ maps variables x to expression types τ. We write subscripts on type variables, using $X_{x.l}$ for the type of formal parameter x projected on the sequence of fields l. We also write subscripts on record types, using $\texttt{\{}\overline{f}\texttt{:}\overline{\tau}\texttt{\}}_\bullet$ to denote that the record

type corresponds to a record literal, and $\{\overline{f}:\overline{\tau}\}_{x.l}$ to denote that it corresponds to the parameter x projected on fields l. We omit subscripts on record types when they are irrelevant. Subscripts on type variables and record types are used only during inference and are ignored by the type checking rules. Notice that in the T-FUN rule, we check that the number of type actuals supplied i equals the number of type parameters required j. We expect E^- to have a standard soundness property but have not checked it.

Consider the following function and three of its infinitely many valid types.[1]

```
def wrap(x){ {orig=x;asucc=succ x.a} }
```

$$\begin{array}{lll}
\sigma_1 & & \texttt{\{a:Nat\}}_{\texttt{x}} \to \texttt{\{orig:\{a:Nat\}}_{\texttt{x}}\texttt{;asucc:Nat\}}_{\bullet} \\
\sigma_2 & \forall X_{\texttt{x.b}}. & \texttt{\{a:Nat;b:}X_{\texttt{x.b}}\texttt{\}}_{\texttt{x}} \to \texttt{\{orig:\{a:Nat;b:}X_{\texttt{x.b}}\texttt{\}}_{\texttt{x}}\texttt{;asucc:Nat\}}_{\bullet} \\
\sigma_3 & & \texttt{\{a:Nat;b:Bool\}}_{\texttt{x}} \to \texttt{\{orig:\{a:Nat;b:Bool\}}_{\texttt{x}}\texttt{;asucc:Nat\}}_{\bullet}
\end{array}$$

There is no best type among these; σ_2 is better than σ_3, but σ_1 and σ_2 are incomparable since σ_1 would allow `wrap` to be called in more contexts but σ_2 would allow its return value to used in more contexts. This demonstrates that E^- does not have the principal type property, so our type inference algorithm considers the calling contexts of `wrap` to determine which type to assign.

3.1 Iterative Static Inference

In this section, we outline an iterative, fully-static type inference algorithm for E^-. We develop the intuition for the algorithm by considering how to type `wrap` and the following calling context.

```
def main(){ let o = wrap({a=1;b=true}).orig in o.b }
```

The first time we process `wrap`, we gather constraints on its argument based on its uses only within the function body. Since the only requirement is that `x` has an `a` field, we assign σ_1 before moving on to `main`. Because of the function's return type and the projection on field `orig`, the variable `o` gets type $\texttt{\{a:Nat\}}_{\texttt{x}}$, so the expression `o.b` is not well-typed. The subscript `x` indicates that the record type originated from the parameter of `wrap`, so if we could require `x` to have a `b` field, then `o.b` would be well-typed. At this point, we record the *caller-induced* field constraint (as opposed to *callee-induced* constraints generated from the body of a function) that `x` has `b` and restart inference from the beginning of the program.[2] In light of this constraint, when we process `wrap` for the second time, we assign σ_2. Notice that the type of the `b` field is left unconstrained. When we process `main` for the second time, the type of `o` is $\texttt{\{a:Nat;b:Bool\}}_{\texttt{x}}$, so `o.b` is well-typed. In general, when a function type is changed from one iteration to the next, calls that were well-typed with the old type may not be with the new type. Multiple valid function types may be incomparable, so there might not be one that satisfies all of its callers. Similarly, although one type may satisfy all

[1] This example is used to motivate bounded quantification in TAPL [9].

[2] An optimized version would process only `wrap` and its (transitive) callers again.

calling contexts in a given program, it may not satisfy other well-behaved calls added to the program in the future.

Our inference algorithm for E^- generates constraint sets C with two kinds of constraints. A type variable can either be equated to a base type when used at a primitive operation or required to have a field because of a projection. Notice that we do not generate arbitrary subtyping constraints between types, only subtyping constraints to denote that a record must have a particular field. This limited form of subtyping constraints enables a simple solving algorithm (Appendix A). Caller-induced constraints will always be of the second kind.

$$C \quad ::= \quad C \cup \{X_{x.l} = B\} \;\mid\; C \cup \{X_{x.l} \leq \{\, f : X_{x.l.f} \,\}\} \;\mid\; \emptyset$$

Constraint Typing for Expressions. In this section, we define a constraint typing relation that derives a type for an expression if constraints induced by the expression are satisfiable. A term e is rewritten to e', because type parameters for function calls must be filled in. For all other kinds of expressions, the original and rewritten terms are equal. We intend the following soundness proposition, as well as a similar completeness one, to hold, though we do not prove it.

$$\text{If } S; \Gamma \vdash e \Rightarrow e' : \tau \mid C \text{ and } \theta = \mathit{Solve}(C), \text{Then } S; \theta\Gamma \vdash \theta e' \,:\, \theta\tau$$

The interesting cases are for field projection and function application. In the following, we assume that $\mathtt{x}$ is the formal parameter for the function currently being processed and y ranges over the parameters of previously processed functions. We first consider two projection rules.

$$\frac{S; \Gamma \vdash e \Rightarrow e' : \{\, \overline{g} : \overline{\tau} \,\} \mid C \qquad \exists j.\ f = g_j}{S; \Gamma \vdash e.f \Rightarrow e'.f : \tau_j \mid C}\ \text{CT-Proj1} \qquad \frac{S; \Gamma \vdash e \Rightarrow e' : X_{\mathtt{x}.l} \mid C \qquad C' = C \cup \{X_{\mathtt{x}.l} \leq \{\, f : X_{\mathtt{x}.l.f} \,\}\}}{S; \Gamma \vdash e.f \Rightarrow e'.f : X_{\mathtt{x}.l.f} \mid C'}\ \text{CT-Proj2}$$

If e is a record type with the desired field f, CT-Proj1 concludes that $e.f$ has the type of the field. If the type of e is a variable $X_{\mathtt{x}.l}$ (one of the arguments for the function that we are currently analyzing), CT-Proj2 imposes the appropriate subtype constraint on $X_{\mathtt{x}.l}$ and $X_{\mathtt{x}.l.f}$ and concludes that $e.f$ has type $X_{\mathtt{x}.l.f}$.

To support backtracking, we define a second constraint expression typing relation that indicates failure with a caller-induced constraint. CT-Proj3 triggers backtracking by producing a caller-induced constraint on the parameter y of a previous function, which immediately propagates through expressions with rules like CT-PropSucc and CT-PropApp1.

$$\frac{S; \Gamma \vdash e \Rightarrow e' : \{\, \overline{g} : \overline{\tau} \,\}_{y.l} \mid C \qquad \nexists j.\ f = g_j}{S; \Gamma \vdash e.f \nwarrow \{X_{y.l} \leq \{\, f : X_{y.l.f} \,\}\}}\ \text{CT-Proj3}$$

$$\frac{S; \Gamma \vdash e \nwarrow C}{S; \Gamma \vdash \mathtt{succ}\ e \nwarrow C}\ \text{CT-PropSucc} \qquad \frac{S; \Gamma \vdash e \nwarrow C}{S; \Gamma \vdash z(e) \nwarrow C}\ \text{CT-PropApp1}$$

The last case that we discuss here is for function calls. The *Call* judgment (defined in Appendix A) takes a formal type π, an actual type τ, and returns a substitution θ that instantiates the free variables of π together with additional

constraints C' required for τ to be a subtype of π (after instantiation). The additional constraints may fail because of missing fields, so there is also a failure version of $Call$, which is propagated by CT-PropApp2.

$$\frac{\begin{array}{c} S(z) = \forall \overline{X}.\ \pi \to \pi' \\ S; \Gamma \vdash e \Rightarrow e' : \tau \mid C \\ Call(\pi, \tau) \vdash (C', \theta) \qquad \overline{\sigma} = \theta \overline{X} \end{array}}{S; \Gamma \vdash z(e) \Rightarrow z[\overline{\sigma}](e') : \theta\pi' \mid C \cup C'}\ \text{CT-App} \qquad \frac{\begin{array}{c} S(z) = \forall \overline{X}.\ \pi \to \pi' \\ S; \Gamma \vdash e \Rightarrow e' : \tau \mid C \\ Call(\pi, \tau) \nwarrow C' \end{array}}{S; \Gamma \vdash z(e) \nwarrow C'}\ \text{CT-PropApp2}$$

Constraint Typing for Functions. We define a constraint typing relation for function definitions that processes one at a time by appealing to the constraint expression typing relation for its body. The judgment refers to a set of caller-induced constraints C_0 from previous iterations, as well as the list of previously processed untyped functions $\overline{d}$, which must be processed again if there are new caller-induced constraints.

$$\frac{\begin{array}{c} S; x : X_x \vdash e \Rightarrow e' : \tau' \mid C \qquad \theta = Solve(C_0 \cup C) \\ \pi = \theta X_x \qquad \pi' = \theta\tau' \qquad \overline{X} = FTV(\pi) \qquad e'' = \theta e' \end{array}}{C_0; \overline{d}; S \vdash \texttt{def}\ z(x)\{e\} \Rightarrow \texttt{def}\ z[\overline{X}](x:\pi)\{e''\} : \pi'}\ \text{CT-Fun}$$

$$\frac{\begin{array}{c} S; x : X_x \vdash e \nwarrow C \qquad C_1 = C_0 \cup C \\ C_1; -; - \vdash \overline{d} \Rightarrow \overline{d'} : S' \qquad C_1; \overline{d}; S' \vdash d_1 \Rightarrow d_2 : \pi' \end{array}}{C_0; \overline{d}; S' \vdash d_1 \Rightarrow d_2 : \pi'}\ \text{CT-Iter}$$

The CT-Fun rule applies when the body of function z has a successful constraint typing derivation. In addition to the constraints C that it produces, the caller-induced constraints C_0 must also be satisfied. The $Solve$ function (defined in Appendix A) attempts to solve C and C_0. If a solution exists, it is used to ascribe a function type to the declaration. The CT-Iter rule handles the case where constraint typing on the function body fails with new caller-induced constraints C. These are combined with the existing caller-induced constraints C_0, and all previous functions $\overline{d}$ are processed again (the third premise). We omit the definition of this relation. If all previous functions can be typed with the updated caller-induced constraints, then typing of the current function resumes (the fourth premise). Notice that the signature S' may differ from S, since the types of previously processed functions may have changed.

Like with constraint typing for expressions, we intend that the following soundness property holds, as well as a similar completeness one.

If $C_0; \overline{d}; S \vdash \texttt{def}\ z(x)\{e\} \Rightarrow \texttt{def}\ z[\overline{X}](x:\pi)\{e''\} : \pi'$,
Then $S \vdash \texttt{def}\ z[\overline{X}](x:\pi)\{e''\} : \pi'$.

We expect that this property will be considerably harder to prove than the one for expressions. One important lemma will be that if the third premise of CT-Iter is satisfied, then resumption of the current function (the fourth premise), will get strictly farther than did the previous attempt (the first premise). Because expressions are finite, this will enable proving termination.

3.2 Inference with Run-time Logs

Now that we have sketched out how static iterative type inference for E^- works, we turn to the question of how run-time logs might help. The need for iteration in our constraint typing rules comes from the fact that when processing a function, we do not know how its return value will be used in calling contexts. We will define an evaluation semantics for E^- to record precisely this information, so that we can define a modified inference algorithm that does not need to backtrack.[3]

In our evaluation semantics, we wrap run-time values with sets of type variables T. We use v to range over raw values, tv over tagged values, and L over run-time logs. Notice that every run-time log is a valid constraint set.

$$\begin{array}{rcl} v & ::= & c \mid \texttt{\{}\,\overline{f}\texttt{=}\,\overline{tv}\,\texttt{\}} \\ tv & ::= & (v, T) \\ L & ::= & L \cup \{X_{x.l} \leq \texttt{\{}\,f : X_{x.l.f}\,\texttt{\}}\} \mid \emptyset \end{array}$$

Our big-step evaluation relation evaluates an expression e in an environment E, which maps variables to tagged values, and produces a tagged value along with a log. The cases for application and projection are the interesting ones.

$$\frac{\begin{array}{c} S(z) = \lambda x.e' \\ (E, e) \Downarrow ((v, T), L) \\ tv_1 = (v, T \cup \{X_x\}) \\ (E[x \mapsto tv_1], e') \Downarrow (tv_2, L') \end{array}}{(E, z\texttt{(}e\texttt{)}) \Downarrow (tv_2, L \cup L')}\ \text{E-App} \qquad \frac{\begin{array}{c} (E, e) \Downarrow ((\texttt{\{}\,\overline{g}\texttt{=}\,\overline{tv}\,\texttt{\}}, T), L) \\ \exists j.\ f = g_j \quad (v_j, T_j) = tv_j \\ L' = \{\, X_{y.l} \leq \texttt{\{}\,f : X_{y.l.f}\,\texttt{\}} \mid X_{y.l} \in T \,\} \\ T' = \{\, X_{y.l.f} \mid X_{y.l} \in T \,\} \end{array}}{(E, e\,.\,f) \Downarrow ((v_j, T_j \cup T'), L \cup L')}\ \text{E-Proj}$$

When a tagged value is passed as an argument to a function with parameter x, E-App adds the variable X_x to its tag set before evaluating the function body. When a tagged value is projected on field f, E-Proj records constraints that each of its type variables have the f field. There are no rules that strip tags from values, so once E-App adds a tag to a value, it will carry around that tag for the rest of its execution. Consequently, if a function returns one of its arguments, the run-time log will capture all subsequent caller-induced constraints.

We can now improve our constraint typing for functions. In place of CT-Fun and CT-Iter, we define a new relation that refers to a run-time log and does not need to maintain previously processed functions. CTL-Fun refers to a log L instead of caller-induced constraints C_0 like CT-Fun does.

$$\frac{\begin{array}{c} S; x : X_x \vdash e \Rightarrow e' : \tau' \mid C \qquad \theta = Solve(L \cup C) \\ \pi = \theta X_x \qquad \pi' = \theta\tau' \qquad \overline{X} = FTV(\pi) \qquad e'' = \theta e' \end{array}}{L; S \vdash \texttt{def}\ \ z\texttt{(}x\texttt{)\{}\,e\,\texttt{\}} \Rightarrow \texttt{def}\ \ z\texttt{[}\overline{X}\texttt{](}x\texttt{:}\pi\texttt{)\{}\,e''\,\texttt{\}} : \pi'}\ \text{CTL-Fun}$$

If the execution that produced L exercised every expression in the program, then L will contain all the caller-induced constraints that the iterative static algorithm generates.[4] Thus, by using the run-time log, the modified inference algorithm

[3] We could instead use normal evaluation and instrument programs to emit logs.

[4] To check, we can add integer identifiers to expressions and log them during execution.

(which runs after execution) does not need to backtrack. We intend this modified algorithm to satisfy analagous soundness and completeness properties to the iterative version.

The fully-static and modified inference algorithms for E^-, which we considered fairly closely, form the basis of the algorithms for the remaining systems, which we will discuss in less detail.

4 Type Inference for System E

System E extends E^- with a typing rule for if-expressions. We extend the type language with three forms: a maximal type `Top`, a type variable that stands for the result of if-expression k and then projected on fields l, and a new kind of record type to indicate that its provenance is an if-expression.

$$\tau \quad ::= \quad \cdots \;\mid\; \texttt{Top} \;\mid\; X_{k.l} \;\mid\; \{\overline{f}:\overline{\tau}\}_{k.l}$$

The new subscripted type variables and record types are used only by the inference algorithm; they are "expanded away" during inference.

Consider the following function and several valid incomparable types. We use tuple notation freely since we can encode them as records with fields `1` and `2`. Each return type has a subscript to indicate its origin from the if-expression.

```
def choose(y,z){ if1 y.n > 0 then y else z }
```

$$\begin{array}{lll}
\sigma_4 & & (\{\texttt{n}:\texttt{Nat}\}_{\texttt{y}} * \{\}_{\texttt{z}}) \to \{\}_1 \\
\sigma_5 & & (\{\texttt{n}:\texttt{Nat}\}_{\texttt{y}} * \{\texttt{n}:\texttt{Top}\}_{\texttt{z}}) \to \{\texttt{n}:\texttt{Top}\}_1 \\
\sigma_6 & & (\{\texttt{n}:\texttt{Nat}\}_{\texttt{y}} * \{\texttt{n}:\texttt{Nat}\}_{\texttt{z}}) \to \{\texttt{n}:\texttt{Nat}\}_1 \\
\sigma_7 & & (\{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Bool}\}_{\texttt{y}} * \{\texttt{n}:\texttt{Nat}\}_{\texttt{z}}) \to \{\texttt{n}:\texttt{Nat}\}_1 \\
\sigma_8 & & (\{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Bool}\}_{\texttt{y}} * \{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Bool}\}_{\texttt{z}}) \to \{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Bool}\}_1 \\
\sigma_9 & & (\{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Top}\}_{\texttt{y}} * \{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Top}\}_{\texttt{z}}) \to \{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Top}\}_1 \\
\sigma_{10} & \forall X_{\texttt{y.b}}. & (\{\texttt{n}:\texttt{Nat};\texttt{b}:X_{\texttt{y.b}}\}_{\texttt{y}} * \{\texttt{n}:\texttt{Nat};\texttt{b}:X_{\texttt{y.b}}\}_{\texttt{z}}) \to \{\texttt{n}:\texttt{Nat};\texttt{b}:X_{\texttt{y.b}}\}_1
\end{array}$$

Although some of these types are better than others, there is no best one. For example, given the following calling contexts, σ_4 can type `main1`, σ_5 can type `main2`, and σ_6 can type `main3`, but these three types are incomparable.

```
def main1(){ let t1 = choose({n=1},{}) in t1 }
def main2(){ let t2 = choose({n=1},{n=true}) in t2.n }
def main3(){ let t3 = choose({n=1},{n=2}) in succ t3.n }
```

As before, our inference algorithm for System E iteratively adds constraints to arguments based on calling contexts. Consider how we arrive at σ_6 for `main3` in three iterations. In the first iteration, we assign σ_4 based on the constraints on `y` within the body of `choose`. When we get to the projection `t2.n`, we restart since `z` must have the `n` field for this projection to succeed. Thus, in the second iteration we assign σ_5. When we get to the `succ` operation, we require another restart since `y` and `z` must have the same type for the `n` field for this primitive operation to succeed. Thus, in the third iteration we assign σ_6.

Our constraint expression typing relation – in particular, the CT-If rule – now produces a "join tree" J that maps if-expression identifiers k to the types of their branches. We also generate two new kinds of constraints, both of which can be caller-induced.[5]

$$\frac{\begin{array}{c} S; \Gamma \vdash e_1 \Rightarrow e_1' \ : \texttt{Bool} \,;\, J_1 \mid C_1 \\ S; \Gamma \vdash e_2 \Rightarrow e_2' \ : \tau_2 \,;\, J_2 \mid C_2 \qquad S; \Gamma \vdash e_3 \Rightarrow e_3' \ : \tau_3 \,;\, J_3 \mid C_3 \\ J = J_1 \circ J_2 \circ J_3 \circ [k \mapsto (\tau_2, \tau_3)] \qquad C = C_1 \cup C_2 \cup C_3 \end{array}}{S; \Gamma \vdash \texttt{if}_k\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{if}_k\ e_1'\ \texttt{then}\ e_2'\ \texttt{else}\ e_3' \ : X_k \,;\, J \mid C}\ \text{CT-If}$$

$$C \quad ::= \quad \cdots \ \mid \ C \cup \{X_{k.l} \leq \texttt{\{}\, f : X_{k.l.f} \,\texttt{\}}\} \ \mid \ C \cup \{X_{k.l}\ \texttt{primop}\}$$

To solve a constraint of the form $X_{k.l} \leq \texttt{\{}\, f : X_{k.l.f} \,\texttt{\}}$, the inference algorithm uses the join tree to expand $X_{k.l}$ to its sources. For each source of the form $X_{y.l'}$, it adds the constraint $X_{y.l'.l} \leq \texttt{\{}\, f : X_{y.l'.l.f} \,\texttt{\}}$, which is a constraint on ordinary type variables. To solve a constraint of the form $X_{k.l}$ `primop`, the algorithm expands $X_{k.l}$ to its sources and equates them. The definition of *Expand* can be found in Appendix B.

Like we did for E^-, we can instrument evaluation so that the inference algorithm never needs to backtrack.

$$L \quad ::= \quad \cdots \ \mid \ L \cup \{X_{k.l} \leq \texttt{\{}\, f : X_{k.l.f} \,\texttt{\}}\} \ \mid \ L \cup \{X_{k.l}\ \texttt{primop}\}$$

$$\frac{(E, e_1) \Downarrow ((\texttt{true}, T_1), L_1) \qquad (E, e_2) \Downarrow ((v, T_2), L_2)}{(E, \texttt{if}_k\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \Downarrow ((v, T_2 \cup \{X_k\}), L_1 \cup L_2)}\ \text{E-IfTrue}$$

After E-IfTrue or E-IfFalse adds the X_k tag to the return value of the if-expression, the modified E-Proj rule (not shown), which treats $X_{k.l}$ variables like it does $X_{x.l}$ variables, gathers the caller-induced constraints that would require iteration in the fully-static algorithm.

5 Type Inference for System $\mathrm{E}_{\leq}$

We now add support for a simple form of bounded quantification. Each type variable in a function definition is declared with a single upper bound, which must be a base type, `Top`, or a record of bounds. In particular, a bound cannot be another type variable. This restriction allows constraint solving for $\mathrm{E}_{\leq}$ to remain simple.

$$\begin{array}{rcl} d & ::= & \cdots \ \mid \ \texttt{def}\ z\texttt{[}\overline{X} \leq \overline{\delta}\texttt{]}\texttt{(}x\texttt{:}\pi\texttt{)\{}\, e \,\texttt{\}} \\ \delta & ::= & B \ \mid \ \texttt{Top} \ \mid \ \texttt{\{}\, \overline{f}\texttt{:}\overline{\delta} \,\texttt{\}} \end{array}$$

[5] So that we can discuss the algorithm at a high-level, we omit a subtle detail from the discussion: the `Top` might also carry a subscript. Constraints of the form $X_{k.l}$ `primop` can only be directly generated within the function containing the if-expression, since type variables $X_{k.l}$ are eliminated when computing the function type. We use $\texttt{Top}_{k.l}$ to enable generating $X_{k.l}$ `primop` from a calling context.

We use U to range over lists of bounded type variables $\overline{X} \leq \overline{\delta}$. Function types become $\forall U.\ \pi \to \pi'$. The subtyping relation is defined to read subtyping assumptions from U, transitively if necessary. The typing judgment for expressions includes the bounds U of the particular function being typed. Before checking the type of the value argument, the function application rule must check that the type parameters supplied satisfy the bounds specified by the function's type.

$$\frac{X_{x.l} \leq \delta \in U}{U \vdash X_{x.l} \ \leq\ \delta}\ \text{S-BQ-Bound} \qquad \frac{U \vdash \delta_1 \ \leq\ \pi \quad U \vdash \pi \ \leq\ \delta_2}{U \vdash \delta_1 \ \leq\ \delta_2}\ \text{S-BQ-Trans}$$

$$\frac{\begin{array}{c} S(z) = \forall \overline{X} \leq \overline{\delta}.\ \pi \to \pi' \qquad n = |\overline{X}| = |\overline{\sigma}| \\ \forall i \in 1..n \qquad \theta_i = [X_1 \mapsto \sigma_1] \circ \cdots \circ [X_{i-1} \mapsto \sigma_{i-1}] \qquad U \vdash \sigma_i \ \leq\ \theta_i \delta_i \\ S; U; \Gamma \vdash e \ :\ \tau \qquad U \vdash \tau \ \leq\ \theta_n \pi \qquad \tau' = \theta_n \pi' \end{array}}{S; U; \Gamma \vdash z\,\texttt{[}\overline{\sigma}\texttt{]}\,\texttt{(}e\texttt{)} \ :\ \tau'}\ \text{T-BQ-App}$$

We now revisit the `wrap` and `choose` examples from previous sections. In $\mathrm{E}_{\leq}$, we can assign the following type to `wrap`.

$$\sigma_{11} \quad \forall X_{\mathtt{x}} \leq \texttt{\{a:Nat\}}.\ \ X_{\mathtt{x}} \to \texttt{\{orig:}X_{\mathtt{x}}\texttt{;asucc:Nat\}}_{\bullet}$$

This type is more general than σ_1, σ_2, and σ_3, the types that we saw in Section 3. Indeed, this is the most general type that can be assigned to `wrap`. In $\mathrm{E}_{\leq}$, we can assign the following type to `choose`.

$$\sigma_{12} \quad \forall X_{\mathtt{y}} \leq \texttt{\{n:Nat\}}.\ \ \texttt{(}X_{\mathtt{y}}\texttt{*}X_{\mathtt{y}}\texttt{)} \to X_{\mathtt{y}}$$

This type is more general than types σ_6 through σ_{10} that we saw in Section 4. However, σ_{12} is still incomparable with σ_4 and σ_5.

The previous example demonstrates that not every program has a principal type in System $\mathrm{E}_{\leq}$.[6] The source of ambiguity is whether or not to "share" a type variable for two parameters returned by an if-expression. To help demonstrate, consider the following versions of σ_4 and σ_5 equivalent to the original ones.

$$\begin{array}{ll} \sigma_4 & \forall X_{\mathtt{y}} \leq \texttt{\{n:Nat\}}.\ \forall X_{\mathtt{z}} \leq \texttt{\{\}}.\ \ \texttt{(}X_{\mathtt{y}}\texttt{*}X_{\mathtt{z}}\texttt{)} \to \texttt{\{\}}_1 \\ \sigma_5 & \forall X_{\mathtt{y}} \leq \texttt{\{n:Nat\}}.\ \forall X_{\mathtt{z.n}} \leq \texttt{Top}.\ \forall X_{\mathtt{z}} \leq \texttt{\{n:}X_{\mathtt{z.n}}\texttt{\}}.\ \ \texttt{(}X_{\mathtt{y}}\texttt{*}X_{\mathtt{z}}\texttt{)} \to \texttt{\{n:Top\}}_1 \end{array}$$

Notice that in σ_4 and σ_5 the constraints on `y` and `z` are kept separate with different type variables. In σ_{12}, on the other hand, the constraints are combined into a single type variable (arbitrarily named $X_{\mathtt{y}}$, though $X_{\mathtt{z}}$ would work just as well). The first factor to consider is whether the separate bounds for two type variables are compatible. Consider the following example.

```
def separateVars(v,w){ if2 iszero v.f && not w.f then v else w }
```

Since `v.f` and `w.f` have different base types, $X_{\mathtt{v}}$ and $X_{\mathtt{w}}$ must be kept separate; there is no single bound that can type both `v` and `w` correctly.

[6] In a system more powerful than $\mathrm{E}_{\leq}$, `choose` might have the principal type $\forall X_{\mathtt{y}}, X_{\mathtt{z}}.\ \{X_{\mathtt{y}} \leq \texttt{\{n:Nat\}}, X_{\mathtt{y}} \leq X_{\mathtt{z}}\} \Rightarrow \texttt{(}X_{\mathtt{y}}\texttt{*}X_{\mathtt{z}}\texttt{)} \to X_{\mathtt{z}}$.

If the constraints on two type variables for a branch are compatible, we start by using a shared variable. In general, sharing a type variable may impose more constraints on a parameter than if it was assigned its own type variable. Some call sites may not provide actuals that are well-typed with these additional requirements. For example, σ_{12} imposes more requirements on `z` than does σ_4. In the presence of a calling context like `choose({n=1},{})`, which is not well-typed when one type variable is shared for `y` and `z`, we trigger a restart and keep $X_{\mathtt{y}}$ and $X_{\mathtt{z}}$ separate in subsequent iterations. Whenever we are forced to keep two type variables separate – because their constraints are incompatible or because there is some incompatible call site – we must track fields for each type variable just we do for System E.

We would like to take advantage of run-time logs to eliminate backtracking. However, it is not clear that the new source of backtracking can be completely avoided, because observing the fields of actuals at run-time does not give an accurate description of their corresponding static types. Consider the following example.

```
def projFG(p,q){ if3 p.f > p.g then p else q }
def rememberF(r){ if4 true then r else {f = 1} }
def main(){ let o = {f=1;g=2} in projFG(o,rememberF(o)) }
```

At run-time, both values passed to `projFG` have fields `f` and `g`, so we might be tempted to conclude that this call site would be well-typed if `p` and `q` share the same type variable $X_{\mathtt{p}}$. Statically, however, the second actual has only `f` because it passes through `rememberF`. Thus, the run-time information about record actuals is not sufficent to deduce that $X_{\mathtt{p}}$ and $X_{\mathtt{q}}$ must be kept separate. On the other hand, we can be sure if two type variables must be kept separate, since if some run-time actual does not have a required field, then its static type will certainly not. For example, the call site `projFG(o,{})` would rule out the possibility of sharing the same type variable for `p` and `q`.

Thus, our modified type inference can rule out situations in which sharing bounded type variables is certainly not possible, so bounds must be kept separate. We can avoid backtracking in this case, since the run-time log will have all fields that might be used, as was the case for System E. If the log does not rule out sharing, the inference algorithm attempts to share the same type variable and proceeds. If a call site is ill-typed, then a restart is necessary so that separate type variables are used. At this point, the log contains all additional fields that will be needed, so further backtracking is avoided. Thus, using run-time logs improves the fully-static, iterative approach but does not completely eliminate the need for iteration.

6 Conclusion

We have sketched out static type inference algorithms for several versions of System E that lack principal types and then improved them with information from run-time executions. Clearly, the formal development of the desired soundness

and completeness properties must be addressed in future work. Furthermore, additional features like recursion, recursive types, nominal object types, and most importantly, higher-order functions, need to be studied for their amenability to inference with run-time logs. In particular, it will be important to determine whether partial or full type inference for System F becomes any easier with the presence of run-time logs. If so, then we may be able to apply these techniques to inferring types for realistic dynamic languages, since many object systems require System F-based type theories. If not, then more heuristic-based approaches will likely be required.

References

1. Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. In *POPL*, 2011.
2. Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to c#. In *ECOOP*, 2010.
3. Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 1999.
4. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 1985.
5. Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA*, 2009.
6. Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *SAC*, 2009.
7. Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating scheme to ml. In *Functional Programming Languages and Computer Architecture*, 1995.
8. Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *POPL*, 1988.
9. Benjamin C. Pierce. *Types and Programming Languages*. 2002.
10. Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.
11. Jeremy Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, 2007.
12. Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In *FOOL*, 1997.
13. Peter Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, 2005.
14. Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *POPL*, 2008.
15. Joe Wells. Typability and type checking in the second-order lambda calculus are equivalent and undecidable. In *LICS*, 1994.

A Some Additional Definitions for System E$^-$

$$\frac{}{Call(X_{y.l}, \tau) \vdash (\emptyset, [X_{y.l} \mapsto \tau])}$$

$$\frac{}{Call(B, B) \vdash (\emptyset, [])} \qquad \frac{}{Call(B, X_{y.l}) \vdash (\{X_{y.l} = B\}, [])}$$

$$\frac{\begin{array}{c}\forall i.\ Call(\tau_i, X_{y.l.f_i}) \vdash (C_i, \theta_i) \\ C = \cup_i\ (\{X_{y.l} \leq \{\, f_i : X_{y.l.f_i} \,\}\} \cup C_i) \\ \theta = \theta_1 \circ \cdots \circ \theta_n\end{array}}{Call(\{\, \overline{f} : \overline{\tau} \,\}, X_{y.l}) \vdash (C, \theta)} \qquad \frac{\begin{array}{c}\forall i.\ \exists ji.\ f_i = g_{ji} \\ Call(\tau_i, \pi_{ji}) \vdash (C_i, \theta_i) \\ C = \cup_i\ C_i \qquad \theta = \theta_1 \circ \cdots \circ \theta_n\end{array}}{Call(\{\, \overline{f_i} : \overline{\tau} \,\}, \{\, \overline{g} : \overline{\pi} \,\}) \vdash (C, \theta)}$$

$$\frac{\begin{array}{c}\forall i.\ \exists ji.\ f_i = g_{ji} \\ \exists i'.\ Call(\tau_{i'}, \pi_{ji'}) \nwarrow C\end{array}}{Call(\{\, \overline{f} : \overline{\tau} \,\}, \{\, \overline{g} : \overline{\pi} \,\}) \nwarrow C} \qquad \frac{\begin{array}{c}\exists i'.\ \forall j.\ f_{i'} \neq g_j \\ C = \{X_{y.l} \leq \{\, f_{i'} : X_{y.l.f_{i'}} \,\}\}\end{array}}{Call(\{\, \overline{f} : \overline{\tau} \,\}, \{\, \overline{g} : \overline{\pi} \,\}_{y.l}) \nwarrow C}$$

$Consistent(C) = true$ iff
1. If $X_{y.l} = B \in C$ and $X_{y.l} = B' \in C$, then $B = B'$
2. If $X_{y.l} = B \in C$ and $X_{y'.l'} \leq \tau \in C$, then $(y, l) \neq (y', l')$

$$TypOf(X_{y.l}, C) = \begin{cases} B & \text{if } X_{y.l} = B \in C \\ \{\, f_k : \theta_k X_{y.l.f_k} \,\}_{y.l} & \text{otherwise, where} \\ & \quad X_{y.l} \leq \{\, f_k : X_{y.l.f_k} \,\} \in C \\ & \quad \theta_k = TypOf(X_{y.l.f_k}, C) \end{cases}$$

$$Solve(C) = \begin{cases} [X_{y.l} \mapsto \tau_{y.l}] & \text{if } Consistent(C) = true \\ & \quad \text{where } \tau_{y.l} = TypOf(X_{y.l}, C) \\ & \quad \text{for each } X_{y.l} \in C \\ fail & \text{if } Consistent(C) = false \end{cases}$$

B Some Additional Definitions for System E

$$\begin{aligned} Path(\tau, []) &= \tau \\ Path(B, f :: l) &= fail \\ Path(\{\, \overline{g} : \overline{\tau} \,\}, f :: l) &= \begin{cases} Path(\tau_j, l) & \text{if } \exists j.\ f = g_j \\ fail & \text{ow} \end{cases} \\ Path(X_{x.l'}, f :: l) &= X_{x.l'.f.l} \end{aligned}$$

$$\begin{aligned} Expand(J, B) &= \{B\} \\ Expand(J, X_{x.l}) &= \{X_{x.l}\} \\ Expand(J, \{\, \overline{f} : \overline{\tau} \,\}) &= \{\{\, \overline{f} : \overline{\tau} \,\}\} \\ Expand(J, X_{k.l}) &= \{\ Path(\tau, l) \mid \tau \in Expand(J, \tau_2) \cup Expand(J, \tau_3)\ \} \\ &\qquad \text{where } J(k) = (\tau_2, \tau_3) \end{aligned}$$