A Flexible Framework for Type Inference with Existential Quantification

Ross Tate[1], Juan Chen[2], and Chris Hawblitzel[2]

[1] University of California, San Diego
[2] Microsoft Research, Redmond

Abstract. Preserving types through compilation guarantees safety of the compiler output without trusting the compiler: the compiler output is annotated with types and can be verified. Type inference can make this technique more practical by inferring most of the needed type information. Existential types, however, are difficult to infer, especially those with implicit pack and unpack operations.
This paper proposes a framework for type inference with existential quantification and demonstrates the framework on a small object-oriented typed assembly language. The framework is flexible enough to support language features such as downward casts, arrays, and interfaces. The framework is based on category theory, the first such application of category theory to type inference to the best of our knowledge.

1 Introduction

Internet users regularly download and execute safe, untrusted code, in the form of Java applets, JavaScript code, Flash scripts, and Silverlight programs. In the past, browsers have interpreted much of this code, but the desire for better performance has recently made just-in-time compilation more common, even for scripting languages. However, compilation poses a security risk: users must trust the compiler, since a buggy compiler could translate safe source code into unsafe assembly language code. Therefore, Necula and Lee [16] and Morrisett *et. al.* [15] introduced *proof-carrying code* and *typed assembly language*. These technologies annotate assembly language with proofs or types that demonstrate the safety of the assembly language, so that the user trusts a small proof verifier or type verifier, rather than trusting a compiler.

Before a verifier can check the annotated assembly language code, someone or something must produce the annotations. Morrisett *et. al.* [15] proposed that a compiler generate these annotations: the compiler preserves enough typing information at each stage of the compilation to generate types or proofs in the final compiler output. The types may evolve from stage to stage; for example, local variable types may change to virtual register types, and then to physical register types and activation record types [15] or stack types [14]. Nevertheless, each stage derives its types from the previous stage's types, and all compiler stages must participate in producing types. Furthermore, typical typed intermediate languages include pseudo-instructions, such as pack and unpack, that coerce one type to another. The compiler stages must also preserve these pseudo-instructions.

Implementing type preservation in an existing compiler may require substantial effort. In our earlier experience [3], we modified 19,000 lines of a 200,000-line compiler to implement type preservation. Even a 10% modification may pose an obstacle to developers trying to retrofit large legacy compilers with type preservation, especially when these modifications require developers to interact with the complex type systems that are typical in typed compiler intermediate languages [15, 14, 13, 3, 9]. To ease the implementation of type preservation in legacy compilers, we'd like to minimize the number of type annotations that the compiler passes from stage to stage, and use *type inference* to fill in the missing types. For example, instead of having each stage explicitly track the type of each register in each basic block, a type inference algorithm could infer the register types (either between each stage, or after compilation).

Type inference is difficult or undecidable for many type systems. Universally and existentially quantified types are challenging to infer [20, 8, 10, 7, 18], and such quantified types are ubiquitous in typed assembly languages. In particular, existentially quantified types describe objects [3, 9], closures [15], and arrays [13] in many typed assembly languages. These existential types are critical to verifying the safety of a program: they ensure that instructions operating on different fields of the same data structure maintain a consistent view of the data. For example, they ensure that the array length checked by an array bounds check instruction matches the array accessed by a subsequent array load or store instruction [13]. To be useful for typed assembly language, type inference must be able to infer operations on existential types across assembly language instructions.

This paper makes two contributions:

1. It proposes a framework, based on category theory, for type inference in type systems with existential quantification. Our application of the framework focuses on quantification over classes, but the framework is also flexible enough to support other forms of quantification, such as quantification over integers [13].
2. It demonstrates the framework on a small object-oriented, class-based typed assembly language "iTal". Specifically, it shows that the framework can infer all the types inside each function of any typeable iTal program, using only the type signature of each function. In particular, iTal requires no type annotations inside a function: no type annotations on interior basic blocks, no type annotations on instructions, and no pseudo-instructions for coercing one type to another.

The framework is not limited to a single language like iTal; it applies to a range of type systems that satisfy certain properties. This is important for scaling up to real-world typed assembly languages, which inevitably require more features (e.g. casts, arrays, interfaces, generics) than found in a simple language like iTal.

2 Background and Related Work

This section motivates the use of existential types in typed assembly languages, and discusses existing approaches to type inference for quantified types.

For many class-based, object-oriented languages without quantified types, type inference is straightforward. For example, Java bytecode omits type annotations from local variables, and uses a forward dataflow analysis to infer these types [11]. The analysis must be able to join types that flow into merge points in the bytecode: if one branch assigns a value of type τ_1 to variable x, and another branch assigns a value of type τ_2 to variable x, then where the two paths merge, x will have type $\tau_1 \sqcup \tau_2$, where $\tau_1 \sqcup \tau_2$ is the least common supertype of τ_1 and τ_2, known as the join (which, even in Java's simple type system, is subtle to define properly [6]).

Like Java bytecode, our framework uses a forward dataflow analysis, but unlike Java bytecode, it supports existential types, and defines joins on existential types. The existential types allow a type system to check individual assembly language instructions for method invocation, array accesses, casts, and other operations (in contrast to Java bytecode, which is forced to treat each of these operations as single, high-level bytecode instructions). Consider the following (incorrect) code for invoking a method on an object p:

```
void Unsafe(Point p, Point q) {
    vt = p.vtable;
    m = vt.method1;
    m(q); }
```

This code above invokes p's method method1, but passes the wrong "this" pointer to method1. The code is unsafe: if p's class is some subclass of `Point`, then method1 may refer to fields defined in the subclass, which q does not necessarily contain, since q may belong to a different subclass of `Point`. Most type systems for object-oriented typed assembly languages use existential types to distinguish between safe method invocations and unsafe code [4, 3, 9]. LIL_C [4], for example, describes p's type and q's type with existential types: both p and q have type $\exists\alpha\ll$`Point`. Ins(α), which says that p and q are each instances of some class α that is a subclass of `Point`, and that the class α contains methods requiring that an instance of α be passed as a "this" pointer. The type system conservatively assumes that the α for p may differ from the α for q, ensuring that only p can be passed to p's methods, and only q can be passed to q's methods.

Type inference Hindley-Milner type inference [12] is used by the ML and Haskell languages. The algorithm for Hindley-Milner inference discovers omitted types by using unification to solve systems of equations between types. For a simple enough type system, this algorithm can infer all types in a program without relying on any programmer-supplied type annotations (unlike our forward dataflow analysis, it does not require a method type signature as a starting point). Unfortunately, this remarkable result does not extend to all type systems. In particular, first-class quantified types are known to make type inference undecidable [20]. Extensions to the Hindley-Milner approach supporting first-class quantified types require some type annotations [8, 10] or pack/unpack annotations [7]. Alternatives to the Hindley-Milner approach, such as local type inference [18], also require some type annotations. Although these extended and alternative algorithms [8, 10, 7, 18] were developed for functional languages, they could be applied to a typed assembly language like iTal by treating each basic block as a (recursive) function. Unfortunately, this would force a compiler to provide type annotations on some of the basic blocks, which our approach avoids.

Much of the difficulty in inferring first-class quantified types stems from the broad range of types that type variables can represent. In the type $\exists\alpha.\ \alpha$, many type systems allow α to represent any type in the type system, including quantified types like $\exists\alpha.\ \alpha$ itself. To simplify type inference, our framework restricts what quantified variables may represent. In this respect, our work is most similar to the Pizza language's internal type system [17], whose existential types quantify over named classes rather than over all types. Although our framework and Pizza's internal type system share some features, such as defining joins for existential types, they differ in important ways. In particular, Pizza's joins exist due to special finiteness properties of the Pizza type system, and therefore are not as broadly applicable as the joins in our framework. In addition, Pizza can only existentially quantify each value separately, but for the applications in this paper it is essential that the same existential bound is shared by many values.

With respect to inference in assembly language, our work is most similar to Coolaid [2], which performs a forward dataflow analysis to infer values and types of registers for code compiled from a Java-like language. Coolaid's inference introduces "symbolic values" to represent unknown values; these symbolic values correspond to existentially quantified variables in iTal's state types. Coolaid is more specialized towards a particular source language and a particular compilation strategy than most typed assembly languages are, whereas iTal encodes objects and classes using more standard, general-purpose types (namely, existential types and bounded quantification). This makes us optimistic that our framework will more easily grow to incorporate more advanced programming language features, such as generics with bounded quantification. Chang *et. al.* [2] state that "We might hope to recover some generality, yet maintain some simplicity, by moving towards an 'object-oriented' TAL". The goal of our framework is to support exactly such an object-oriented TAL (Typed Assembly Language).

3 Language iTal

This section explains a small class-based object-oriented typed assembly language iTal (inferrable Tal), which supports type inference with existential types: type inference can infer all the types inside each function of any typeable iTal program, using only the function signature. It requires no type annotations on basic blocks, no annotations on instructions, and no type coercion instructions.

Appendix A defines a source language and Appendix B describes the translation from the source language to iTal. Appendix C proves the transitivity of state type subtyping. Appendix D proves the soundness of iTal.

The purpose of iTal is to shed some light on the general framework in the next section. iTal focuses on only core object-oriented features such as classes, single inheritance, object layout, field fetch, and virtual method invocation. Our framework is not limited to iTal. It applies to more expressive languages with casts, interfaces, arrays, control stacks, and records. The first three features require more significant changes in the type system and are discussed later in the paper. Support for the other features is mostly straightforward and omitted.

iTal borrows ideas from LIL_C, a low-level object-oriented typed intermediate language [4]. It preserves classes, objects, and subclassing, instead of compiling them away as in most previous object encodings. iTal is even lower level than LIL_C in that iTal makes control flow explicit.

iTal uses type $\text{Ins}(C)$ to represent only objects of "exact" class C, not C's subclasses, unlike most source languages. An existential type $\exists\alpha\ll C.\ \text{Ins}(\alpha)$ represents objects of C and C's subclasses where type variable α indicates the dynamic types of the objects. The subclassing bound C on α means that, the dynamic type α is a subclass of the static type C. The source language type C is translated to the existential type.

The separation between static and dynamic types guarantees the soundness of dynamic dispatch. A type system without such separation cannot detect the error in the "Unsafe" example in the previous section.

Type variables in iTal have subclassing bounds and are instantiated with only class names. The bounds cannot be arbitrary types. This simplifies both type inference and type checking in iTal.

The syntax of iTal is shown below.

class type	ω	$::= \alpha \mid C$
type	τ	$::= \text{Int} \mid \text{Code}(\Phi) \mid \text{Ins}(\omega) \mid \text{Vtable}(\omega) \mid \exists\alpha\ll C.\ \text{Ins}(\alpha)$
value	v	$::= n \mid \ell$
operand	o	$::= v \mid r \mid [r+n]$
instr	ι	$::= \text{bop}\ r, o \mid \text{mov}\ r, o \mid \text{mov}\ [r_1+n], r_2 \mid \text{call}\ o$
binary op	bop	$::= \text{add} \mid \text{sub} \mid \text{mul} \mid \text{div}$
instrs	ιs	$::= \text{jmp}\ \ell \mid \text{jz}\ o, \ell_t, \ell_f \mid \text{ret} \mid \iota; \iota s$
func prec	Φ	$::= \{r_i : \tau_i\}_{i=1}^{n}$
function	f	$::= \Phi\ \{\overrightarrow{\ell : \iota s}\}$
heap value	hv	$::= \mathit{ins}(C)\{v_1, \ldots, v_n\} \mid \mathit{vtable}(C)\{v_1, \ldots, v_n\} \mid f$
heap	H	$::= \bullet \mid H, \ell \mapsto hv$
reg bank	R	$::= \bullet \mid R, r \mapsto v$
frame	F	$::= (R; \iota s)$
prog	P	$::= (H; \overrightarrow{F})$

A class type ω is either a type variable (ranged over by α, β, and γ) or a class name (ranged over by A, B, and C). A special class named Object is a superclass of any other class type.

iTal supports primitive type Int and code pointer type Code(Φ) where Φ describes function signatures. The rest are object-oriented: type Ins(ω) describes objects of "exact" ω; type Vtable(ω) represents the virtual method table of class ω; type $\exists\alpha{\ll}C$. Ins(α) represents objects of C and C's subclasses. The existential type can be used to type fields in objects and registers in function signatures, but not registers in basic block preconditions.

Values in iTal include integers n and heap labels ℓ. Operands include values, registers r, and memory words $[r + n]$ (the value at memory address $r + n$). All values are word-sized.

Instructions in iTal are standard. Instruction bop r, o computes "r bop o" and assigns the result to r. Instruction mov r, o moves the value of o to register r. Instruction mov $[r_1 + n], r_2$ stores the value in r_2 into the memory word at address $r_1 + n$. Instruction call o calls a function o.

Instruction sequences ιs consist of a sequence of the above instructions ended with a control transfer instruction. Instruction jmp ℓ jumps to a block labeled ℓ. Instruction jz o, ℓ_t, ℓ_f branches to ℓ_t if o equals 0 and to ℓ_f otherwise. Instruction ret returns to the caller.

A function Φ $\{\overrightarrow{\ell : \iota s}\}$ specifies its signature Φ and a sequence of basic blocks, each of which has a label ℓ and a body ιs. The notation $\overrightarrow{a}$ means a sequence of items in a. For simplicity, functions do not return values.

iTal has only three kinds of heap values: objects $ins(C)\{v_1, \ldots, v_n\}$ meaning an instance of C with fields $v_1, \ldots, v_n$; vtables $vtable(C)\{v_1, \ldots, v_n\}$ meaning the vtable of C with methods $v_1, \ldots, v_n$; and functions.

A program contains a heap H and a sequence of program frames. The heap is a mapping from heap labels to heap values. Each program frame consists of a register bank and an instruction sequence. A register bank is a mapping from registers to values. iTal uses program frames to simulate the control stacks and the return addresses. Each frame represents the state of a function that is currently called and has not yet returned: the register bank represents the state of registers, and the instruction sequence represents what instructions are to be executed. The leftmost frame is for the function that is currently executed. When calling a function, the execution puts a new frame to the left of the existing frames and copies the register bank of the caller to the callee. When the callee returns, the frame for the callee is removed and the control transfers back to the caller to the instruction right after the call.

Using program frames results in simple type systems but maybe unconventional abstract machines. Supporting realistic control stacks is orthogonal to the core idea of this paper.

Static Semantics We first introduce the environments used by the static semantics.

class decl	cdecl	$::= C : B\{\tau_1, \ldots, \tau_n, \Phi_1, \ldots, \Phi_m\}$
class decls	Θ	$::= \bullet \mid \Theta, \text{cdecl}$
heap value env	Ψ_v	$::= \bullet \mid \Psi_v, \ell : \tau$
code label env	Ψ_c	$::= \bullet \mid \Psi_c, \ell : \Sigma$
constr env	Δ	$::= \bullet \mid \Delta, \alpha \ll C$
reg type	Γ	$::= \bullet \mid \Gamma, r : \tau$
state type	Σ	$::= \exists \Delta.\Gamma$

A class declaration $C : B\{\tau_1, \ldots, \tau_n, \Phi_1, \ldots, \Phi_m\}$ introduces a class C with superclass B and fields types $\tau_1, \ldots, \tau_n$ and methods signatures $\Phi_1, \ldots, \Phi_m$. It specifies all fields and methods of C, including those from the superclasses. Method bodies are translated to functions on the heap. Therefore, class declarations contain only method signatures. Θ is a sequence of class declarations, which the compiler preserves to iTal.

Heap value environment Ψ_v maps heap value labels to their types, and code label environment Ψ_c maps basic block labels to their preconditions (state types). Ψ_c can be a partial map if there is unreachable code. Domains of Ψ_v and Ψ_c are disjoint. The compiler preserves Ψ_v during the compilation. The type inference computes Ψ_c from Ψ_v. The type checker checks each basic block, given Ψ_c. We use Ψ to denote the union of Ψ_v and Ψ_c.

The constraint environment Δ is a sequence of type variables and their bounds. Each type variable has a superclass bound. The register bank type Γ is a partial map from registers to types.

iTal uses state types Σ, another form of existential types, to represent preconditions of basic blocks. A state type $\exists\Delta.\Gamma$ specifies a constraint environment Δ and a register bank type Γ. iTal automatically "unpacks" a register when it is assigned a value with an existential type $\exists\alpha{\ll}C.\ \text{Ins}(\alpha)$: the existentially quantified type variable is lifted to the constraint environment of the state type, and the register has an instance type. In a state type $\exists\Delta.\Gamma$, Δ records the type variables for the "unpacked" registers so far, and Γ never maps a register to an existential type $\exists\alpha{\ll}C.\ \text{Ins}(\alpha)$. This convention eliminates the explicit "unpack" and makes type inference and type checking easier. Rules corresponding to the traditional "pack" operation will be explained later in the section.

State type Σ in iTal can be viewed as $\Sigma \to \text{void}$ in traditional type systems[3].

Subclassing. iTal preserves source-level subclassing between class names. Judgment $\Theta; \Delta \vdash \omega_1 \ll \omega_2$ means that under the class declarations Θ and the constraint environment Δ, class type ω_1 is a subclass of ω_2. Subclassing is reflexive (rule sc-ref) and transitive (rule sc-trans). A class C is a subclass of B if C declares so in its declaration (rule sc-class). A type variable α is a subclass of a class name C if C is α's bound (rule sc-var).

[3] Therefore, the existentially quantified type variables in Σ can be lifted out to the top level and become universally quantified.

$$\frac{\Theta; \Delta \vdash \omega}{\Theta; \Delta \vdash \omega \ll \omega}\text{ sc-ref} \qquad \frac{\Theta; \Delta \vdash \omega_1 \ll \omega_2 \quad \Theta; \Delta \vdash \omega_2 \ll \omega_3}{\Theta; \Delta \vdash \omega_1 \ll \omega_3}\text{ sc-trans}$$

$$\frac{\Theta(C) = C : B\{\ldots\}}{\Theta; \Delta \vdash C \ll B}\text{ sc-class} \qquad \frac{\alpha \ll C \in \Delta}{\Theta; \Delta \vdash \alpha \ll C}\text{ sc-var}$$

Subtyping between State Types. Subtyping between two state types is used to check control transfer. The judgment $\Theta \vdash \Sigma_1 \leq \Sigma_2$ means that under class declarations Θ, state type Σ_1 is a subtype of Σ_2.

$$\frac{\begin{array}{c}\theta : dom(\Delta') \to (dom(\Delta) \cup dom(\Theta)) \\ \forall r \in dom(\Gamma'),\ \Theta; \Delta \vdash \Gamma(r) = \Gamma'(r)[\theta] \\ \forall \alpha \ll C \in \Delta',\ \Theta; \Delta \vdash \theta(\alpha) \ll C\end{array}}{\Theta \vdash (\exists \Delta.\Gamma) \leq (\exists \Delta'.\Gamma')}\text{ st-sub}$$

The non-standard rule st-sub is the key to type inference in iTal. It allows subtyping between two state types without first unpacking one type and then packing to the second type. No type coercion is necessary.

A state type $\exists \Delta.\Gamma$ is a subtype of $\exists \Delta'.\Gamma'$, if there is a substitution θ that maps any type variable in Δ' to either a type variable in Δ or a constant class name in Θ, such that $\Gamma'(r)$ after substitution is the same as $\Gamma(r)$ for all registers in Γ'. The substitution needs to preserve subclassing in Δ', that is, for each constraint $\alpha \ll C$ in Δ', $\theta(\alpha)$ should be a subclass of C under Δ. The substitution is computed during type inference and made ready to use by the type checker.

We can derive that $\Theta \vdash \exists \Delta.(\Gamma, r : \text{Ins}(C)) \leq \exists(\Delta, \alpha \ll C).(\Gamma, r : \text{Ins}(\alpha))$ from st-sub, one case of the traditional "pack" rule for existential types.

Subtyping between state types is reflexive and transitive, as implied by the st-sub rule. For reflexivity, we can use the identity substitution. For transitivity, if $\Theta \vdash (\exists \Delta_1.\Gamma_1) \leq (\exists \Delta_2.\Gamma_2)$, and $\Theta \vdash (\exists \Delta_2.\Gamma_2) \leq (\exists \Delta_3.\Gamma_3)$, then there is a substitution $\theta_1 : \Delta_2 \to (\Delta_1 \cup \Theta)$ and another substitution $\theta_2 : \Delta_3 \to (\Delta_2 \cup \Theta)$. The composition of θ_1 and θ_3 is a mapping from Δ_3 to $\Delta_1 \cup \Theta$ and satisfies the requirements for subtyping between $\exists \Delta_1.\Gamma_1$ and $\exists \Delta_3.\Gamma_3$.

Before explaining the typing rules for instructions, we describe some judgments those typing rules use.

Value Typing. The judgment $\Theta; \Psi \vdash v : \tau$ means that under the class declarations Θ and the heap environment Ψ, value v has type τ. The two rules for integers and heap labels are straightforward.

$$\frac{}{\Theta; \Psi \vdash n : \text{Int}}\text{ vt-int} \qquad \frac{}{\Theta; \Psi \vdash \ell : \Psi_v(\ell)}\text{ vt-lbl}$$

Operand Typing. The judgment $\Theta; \Psi; \Delta; \Gamma \vdash o : \tau$ means that under environments Θ, Ψ, Δ, and Γ, operand o has type τ. The rule ot-vtable means that the first field of an object with type $\text{Ins}(\omega)$ is the vtable (with type $\text{Vtable}(\omega)$). The rules ot-field and ot-meth specify field fetch from objects and method fetch from vtables respectively. Both rules use class declarations in Θ to decide which field or method to access. The helper function *mtype* is defined as $\mathit{mtype}(\omega, \{r_i : \tau_i\}_{i=1}^{n}) = (r_{\text{this}} : \omega, \{r_i : \tau_i\}_{i=1}^{n})$ where r_{this} is a special register to hold the "this" pointer.

$$\frac{\Theta; \Psi \vdash v : \tau}{\Theta; \Psi; \Delta; \Gamma \vdash v : \tau}\ \text{ot-v} \qquad \frac{}{\Theta; \Psi; \Delta; \Gamma \vdash r : \Gamma(r)}\ \text{ot-r}$$

$$\frac{\Theta; \Psi; \Delta; \Gamma \vdash r : \text{Ins}(\omega)}{\Theta; \Psi; \Delta; \Gamma \vdash [r + 0] : \text{Vtable}(\omega)}\ \text{ot-vtable}$$

$$\frac{\begin{array}{c}\Theta; \Psi; \Delta; \Gamma \vdash r : \text{Ins}(\omega) \quad \Theta; \Delta \vdash \omega \ll C \\ \Theta(C) = C : B\{\tau_1, \ldots, \tau_n, \Phi_1, \ldots, \Phi_m\} \quad 1 \le j \le n\end{array}}{\Theta; \Psi; \Delta; \Gamma \vdash [r + j] : \tau_j}\ \text{ot-field}$$

$$\frac{\begin{array}{c}\Theta; \Psi; \Delta; \Gamma \vdash r : \text{Vtable}(\omega) \quad \Theta; \Delta \vdash \omega \ll C \\ \Theta(C) = C : B\{\tau_1, \ldots, \tau_n, \Phi_1, \ldots, \Phi_m\} \quad 1 \le k \le m\end{array}}{\Theta; \Psi; \Delta; \Gamma \vdash [r + k] : \text{Code}(\mathit{mtype}(\omega, \Phi_k))}\ \text{ot-meth}$$

Assignability. The judgment $\Theta; \Delta \vdash \tau_1 \hookleftarrow \tau_2$ means that under the environments Θ and Δ, a value of type τ_2 can be assigned to a memory location with type τ_1. In most type systems assignability is the same as subtyping: $\Theta; \Delta \vdash \tau_1 \hookleftarrow \tau_2$ if τ_2 is a subtype of τ_1. iTal uses assignability to avoid confusion with subtyping between state types. The rule at-pack handles "packing" subclass objects to superclass objects (with existential types).

$$\frac{}{\Theta; \Delta \vdash \tau \hookleftarrow \tau}\ \text{at-ref} \qquad \frac{\Theta; \Delta \vdash \omega \ll C}{\Theta; \Delta \vdash \exists \alpha {\ll} C.\ \text{Ins}(\alpha) \hookleftarrow \text{Ins}(\omega)}\ \text{at-pack}$$

Assign Registers. The judgment $\Delta; \Gamma \vdash \{r \leftarrow \tau\}(\Delta'; \Gamma')$ means that assigning a value of type τ to register r changes the constraint environment to Δ' and the register type to Γ'. When τ is an existential type $\exists \alpha {\ll} C.\ \text{Ins}(\alpha)$ (rule asgn-e), the assignment automatically lifts the existentially quantified type variable to the constraint environment so the register does not have an existential type.

$$\frac{\begin{array}{c}\Delta' = \Delta; \beta \ll C \\ \Gamma' = \Gamma[r : \text{Ins}(\beta)] \quad \beta \notin \mathit{dom}(\Delta)\end{array}}{\Delta; \Gamma \vdash \{r \leftarrow \exists \alpha {\ll} C.\ \text{Ins}(\alpha)\}(\Delta'; \Gamma')}\ \text{asgn-e} \qquad \frac{\tau \ne \exists \alpha {\ll} C.\ \text{Ins}(\alpha)}{\Delta; \Gamma \vdash \{r \leftarrow \tau\}(\Delta; \Gamma[r : \tau])}\ \text{asgn}$$

Instruction Sequence Typing. The judgment $\Theta; \Psi; \Delta; \Gamma \vdash \iota s$ means that under environments Θ, Ψ, Δ, and Γ, instruction sequence ιs is well-typed. Checking the control transfer instructions (t-jmp and t-je) uses subtyping between state types to guarantee control flow safety. Rule t-call uses assignability to guarantee that arguments can be assigned to formals, and t-movM guarantees weak updates of memory locations (object fields).

When checking the control transfer instructions jmp and je (rules t-jmp and t-je), the checker requires that the current state type be a subtype of the precondition of the target.

In rule t-ret, checking the return instruction does nothing because iTal requires that the caller save the program state before calling the callee, and that the return instruction restore the program state.

Rule t-call checks that the call target has a code type, and that types for parameters match those in the precondition, and that the rest of the instructions is well-typed under the current environments.

Rule t-bop requires that both operands of a binary operation be integers. The environments are unchanged. Rule t-movR checks moving a value to a register, which might change the environments. The rest of the instruction sequence is checked under the new environments. Rule t-movM checks moving a value to a memory location. In iTal, the only memory locations that can be updated are object fields and only strong updates (updates with a value of the same type as before) are allowed. The environments are unchanged. Here, the rule uses the assignability rule to guarantee strong updates.

$$\frac{}{\Theta;\Psi;\Delta;\Gamma \vdash \text{ret}}\ \text{t-ret} \qquad \frac{\Theta \vdash \exists\Delta.\Gamma \leq \Psi_c(\ell)}{\Theta;\Psi;\Delta;\Gamma \vdash \text{jmp}\ \ell}\ \text{t-jmp}$$

$$\frac{\Theta;\Psi;\Delta;\Gamma \vdash o : \text{Int} \quad \Theta \vdash \exists\Delta.\Gamma \leq \Psi_c(\ell_t) \quad \Theta \vdash \exists\Delta.\Gamma \leq \Psi_c(\ell_f)}{\Theta;\Psi;\Delta;\Gamma \vdash \text{jz}\ o, \ell_t, \ell_f}\ \text{t-je}$$

$$\frac{\begin{array}{c}\Theta;\Psi;\Delta;\Gamma \vdash o : \text{Code}(\{r_i : \tau_i\}_{i=1}^{n}) \quad \Theta;\Psi;\Delta;\Gamma \vdash \iota s \\ \Theta;\Delta \vdash \tau_i \hookleftarrow \Gamma(r_i) \ \ \forall 1 \leq i \leq n\end{array}}{\Theta;\Psi;\Delta;\Gamma \vdash \text{call}\ o;\iota s}\ \text{t-call}$$

$$\frac{\Theta;\Psi;\Delta;\Gamma \vdash r : \text{Int} \quad \Theta;\Psi;\Delta;\Gamma \vdash o : \text{Int} \quad \Theta;\Psi;\Delta;\Gamma \vdash \iota s}{\Theta;\Psi;\Delta;\Gamma \vdash \text{bop}\ r, o;\iota s}\ \text{t-bop}$$

$$\frac{\Theta;\Psi;\Delta;\Gamma \vdash o : \tau \quad \Delta;\Gamma \vdash \{r \leftarrow \tau\}(\Delta';\Gamma') \quad \Theta;\Psi;\Delta';\Gamma' \vdash \iota s}{\Theta;\Psi;\Delta;\Gamma \vdash \text{mov}\ r, o;\iota s}\ \text{t-movR}$$

$$\frac{\begin{array}{c}\Theta;\Psi;\Delta;\Gamma \vdash r_1 : \text{Ins}(\omega) \quad \Theta;\Psi;\Delta;\Gamma \vdash [r_1 + m] : \tau_m \\ \Theta;\Psi;\Delta;\Gamma \vdash r_2 : \tau \quad \Theta;\Delta \vdash \tau_m \hookleftarrow \tau \quad \Theta;\Psi;\Delta;\Gamma \vdash \iota s\end{array}}{\Theta;\Psi;\Delta;\Gamma \vdash \text{mov}\ [r_1 + m], r_2;\iota s}\ \text{t-movM}$$

Program Typing. The judgment $\Theta;\Psi \vdash P$ means that under the environments Θ and Ψ, the program P is well-typed. As explained previously, iTal uses program frames to simulate the call stack. A well-typed program requires that each program frame in the program be well-typed (rule t-prog): in each frame, the register bank is typed and the instruction sequence is well-typed under the register bank type.

$$\frac{\begin{array}{c}\vdash \Theta \quad \Theta \vdash H : \Psi \\ F_i = (R_i, \iota s_i) \quad \Theta; \Psi \vdash R_i : \Gamma_i \quad \Theta; \Psi; \bullet; \Gamma_i \vdash \iota s_i \;\; \forall 1 \leq i \leq n\end{array}}{\Theta; \Psi \vdash (H; \overrightarrow{F})} \text{ t-prog}$$

The rest of the static semantics is straightforward and will be explained briefly.

Judgment $\vdash \Theta$ specifies well-formedness of class declarations. Judgment $\Theta \vdash$ cdecl specifies well-formedness of one class declaration. Each class declaration in Θ needs to be well-formed to make Θ well-formed. If a class has a superclass, the superclass has to appear before the subclass to avoid cycles in subclassing. Each class declaration contains types for all its fields and methods, including those from superclasses.

$$\frac{\Theta = \text{cdecl}_1, \ldots, \text{cdecl}_n \quad \Theta \vdash \text{cdecl}_i \;\; \forall 1 \leq i \leq n}{\vdash \Theta} \text{ t-cdecls}$$

$$\frac{\begin{array}{c}\Theta \vdash B : \_\{\tau_1, \ldots, \tau_j, \Phi_1, \ldots, \Phi_k\} \\ B \text{ appears before } C \text{ in } \Theta, j \leq n, k \leq m \\ \Theta; \bullet \vdash \tau_i \;\; \forall j < i \leq n \quad \Theta; \bullet \vdash \text{Code}(\Phi_i) \;\; \forall k < i \leq m\end{array}}{\Theta \vdash C : B\{\tau_1, \ldots, \tau_n, \Phi_1, \ldots, \Phi_m\}} \text{ t-cdecl}$$

Judgment $\Theta \vdash H : \Psi$ means that the heap H has type Ψ.

$$\frac{dom(H) = dom(\Psi) \quad \Theta; \Psi \vdash H(\ell) : \Psi(\ell) \;\forall\; \ell \in dom(H)}{\Theta \vdash H : \Psi} \text{ t-heap}$$

Judgment $\Theta; \Psi \vdash R : \Gamma$ means that the register bank R has type Γ.

$$\frac{dom(R) = dom(\Gamma) \quad \Theta; \Psi \vdash R(r) : \Gamma(r) \;\forall\; r \in dom(R)}{\Theta; \Psi \vdash R : \Gamma} \text{ t-rb}$$

Judgment $\Theta; \Delta' \vdash \exists\Delta.\Gamma$ means that under environments Θ and Δ', the state type $\exists\Delta.\Gamma$ is well-formed. For each register r in the domain of R, $\Gamma(r)$ cannot be an existential type $\exists\alpha{\ll}\omega.\ \text{Ins}(\alpha)$.

$$\frac{\Theta; \Delta', \Delta \vdash \Gamma(r) \quad \Gamma(r) \neq \exists\alpha{\ll}\omega.\ \text{Ins}(\alpha) \;\forall r \in dom(\Gamma)}{\Theta; \Delta' \vdash \exists\Delta.\Gamma} \text{ w-st}$$

Judgment $\Theta; \Delta \vdash \tau$ means that under environments Θ and Δ, type τ is well-formed. The free type variables in τ should be in Δ.

$$\frac{\text{freeTv}(\tau) \subseteq dom(\Delta)}{\Theta; \Delta \vdash \tau} \text{ wt}$$

Judgment $\Theta;\Psi \vdash hv : \tau$ means that under environments Θ and Δ, heap value hv has type τ. Rules hv-obj and hv-vtable check objects and vtables respectively. Fields in an objet should be assigned values with compatible types. Rule hv-func checks a function. The precondition of the first basic block is the function signature with all registers unpacked. The helper "$unpack(\Phi)$" is defined as:

$$\begin{array}{lll}
unpack(\bullet) & = \exists\bullet.\bullet & \\
unpack(\Phi, r:\tau) & = \exists(\Delta, \beta \ll C).(\Gamma, r : \mathrm{Ins}(\beta)) & \text{if } \begin{pmatrix} unpack(\Phi) = \exists\Delta.\Gamma \\ \tau = \exists\alpha{\ll}C.\ \mathrm{Ins}(\alpha) \end{pmatrix} \\
unpack(\Phi, r:\tau) & = \exists\Delta.(\Gamma, r:\tau) & \text{if } \begin{pmatrix} unpack(\Phi) = \exists\Delta.\Gamma \\ \tau \neq \exists\alpha{\ll}C.\ \mathrm{Ins}(\alpha) \end{pmatrix}
\end{array}$$

$$\frac{\begin{array}{c}\Delta(C) = C : B\{\tau'_1, \ldots, \tau'_n, \overrightarrow{\Phi}\} \quad \Theta;\Psi \vdash v_0 : \mathrm{Vtable}(C) \\ \Theta;\Psi \vdash v_i : \tau_i \quad \Theta;\bullet \vdash \tau'_i \hookleftarrow \tau_i \forall 1 \le i \le n\end{array}}{\Theta;\Psi \vdash ins(C)\{v_0, v_1, \ldots, v_n\} : \mathrm{Ins}(C)} \text{ hv-obj}$$

$$\frac{\begin{array}{c}\Delta(C) = C : B\{\tau'_1, \ldots, \tau'_n, \Phi_1, \ldots, \Phi_m\} \\ \Theta;\Psi \vdash v_i : \mathrm{Code}(mtype(C, \Phi_i)) \forall 1 \le i \le n\end{array}}{\Theta;\Psi \vdash \mathrm{Vtable}(C)\{v_1, \ldots, v_n\} : \mathrm{Vtable}(C)} \text{ hv-vtable}$$

$$\frac{\begin{array}{c}\Theta;\bullet \vdash \Phi \qquad \Psi_c(\ell_1) = unpack(\Phi) \\ \Psi_c(\ell_i) = \exists\Delta_i.\Gamma_i \quad \Theta;\Psi;\Delta_i;\Gamma_i \vdash \iota s_i \ \forall 1 \le i \le n\end{array}}{\Theta;\Psi \vdash \Phi\ \{\ell_1 : \iota s_1, \ldots, \ell_n : \iota s_n\} : \mathrm{Code}(\Phi)} \text{ hv-func}$$

Dynamic Semantics. This section explains how programs evaluate. Judgment $P \to P'$ means that program P evaluates one step to P'.

When the top frame sees a jump instruction, the instructions for the target block is loaded into the top frame and execution continues (rule jmp). The branch instruction is evaluated similarly.

Evaluating the return instruction simply throws away the top frame (the callee) and goes back to the second frame (the caller) (rule ret). The caller has the state before calling the callee.

Evaluating the move instruction changes the register bank (rule movR) or the heap (rule movM), depending on whether a new value is moved into a register or a memory location.

Evaluating the call instruction creates a new frame as the new top. In the new frame, the values for the parameters are copied from the caller, and the instructions are from the first block of the callee.

$$\frac{H(\ell) = \iota s_\ell}{(H; (R; \text{jmp } \ell) :: \overrightarrow{F}) \to (H; (R; \iota s_\ell) :: \overrightarrow{F})} \text{ jmp}$$

$$\frac{H(\ell_t) = \iota s_t}{(H; (R, \text{jz } 0, \ell_t, \ell_f) :: \overrightarrow{F}) \to (H; (R; \iota s_t) :: \overrightarrow{F})} \text{ jeT}$$

$$\frac{n \neq 0 \quad H(\ell_f) = \iota s_f}{(H; (R; \text{jz } n, \ell_t, \ell_f) :: \overrightarrow{F}) \to (H; (R; \iota s_f) :: \overrightarrow{F})} \text{ jeF}$$

$$\frac{}{(H; (R; \text{ret}) :: \overrightarrow{F}) \to (H; \overrightarrow{F})} \text{ ret}$$

$$\frac{R' = R_1[r \mapsto R(r) \text{ bop } n]}{(H; (R; (\text{bop } r, n; \iota s)) :: \overrightarrow{F}) \to (H; (R'; \iota s) :: \overrightarrow{F})} \text{ bop}$$

$$\frac{}{(H; (R; (\text{mov } r, v; \iota s)) :: \overrightarrow{F}) \to (H; (R[r \mapsto v]; \iota s) :: \overrightarrow{F})} \text{ movR}$$

$$\frac{\begin{array}{c} H(\ell) = \mathit{ins}(C)\{v_1, \ldots, v_n\} \quad 1 \leq m \leq n \\ H' = H[\ell \mapsto \mathit{ins}(C)\{v_1, \ldots, v_{m-1}, v, v_{m+1}, \ldots, v_n\}] \end{array}}{(H; (R; (\text{mov } [\ell + m], v; \iota s)) :: \overrightarrow{F}) \to (H'; (R; \iota s) :: \overrightarrow{F})} \text{ movM}$$

$$\frac{H(\ell) = \Phi \ \{\ell_1 : \iota s_1, \ldots, \ell_n : \iota s_n\} \quad R'(r) = R(r) \ \forall r \in \mathit{dom}(\Phi)}{(H; (R; (\text{call } \ell; \iota s)) :: \overrightarrow{F}) \to (H; (R'; \iota s_1) :: (R; \iota s) :: \overrightarrow{F})} \text{ call}$$

Type Inference Type inference computes preconditions for each basic block in a function from the function signature. The precondition of the entry block is the function signature with all registers unpacked.

Type inference uses a forward data flow analysis, starting from the entry block. For each basic block, if type inference finds a precondition, it then type checks the instruction sequence in the block, until it reaches the control transfer instruction. If the control transfer instruction is "ret", the block is done. Otherwise ("jmp" or "je"), type inference propagates the current state type to the target(s). If a target has no precondition, the current state type will be the new precondition for the target. Otherwise, type inference computes the join of the current state type and the precondition of the target, and uses the result as the new precondition for the target. If the precondition of the target changes, type inference goes through the target again to propagate the changes further.

Type inference continues until it finds a fixpoint. When joining two state types, the result is a super type of the two state types. The type system does not have an infinite chain of super types for any given state type, which guarantees termination of type inference.

Join Computing the join (least upper bound) of two state types is the most important task during type inference. We use $\Sigma_1 \sqcup \Sigma_2$ to represent the join of Σ_1 and Σ_2, and $\hat{\alpha}$ to represent variables in the join (generalization variables).

The join operation is performed in two steps. The first step generalizes the two register bank types to a common super type of both Γ_1 and Γ_2. The generalization is done by replacing every occurrence of class types where Γ_1 and Γ_2 differ with a fresh type variable. The bound of the fresh type variable will be computed in the second step of join. For example, the generalization of $\{r_1 : \text{Ins}(C), r_2 : \text{Ins}(\alpha), r_3 : \text{Ins}(C)\}$ and $\{r_1 : \text{Ins}(D), r_2 : \text{Ins}(\beta), r_3 : \text{Ins}(C)\}$ is $\{r_1 : \text{Ins}(\hat{\gamma_1}), r_2 : \text{Ins}(\hat{\gamma_2}), r_3 : \text{Ins}(C)\}$. The type of r_1 in the join is a fresh type variable $\hat{\gamma_1}$ because the types of r_1 differ in the two register bank types. Similarly, the type of r_2 is another fresh type variable. The type of r_3 is $\text{Ins}(C)$ because the two register bank types agree with the type of r_3.

The generalization omits registers that appear in Γ_1 or Γ_2 but not both. For example, if Γ_1 has types for r_1 and r_2 and Γ_2 has types for r_1 and r_3, the generalization will compute a register bank type with type for r_1 only.

The generalization also remembers where the fresh generalization variables come from. If two types ω_1 and ω_2 are generalized to a type variable $\hat{\alpha}$, then $\hat{\alpha}$ comes from ω_1 in Γ_1 and ω_2 in Γ_2, recorded as $\hat{\alpha} \mapsto \omega_1$ and $\hat{\alpha} \mapsto \omega_2$ respectively. In the previous generalization example, the generalization records two maps: the map for Γ_1 is $\hat{\gamma_1} \mapsto C, \hat{\gamma_2} \mapsto \alpha$, and the map for Γ_2 is $\hat{\gamma_1} \mapsto D, \hat{\gamma_2} \mapsto \beta$. The mappings are called representations for Γ_1 and Γ_2 respectively. They will be used in the second step to compute bounds for $\hat{\gamma_1}$ and $\hat{\gamma_2}$.

The judgment $\vdash Gen(\Gamma_1, \Gamma_2) \rightsquigarrow \Gamma_G : (R_1; R_2)$ means generalizing Γ_1 and Γ_2 to Γ_G. The representation for Γ_1 is R_1 and the one for Γ_2 is R_2.

$$\frac{dom(\Gamma_1) \cap dom(\Gamma_2) = \varnothing}{\vdash Gen(\Gamma_1, \Gamma_2) \rightsquigarrow \varnothing : (\varnothing; \varnothing)} \qquad \frac{\begin{array}{c} Gen(\Gamma_1 \backslash r, \Gamma_2 \backslash r) \rightsquigarrow \Gamma' : (R_1'; R_2') \\ Gen(\Gamma_1(r), \Gamma_2(r)) \text{ undefined} \end{array}}{\vdash Gen(\Gamma_1, \Gamma_2) \rightsquigarrow \Gamma' : (R_1'; R_2')}$$

$$\frac{\begin{array}{c} Gen(\Gamma_1 \backslash r, \Gamma_2 \backslash r) \rightsquigarrow \Gamma' : (R_1'; R_2') \\ (R_1', R_2') \vdash Gen(\Gamma_1(r), \Gamma_2(r)) \rightsquigarrow \tau : (R_1; R_2) \quad \Gamma = \Gamma', r : \tau \end{array}}{\vdash Gen(\Gamma_1, \Gamma_2) \rightsquigarrow \Gamma : (R_1; R_2)}$$

To generalize two register bank types, we need to generalize the types for each register in $dom(\Gamma_1) \cap dom(\Gamma_2)$. The judgment $(R_1, R_2) \vdash Gen(\tau_1, \tau_2) \rightsquigarrow \tau : (R_1'; R_2')$ means that generalizing two types τ_1 and τ_2 results in type τ and extends the representations to R_1' and R_2'. Generalization of types focus on type variables and classes. If τ_1 and τ_2 are the same type τ, the result is τ (that is, the first rule dominates). Otherwise, generalization of two instance types or vtable types introduces a fresh generalization variable $\hat{\alpha}$ and adds to the representations mappings from $\hat{\alpha}$ to the corresponding class types.

$$\frac{}{(R_1, R_2) \vdash \mathit{Gen}(\tau, \tau) \leadsto \tau : (R_1; R_2)}$$

$$\frac{\hat{\alpha} \text{ fresh}}{(R_1, R_2) \vdash \mathit{Gen}(\begin{matrix}\mathrm{Ins}(\omega_1),\\ \mathrm{Ins}(\omega_2)\end{matrix}) \leadsto \mathrm{Ins}(\hat{\alpha}) : (R_1, \hat{\alpha} \mapsto \omega_1; R_2, \hat{\alpha} \mapsto \omega_2)}$$

$$\frac{\hat{\alpha} \text{ fresh}}{(R_1, R_2) \vdash \mathit{Gen}(\begin{matrix}\exists\alpha\ll\omega_1.\ \mathrm{Ins}(\alpha),\\ \exists\beta\ll\omega_2.\ \mathrm{Ins}(\beta)\end{matrix}) \leadsto \exists\gamma\ll\hat{\alpha}.\ \mathrm{Ins}(\gamma) : (R_1, \hat{\alpha} \mapsto \omega_1; R_2, \hat{\alpha} \mapsto \omega_2)}$$

$$\frac{\hat{\alpha} \text{ fresh}}{(R_1, R_2) \vdash \mathit{Gen}(\begin{matrix}\mathrm{Vtable}(\omega_1),\\ \mathrm{Vtable}(\omega_2)\end{matrix}) \leadsto \mathrm{Vtable}(\hat{\alpha}) : (R_1, \hat{\alpha} \mapsto \omega_1; R_2, \hat{\alpha} \mapsto \omega_2)}$$

$$\frac{\begin{matrix}\forall 1 \le i \le n, (R_{1,i-1}, R_{2,i-1}) \vdash \mathit{Gen}(\tau_i, \tau_i') \leadsto \tau_i'' : (R_{1,i}; R_{2,i})\\ R_{1,0} = R_1, R_{2,0} = R_2\end{matrix}}{(R_1, R_2) \vdash \mathit{Gen}(\begin{matrix}\mathrm{Code}(\{r_i : \tau_i\}_{i=1}^n),\\ \mathrm{Code}(\{r_i : \tau_i'\}_{i=1}^n)\end{matrix}) \leadsto \mathrm{Code}(\{r_i : \tau_i''\}_{i=1}^n) : (R_{1,n}; R_{2,n})}$$

The second step of join is factorization of the two representations. The goal is to compute the constraint environment of the result state type. This step is done by unifying equivalent generalization variables.

We define an equivalence relation for generalization variables: $\hat{\alpha} \equiv \hat{\beta}$ if $R_1(\hat{\alpha}) = R_1(\hat{\beta})$ and $R_2(\hat{\alpha}) = R_2(\hat{\beta})$, and use $\langle\hat{\alpha}\rangle$ to denote the arbitrarily chosen representative for the equivalence class where $\hat{\alpha}$ belongs.

The judgment $\mathit{Fact}(R_1, R_2) \leadsto \Delta$ means that factorization of R_1 and R_2 computes Δ, the constraint environment for the result state type. $\mathit{dom}(\Delta)$ is the set of equivalence class representatives. A variable $\hat{\alpha}$ in $\mathit{dom}(\Delta)$ has bound $\mathit{LUB}(R_1(\hat{\alpha}), R_2(\hat{\alpha}))$, the least common superclass name of $R_1(\hat{\alpha})$ and $R_2(\hat{\alpha})$.

The following rule describes the join process: generalization of Γ_1 and Γ_2 produces Γ_G; factorization of two representations produces Δ. The join result contains Δ and Γ_G with all variables $\hat{\alpha}$ replaced with $\langle\hat{\alpha}\rangle$. The join produces two substitutions. The one for $\exists\Delta_1.\Gamma_1$ is $\theta_1 : \mathit{dom}(\Delta) \to (\mathit{dom}(\Delta_1) \cup \mathit{dom}(\Theta))$ defined as $\theta_1(\langle\hat{\alpha}\rangle) = R_1(\hat{\alpha})$, which shows that the join is a super type of $\exists\Delta_1.\Gamma_1$ using rule st-sub. The other substitution is similar.

$$\frac{\begin{matrix}\mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \varnothing \quad \vdash \mathit{Gen}(\Gamma_1, \Gamma_2) \leadsto \Gamma_G : (R_1; R_2)\\ \mathit{Fact}(R_1, R_2) \leadsto \Delta \quad \Gamma = \Gamma_G[\theta] \quad \theta(\hat{\alpha}) = \langle\hat{\alpha}\rangle \forall\hat{\alpha} \in \Gamma_G\end{matrix}}{(\exists\Delta_1.\Gamma_1) \sqcup (\exists\Delta_2.\Gamma_2) = \exists\Delta.\Gamma}$$

Extending iTal iTal contains only enough constructs to support single inheritance, field fetch, and method invocation. For example, each type variable can have only a class name as the superclass bound, as the simple bound is enough to represent objects. Type inference, especially joining two types, is straightforward. iTal may need more expressive type constructs to support more features, making joins more difficult to determine.

For example, supporting downward cast needs existential types with lower type variable bounds (see [4]). Introducing type variable bounds requires multiple variable bounds. Joining $\exists\alpha_1 \ll \beta_1, \beta_1 \ll \gamma_1.\{r_1 : \mathrm{Ins}(\alpha_1), r_2 : \mathrm{Ins}(\beta_1), r_3 : \mathrm{Ins}(\gamma_1)\}$ with $\exists\alpha_2 \ll \gamma_2, \gamma_2 \ll \beta_2.\{r_1 : \mathrm{Ins}(\alpha_2), r_2 : \mathrm{Ins}(\beta_2), r_3 : \mathrm{Ins}(\gamma_2)\}$ results in a state type $\exists\Delta.\{r_1 : \mathrm{Ins}(\alpha), r_2 : \mathrm{Ins}(\beta), r_3 : \mathrm{Ins}(\gamma)\}$ where in Δ, α is a subclass of both β and γ, but β and γ do not have any subclassing relation.

Other language features, such as multiple inheritance, arrays, and generics, require more complex bounds. Records, out parameters, and narrowing returns require more complex types for registers. Given such type systems, it is nontrivial to decide whether the join of two types always exists. The rest of the paper explains a general framework for type inference, which guarantees the existence of a join as long as the type system satisfies the requirements of the framework.

Section 4 gives a high-level overview of the challenges of existential types and the concepts behind this framework. Section 5 provides the details of the framework while providing the reasons for how things are modeled and why they work. Section 6 gives a concrete example of how to find factorization structures, the essential concept behind this framework. Section 7 provides models for common operations besides joins, namely introducing variables and adding inferred constraints or equalities to bounds. Section 8 demonstrates how to add language features by changing the expressibility of existential bounds. Section 9 introduces the current shortcomings of this framework and how we hope to address them. Methods for other common operations, such as introducing variables and adding inferred constraints or equalities, can be found along with other tools in the technical report [19].

4 Overview of the Framework

Existential types are extremely unwieldy. Joins are difficult enough to determine and prove correct even without using existential quantification. When joining existential types, it can be hard to predict what effect changes on the type system will have on the process. A simple change, such as adding subtyping for null pointers, may result in joins not existing. Complex changes, such as adding generics, may turn out to be almost trivial. This framework can help type system designers understand which changes are safe and which are destructive.

For purposes of brevity and good examples we will mostly be using record types with record subtyping. The type $\mathrm{Rec}(A, B, C)$ represents a record whose first field is a pointer to an instance of class A, second field to class B, and third field to class C. The tails of records can be safely forgotten, expressed as the prefix subtyping property $\mathrm{Rec}(A, B, C) \leq \mathrm{Rec}(A, B) \leq \mathrm{Rec}(A)$ and their like.

Existential Modeling Intuitively, any value v modeling $\mathrm{Rec}(C)$ also models $\exists\alpha \ll C.\mathrm{Rec}(\alpha)$. With this intuition, $\mathrm{Rec}(C)$ models $\exists\alpha \ll C.\mathrm{Rec}(\alpha)$. Similarly, Rec($A$) or Rec($B$) also model $\exists\alpha \ll C.\mathrm{Rec}(\alpha)$, supposing A and B extend C. The reasoning behind this is that α can be assigned to A, which translates Rec(α) to Rec(A), and this assignment is valid since $A \ll C$ holds.

Many existential type systems do not incorporate this reasoning into their subtypings since such simple subtyping rules can actually lead to quite complicated subtyping operations. For example, if $B \ll C$ and $X \ll Y$ hold, then the join of $\mathrm{Rec}(B, C, C)$ and $\mathrm{Rec}(X, X, Y)$ is $\exists\alpha \ll \beta \ll \gamma.\mathrm{Rec}(\alpha, \beta, \gamma)$. Three variables have to be introduced, even though both sides only refer to two classes each, and they have to be bound to reflect the left-to-right subclass structure in both records.

This framework is designed to allow this subtyping rule, along with the more complicated one described next, and provide methods for subtypes and joins.

Existential Subtyping Besides subtyping between non-existential types and existential types, we also want subtyping between existential types and existential types. Suppose $\exists vars_1.\tau_1$ is a subtype of $\exists vars_2.\tau_2$. This should mean that any non-existential type τ which models $\exists vars_1.\tau_1$ also models $\exists vars_2.\tau_2$. Ideally any two existential types with this property would be subtypes, but this property may hold for unnatural reasons. For example, the bounds $0 \leq n \leq 1$ and $n^2 = n$ have the same models over the integers, namely $n = 0$ and $n = 1$. This is not a property general to ordered rings though: $0 \leq n \leq 1$ has many more models over the reals than $n^2 = n$. Thus the equivalence of the two bounds over the integers holds for unnatural reasons.

We aim to capture the natural cases. We say that $\exists vars_1.\tau_1$ is a subtype of $\exists vars_2.\tau_2$ if there is a valid assignment of the variables $vars_2$ into the space defined by $vars_1$ such that τ_1 is a subtype of τ_2 after substitution using the assignment. For example, $\exists\alpha \ll \beta \ll C.\mathrm{Rec}(\alpha, C, \beta)$ is a subtype of $\exists\gamma \ll \delta.\mathrm{Rec}(\gamma, \delta)$ as evidenced by assigning γ to α and δ to C due to prefix subtyping. This way any valid assignment of α and β to exact classes can be extended to a valid assignment of γ and δ to exact classes. Note that the names of variables are insignificant: $\exists\alpha \ll \beta \ll C.\mathrm{Rec}(\alpha, C, \beta)$ is also a subtype of $\exists\alpha \ll \beta.\mathrm{Rec}(\alpha, \beta)$.

Structural and General Types In order to reason about joins, we first need to recognize some patterns in type systems. Here we recognize structural types, general types, and generalized subtyping.

Every type has some underlying structure. The structure of $\mathrm{Rec}(\mathrm{Vtable}(C), C)$ is $\mathrm{Rec}(\mathrm{Vtable}(\star), \star)$. The structure forgets the classes being referred to. We call such forms structural types. The space of structural types is much simpler since it is not concerned with classes or variables. A subtyping can be imposed on structural types as well. What should hold is that, since $\mathrm{Rec}(\mathrm{Vtable}(C), C)$ is a subtype of $\mathrm{Rec}(\mathrm{Vtable}(C))$ via prefix subtyping, $\mathrm{Rec}(\mathrm{Vtable}(\star), \star)$ should also be a subtype of $\mathrm{Rec}(\mathrm{Vtable}(\star))$. Thus the function mapping types to their structural type should be subtype-preserving.

Any structural type has a general type as well. The general type for $\mathrm{Rec}(\mathrm{Vtable}(\star), \star)$ is $\mathrm{Rec}(\mathrm{Vtable}(\alpha), \beta)$. Each location which can refer to a class is given its own variable. What makes this type general is that any type can be represented by the general type of its structural type by some mapping of the general variables to actual classes. For example, $\text{Rec}(\text{Vtable}(C), C)$ is represented by $\text{Rec}(\text{Vtable}(\alpha), \beta)$ by the map $(\alpha \mapsto C, \beta \mapsto C)$. This map, which we call a representation, means location α has value C and location β has value C.

Many subtyping rules can be phrased in terms of general types. For example, the subtyping rule for $\text{Rec}(B, C) \leq \text{Rec}(B)$ can be phrased more generally by $\text{Rec}(\alpha, \beta) \leq \text{Rec}(\alpha)$. This subtyping rule has the property that all locations in the supertype come from locations in the subtype. $\text{Rec}(\alpha, \beta)$ is the general type for $\text{Rec}(B, C)$. $\text{Rec}(\gamma)$ is the general type for $\text{Rec}(B)$. The subtyping rule can be rephrased as $\text{Rec}(\alpha, \beta) \leq \text{Rec}(\gamma)[\gamma \mapsto \alpha]$. $(\gamma \mapsto \alpha)$ specifies how locations in the supertype acquire their value from locations in the subtype.

For some type systems all such subtype rules can be phrased in such a manner. For the purposes of brevity and simplicity, we only attempt to model such type systems in the scope of this paper. Other type systems can be modeled but require more complex concepts to do so.

Joins The major challenge of joining existential types is that the variables in one type have no meaning in the other type. This is why many type systems require a bijection of variables in order to share their meaning across the existential quantification boundary. However, this prevents even basic subtyping rules such as $\text{Rec}(C) \leq \exists \alpha \ll C.\text{Rec}(\alpha)$.

The approach we take does not attempt to relate the variables of the different existential types to each other. Instead we use structural types, general types, and generalized subtyping to find a mutual connection between the types being existentially quantified.

Suppose we want to join $\exists \alpha \ll C.\text{Rec}(\alpha, C)$ and $\exists \beta \gg C.\text{Rec}(C, \beta, C)$. In order to deal with the fact that α has no meaning in $\text{Rec}(C, \beta, C)$ and vice-versa, we forget variables and classes altogether and look at the structural types $\text{Rec}(\star, \star)$ and $\text{Rec}(\star, \star, \star)$. The space of structural types is much simpler, so we determine their join in that space: $\text{Rec}(\star, \star)$. Since $\text{Rec}(\star, \star)$ is a subtype of $\text{Rec}(\star, \star)$ and $\text{Rec}(\star, \star, \star)$, we use generalized subtyping to produce a map from the locations in $\text{Rec}(\gamma_1, \gamma_2)$ to the locations in $\text{Rec}(\alpha_1, \alpha_2)$ and $\text{Rec}(\beta_1, \beta_2, \beta_3)$. The locations in $\text{Rec}(\alpha_1, \alpha_2)$ have a representation for $\text{Rec}(\alpha, C)$: $(\alpha_1 \mapsto \alpha, \alpha_2 \mapsto C)$. Similarly, $\text{Rec}(C, \beta, C)$ is represented by $(\beta_1 \mapsto C, \beta_2 \mapsto \beta, \beta_3 \mapsto C)$. The representations and generalized subtype maps can be composed to produce two maps from the locations in $\text{Rec}(\gamma_1, \gamma_2)$: $(\gamma_1 \mapsto \alpha, \gamma_2 \mapsto C)$ and $(\gamma_1 \mapsto C, \gamma_2 \mapsto \beta)$. From this we produce a new representation which has all the properties that both of these representations share. For example, both representations map γ_1 to something which extends C, and γ_2 to something extended by C. Also, both representations map γ_1 to something extending what γ_2 maps to. This results in a new type $\exists \delta \ll C \ll \delta'.\text{Rec}(\delta, \delta')$, which joins the two existential types.

Requirements To summarize, this framework for joining existential types has three requirements. The first is that structural types have joins. The second is that for every subtype rule, the values for the 'locations' in the supertype come from values in the subtype. The third is that bounds and substitutions need to have a factorization structure. This third requirement is explained in the following section, and an example based on iTal is provided in Section 6.

There is a fourth requirement for this framework to be applied practically. The representations for non-existential types need to be valid substitutions of existential types and need to be tight with respect to the factorization structure. Without this, non-existential types either cannot be expressed or cannot be joined using existential types. To see this, consider a traditional usage of existential types for which variables can only map to variables in substitutions. In this type system, $\text{Ins}(C)$ and $\text{Ins}(X)$ do not have the join $\exists\alpha.\text{Ins}(\alpha)$ since α cannot be substituted with C or X in the subtyping since they are not variables. The general type for $\text{Ins}(C)$ is $\text{Ins}(\alpha)$, but the representation $(\alpha \mapsto C)$ is not a valid substitution of existential types since variables cannot map to constants. Hence this traditional usage of existential types does not have joins.

Computability This framework only provides the construction of the join, it does not show that the construction is computable (for reasons described in the future research section). In addition, if this is to be used for abstract interpretation [5], then there is most likely the additional requirement that existential subtyping is well-founded (no infinite sequences of strict supertypes). Although this framework does not answer this question, it does offer some help. If structural subtyping is well-founded, then all that needs to be shown is that, for a fixed number of 'variables', bounds are well-founded with respect to weakening. For example, the bound $\alpha \ll C$ is weaker than the bound $\alpha \ll B$ when B extends C. If subclassing in Θ is well-founded, then weakening of bounds with one variable α is also well-founded.

5 Formalization of the Framework

The concepts and processes described informally above can all be formalized using (mostly basic) category theory. The approach is somewhat unconventional, as are the definitions, so we will explain them in detail. Readers uninterested in formalization may want to just read about factorization structures and skip to Section 6 or jump straight to Section 8. Readers not comfortable with category may prefer our ground-up explanation of this framework [19]

Basic Type Systems Rather than viewing the type $\text{Rec}(\alpha)$ as a type with a free variable α, we view it as a type defined in the environment with a class called α. Similarly, the type $\text{Rec}(C)$ is a type defined in the environment with a class called C. $\text{Rec}(\alpha)$ and $\text{Rec}(C)$ are both types defined in the environment with both classes α and C. At this point, variables and constants are interchangeable. Their distinction only occurs in another context described later.

The map $(\alpha \mapsto C) : \{\alpha\} \to \{C\}$ extends to a substitution mapping types defined in the environment $\{\alpha\}$ to types defined in the environment $\{C\}$. Similarly for the inclusion $(\alpha \mapsto \alpha) : \{\alpha\} \to \{\alpha, C\}$ or the map $(C \mapsto \alpha) : \{C\} \to \{\alpha\}$. Thus maps between environments can be extended to maps between type spaces.

This process can by formalized by a functor $\mathit{Type} : \mathbf{Env} \to \mathbf{Prost}$. $\mathbf{Env}$ is some category of environments. For the sample type system in this section, $\mathbf{Env}$ is partially ordered sets and relation-preserving functions. $\mathbf{Prost}$ is the category of preordered sets and relation-preserving functions. Type assigns an environment $\mathcal{E}$ to the set $\mathit{Type}(\mathcal{E})$ of non-existential types defined in that environment equipped with the subtyping relation in that environment. Type assigns environment maps f to subtype-preserving substitutions $[f]$ mapping types defined in one environment to types defined in another environment.

Existential Types Rather than viewing $\exists\alpha$ as something which bounds the free variable α in $\mathrm{Rec}(\alpha)$, view it as something which extends the environment by introducing a new variable, producing a new environment which $\mathrm{Rec}(\alpha)$ is defined in. Suppose Θ is the environment with no variables representing the class hierarchy defined by the program. $\Theta[\alpha]$ then is the environment Θ plus a class variable α. $\iota : \Theta \to \Theta[\alpha]$ is the map including Θ into $\Theta[\alpha]$. $\exists\alpha$ then extends Θ via ι and $\mathrm{Rec}(\alpha)$ is defined in $\Theta[\alpha]$.

With this in mind we introduce the category $\Theta \downarrow \mathbf{Env}$ of objects over Θ. An object of $\Theta \downarrow \mathbf{Env}$ is a morphism $e : \Theta \to \mathcal{E}$ of $\mathbf{Env}$ from Θ to some environment $\mathcal{E}$. A morphism $f : (\Theta \xrightarrow{e} \mathcal{E}) \to (\Theta \xrightarrow{e'} \mathcal{E}')$ of $\Theta \downarrow \mathbf{Env}$ is a morphism $f : \mathcal{E} \to \mathcal{E}'$ of $\mathbf{Env}$ such that $e;f = e'$. (Note that we use ; instead of $\circ$ for reverse composition.) In other words, f needs to preserve the structure of Θ within $\mathcal{E}$ and $\mathcal{E}'$ specified by e and e'.

Here is where variables and constants become distinguishable. Say Θ is $\{C\}$. Let $\iota_C : \{C\} \to \{C, \alpha\}$ be the inclusion function. Then the map $(C \mapsto \alpha, \alpha \mapsto \alpha) : \{C, \alpha\} \to \{C, \alpha\}$ is not a morphism in $\Theta \downarrow \mathbf{Env}$ from ι_C to ι_C. The additional structure specified by ι_C forces the constant C to stay fixed.

With this an existential type can be viewed as an object $\Theta \xrightarrow{e} \mathcal{E}$ of $\Theta \downarrow \mathbf{Env}$ and a type τ defined in the type space $\mathit{Type}(\mathcal{E})$ for the extended environment. This type is expressed as $\exists e.\tau$. So what was expressed as $\exists\alpha.\mathrm{Rec}(\alpha)$ is now expressed as $\exists\iota.\mathrm{Rec}(\alpha)$ where $\iota : \Theta \to \Theta[\alpha]$. $\exists e.\tau$ is a subtype of $\exists e'.\tau'$ when there is a morphism $f : e' \to e$ of $\Theta \downarrow \mathbf{Env}$ with the property that τ is a subtype of $\tau'[f]$ in the type space for $\mathcal{E}$. Non-existential types can be inserted into this framework by viewing them as types defined in the trivial extension $\mathit{id}_\Theta : \Theta \to \Theta$. $\exists \mathit{id}_\Theta.\tau$ is a subtype of $\exists e'.\tau'$ precisely when τ models $\exists e'.\tau'$ in the sense described earlier.

For purposes of simplification and generality, we will model the bounds for existential types as $\mathbf{Env}_\Theta$ which then has a functor $\mathit{Type} : \mathbf{Env}_\Theta \to \mathbf{Prost}$ specifying the types definable in the bounded space. Typically $\mathbf{Env}_\Theta$ will be $\Theta \downarrow \mathbf{Env}$ or some closely related category, with Type specifying the type space of the extended environment. With this we define the category $\exists\mathbf{Type}_\Theta$. The objects are a pair $\langle e, \tau\rangle$ where e is an object of $\mathbf{Env}_\Theta$ and τ is a type in $\mathit{Type}(e)$. A morphism $f : \langle e, \tau\rangle \to \langle e', \tau'\rangle$ in $\exists\mathbf{Type}_\Theta$ is a morphism $f : e \to e'$ of $\mathbf{Env}_\Theta$

with the property that τ' is a subtype of $\tau[f]$ in $Type(e')$. Thus f serves as evidence that $\exists e'.\tau'$ is a subtype of $\exists e.\tau$. f is called strict if in fact $\tau' = \tau[f]$.

Structural Types The structural type $\mathrm{Rec}(\star)$ can also be interpreted as a type defined in the somewhat trivial environment $\{\star\}$. We model the triviality of $\{\star\}$ more generally as being the terminal object $\bigstar$ of the category $\mathbf{Env}_\Theta$. $\bigstar$ being a terminal object means that every object e of $\mathbf{Env}_\Theta$ has exactly one morphism to $\bigstar$, denoted as $!_e : e \to \bigstar$ (or simply as $!$). So, given a type $\tau \in Type(e)$, the structural type is $\tau[!_e] \in Type(\bigstar)$. A terminal object in **Set**, the category of sets and functions, is a singleton set, such as $\{\star\}$.

General Types $Type(\bigstar)$ is a preordered set, which can be made into a category. $\mathbf{Type}(\bigstar)^{op}$ is the category with objects being structural types and with a unique morphism from τ to τ' when τ is a structural supertype of τ'.

We represent generalization as a functor $Gen_\bigstar : \mathbf{Type}(\bigstar)^{op} \to \exists\mathbf{Type}_\Theta$. $Gen_\bigstar(\tau)$ specifies the general type and environment for the structural type τ. For example, $Gen_\bigstar(\mathrm{Rec}(\star,\star))$ is $\langle \Theta[\alpha,\beta], \mathrm{Rec}(\alpha,\beta)\rangle$. Given two structural types $\tau \geq \tau'$, $Gen_\bigstar(\tau \geq \tau') : Gen_\bigstar(\tau) \to Gen_\bigstar(\tau')$ specifies where the locations in τ come from in the locations of τ' to make the subtyping hold. For example, $Gen_\bigstar(\mathrm{Rec}(\star) \geq \mathrm{Rec}(\star,\star)) : \langle \Theta[\gamma], \mathrm{Rec}(\gamma)\rangle \to \langle \Theta[\alpha,\beta], \mathrm{Rec}(\alpha,\beta)\rangle$ is $(\gamma \mapsto \alpha)$. This is the formalization of generalized subtyping.

$Gen_\bigstar$ can be extended to a functor $Gen : \exists\mathbf{Type}_\Theta \to \exists\mathbf{Type}_\Theta$ by $Gen(\langle e,\tau\rangle) = Gen_\bigstar(\tau[!_e])$. Given $f : \langle e,\tau\rangle \to \langle e',\tau'\rangle$, then $\tau[f] \geq \tau'$ so $\tau[!_e] = \tau[f][!_{e'}] \geq \tau'[!_{e'}]$. Therefore we can define $Gen(f)$ as $Gen(\tau[!_e] \geq \tau'[!_{e'}])$. Gen maps an existential subtyping to the generelized subtyping behind it.

Representations We model representations by a strict natural transformation $rep : Gen \to Id_{\exists\mathbf{Type}_\Theta}$. This specifies for each $\langle e,\tau\rangle$ a strict morphism $rep_{\langle e,\tau\rangle} : Gen_\bigstar(\tau[!]) \to \langle e,\tau\rangle$ mapping the locations in τ to their values in the extended environment e. Naturality captures the idea that representations interact well with subtyping and substitution.

Factorization Structures Here we need to investigate a lesser known concept from category theory known as factorization structures. We do so by observing properties of **Set** and then generalizing to other categories.

In **Set**, any function $f : \mathcal{X} \to \mathcal{Y}$ can be factored into a surjection and an injection: $\mathcal{X} \xrightarrow{f^*} f[\mathcal{X}] \xhookrightarrow{im(f)} \mathcal{Y}$. $f[\mathcal{X}]$ is the image of f and can be defined in two ways. The first defines a subset of $\mathcal{Y}$: $\{f(x) \mid x \in \mathcal{X}\}$. The second groups elements of $\mathcal{X}$ into equivalence classes: $\mathcal{X}/\sim$ where $x \sim x'$ is defined as $f(x) = f(x')$. $f^* : \mathcal{X} \twoheadrightarrow f[\mathcal{X}]$ is the surjective function mapping elements of $\mathcal{X}$ to their equivalence classes. $im(f) : f[\mathcal{X}] \hookrightarrow \mathcal{Y}$ is the injective inclusion of the subset of $\mathcal{Y}$ into $\mathcal{Y}$. This ability to factor every function into a surjection and injection means **Set** has (Surjection, Injection)-factorizations.

Set has a more interesting but lesser-known property. Suppose the following commutes in **Set**:

$$\begin{tikzcd}
\mathcal{A} \arrow[r, two heads, "e"] \arrow[d, "f"'] & \mathcal{B} \arrow[d, "g"] \\
\mathcal{C} \arrow[r, hook, "m"] & \mathcal{D}
\end{tikzcd}$$

where e is surjective and m is injective. Define $d : \mathcal{B} \to \mathcal{C}$ by $\forall a \in \mathcal{A}.\ d(e(a)) = f(a)$. This turns out to be a well-defined function since e is surjective, m is injective, and the square commutes. d then is the unique diagonal making the following commute:

$$\begin{tikzcd}
\mathcal{A} \arrow[r, two heads, "e"] \arrow[d, "f"'] & \mathcal{B} \arrow[d, "g"] \arrow[dl, dashed, "d"'] \\
\mathcal{C} \arrow[r, hook, "m"'] & \mathcal{D}
\end{tikzcd}$$

The fact that such a d always exists and is unique in these situations means that **Set** has unique (Surjection, Injection)-diagonalizations.

These concepts can be generalized to sources. A source is an indexed family of morphisms all with the same domain, denoted by $(\mathcal{A} \xrightarrow{f_i} \mathcal{B}_i)_{i \in \mathcal{I}}$. A mono-source in **Set** is a source with the property that $\forall a, b \in \mathcal{A}.\ \big(\forall i \in \mathcal{I}.\ f_i(a) = f_i(b)\big) \Rightarrow a = b$, which generalizes injections. In **Set**, any source $(\mathcal{A} \xrightarrow{f_i} \mathcal{B}_i)_{i \in \mathcal{I}}$ can be factored into a surjection $e : \mathcal{A} \twoheadrightarrow \tilde{\mathcal{A}}$ and a mono-source $(\tilde{\mathcal{A}} \xrightarrow{m_i} \mathcal{B}_i)_{i \in \mathcal{I}}$ (meaning $\forall i \in \mathcal{I}.\ e; m_i = f_i$). This property means **Set** has (Surjection, Mono-Source)-factorizations.

Similarly, if the following diagram commutes in **Set** for all $i \in \mathcal{I}$:

$$\begin{tikzcd}
\mathcal{A} \arrow[r, two heads, "e"] \arrow[d, "f"'] & \mathcal{B} \arrow[d, "g_i"] \\
\mathcal{C} \arrow[r, "m_i"] & \mathcal{D}_i
\end{tikzcd}$$

where e is surjective and $(\mathcal{C} \xrightarrow{m_i} \mathcal{D}_i)_{i \in \mathcal{I}}$ is a mono-source, then there is a unique diagonal $d : \mathcal{B} \to \mathcal{C}$ commuting for all $i \in \mathcal{I}$. This property means **Set** has unique (Surjection, Mono-Source)-diagonalizations.

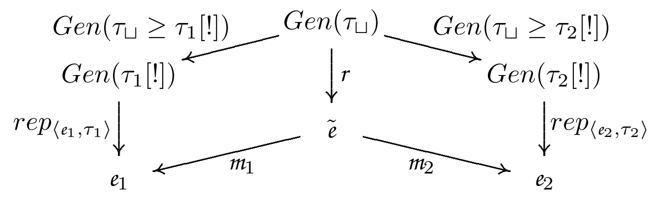
To generalize this, replace **Set** with an arbitrary category, surjection with a class $\mathcal{E}$ of morphisms in that category, and mono-source with a conglomerate $\mathcal{M}$ of sources in that category. A category has an $(\mathcal{E}, \mathcal{M})$ factorization structure if it has both $(\mathcal{E}, \mathcal{M})$-factorizations and unique $(\mathcal{E}, \mathcal{M})$-diagonalizations. We assume $\mathbf{Env}_\Theta$ has some such factorization structure for finite sources.

$\mathcal{E}$ should be as large as possible, ideally being all epimorphisms in the category. The intuition behind factorization is that it makes an intermediate object with as many things true in it as possible. For example, defining $f[\mathcal{X}]$ as ${}^{\mathcal{X}}/_{\sim}$ demonstrates the idea of adding all equalities over $\mathcal{X}$ which f permits.

Tight Bounds The existential type $\exists\alpha \ll \beta.\mathrm{Rec}(\alpha)$ has the unused variable β. For the purposes of joining, the bound of an existential type needs to be tight with respect to the quantified type. We formalize this using the $(\mathcal{E},\mathcal{M})$ factorization structure of $\mathbf{Env}_\Theta$. The bound for $\exists e.\tau$ is tight if $\mathit{rep}_{\langle e,\tau\rangle}$ belongs to $\mathcal{E}$. Typically this means something like every variable introduced by e is used by a location of τ since $\mathit{rep}_{\langle e,\tau\rangle}$ is surjective in some sense.

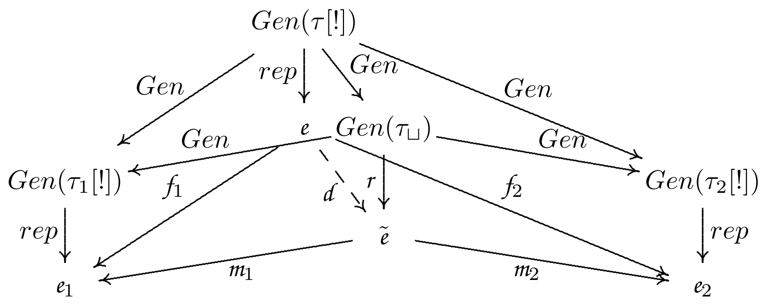
If an existential type is not tight, there is a smallest tight existential supertype. Given $\exists e.\tau$, factor $\mathit{rep}_{\langle e,\tau\rangle} : \mathit{Gen}(\tau[!]) \to e$ into an $\mathcal{E}$-morphism $r : \mathit{Gen}(\tau[!]) \to \tilde{e}$ and an $\mathcal{M}$-morphism $m : \tilde{e} \to e$. Suppose $\tilde{\tau}$ is the general type for τ, then $\exists\tilde{e}.\tilde{\tau}[r]$ is a tight existential supertype of $\exists e.\tau$ as evidenced by m. In addition, $\exists\tilde{e}.\tilde{\tau}[r]$ is a subtype of all other tight existential supertypes of $\exists e.\tau$ because $\mathbf{Env}_\Theta$ has $(\mathcal{E},\mathcal{M})$-diagonalizations.

Joins Given two existential types $\exists e_1.\tau_1$ and $\exists e_2.\tau_2$, the join can be constructed as follows. First, let $\tau_\sqcup = \tau_1[!] \sqcup_\bigstar \tau_2[!]$, the structural join. Then $\tau_1[!] \le \tau_\sqcup$ and $\tau_2[!] \le \tau_\sqcup$, so there is the following 2-source and its $(\mathcal{E},\mathcal{M})$-factorization:


$$\begin{tikzcd}
 & \mathit{Gen}(\tau_\sqcup) \arrow[dl, "\mathit{Gen}(\tau_\sqcup \ge \tau_1[!])"'] \arrow[dr, "\mathit{Gen}(\tau_\sqcup \ge \tau_2[!])"] \arrow[dd, "r"] & \\
\mathit{Gen}(\tau_1[!]) \arrow[d, "\mathit{rep}_{\langle e_1,\tau_1\rangle}"'] & & \mathit{Gen}(\tau_2[!]) \arrow[d, "\mathit{rep}_{\langle e_2,\tau_2\rangle}"] \\
e_1 & \tilde{e} \arrow[l, "m_1"] \arrow[r, "m_2"'] & e_2
\end{tikzcd}$$

Let $\langle\tilde{\mathcal{G}},\tilde{\tau}\rangle$ be $\mathit{Gen}(\tau_\sqcup)$. The join is $\exists\tilde{e}.\tilde{\tau}[r]$, which is a supertype via m_1 and m_2.

Suppose $\exists e.\tau$ is a tight existential supertype of both $\exists e_1.\tau_1$ and $\exists e_2.\tau_2$ as evidenced by $f_1 : e \to e_1$ and $f_2 : e \to e_2$. Then $\tau_1[!] \le \tau[!]$ and $\tau_2[!] \le \tau[!]$ so, by property of joins, $\tau_\sqcup \le \tau[!]$. This means there is the following commutative diagram with induced $(\mathcal{E},\mathcal{M})$-diagonalization d:


$$\begin{tikzcd}
 & & \mathit{Gen}(\tau[!]) \arrow[ddll, "\mathit{Gen}"'] \arrow[d, "\mathit{rep}"'] \arrow[dr, "\mathit{Gen}"] \arrow[ddrr, "\mathit{Gen}"] & & \\
 & & e \arrow[dddll, "f_1"] \arrow[dddrr, "f_2"] \arrow[ddr, dashed, "d"'] & \mathit{Gen}(\tau_\sqcup) \arrow[dlll, "\mathit{Gen}"'] \arrow[dr, "\mathit{Gen}"] \arrow[dd, "r"] & \\
\mathit{Gen}(\tau_1[!]) \arrow[dd, "\mathit{rep}"'] & & & & \mathit{Gen}(\tau_2[!]) \arrow[dd, "\mathit{rep}"] \\
 & & & \tilde{e} \arrow[dlll, "m_1"] \arrow[dr, "m_2"'] & \\
e_1 & & & & e_2
\end{tikzcd}$$

It is simple to verify $\tilde{\tau}[r] \le \tau[d]$. Therefore $\exists\tilde{e}.\tilde{\tau}[r]$ is a subtype of $\exists e.\tau$ via d.

To get an intuition of what is happening, we will break the join process down into a few steps. First $r_1 : \mathit{Gen}(\tau_\sqcup) \to e_1$, defined as $\mathit{Gen}(\tau_\sqcup \ge \tau_1[!]); \mathit{rep}_{\langle e_1,\tau_1\rangle}$, and r_2, defined similarly, specify the values the locations of $\tau_\sqcup$ have in both e_1 and e_2. The factorization produces a representation r in which everything that is true for both r_1 and r_2 is true for r. If two locations map to equal values by both r_1 and r_2, then they will map to equal values by r. If a location maps to values with the same property, such as 'is a subclass of C', then the location will map to a value with that property by r. Therefore the joined existential type will have as few variables as possible with as many constraints as possible.

Requirements An existential type system fits our framework for joining tight existential types if it satisfies the following properties. There is a category $\mathbf{Env}_{\Theta}$ of existential bounds and valid assignments of variables. There is a functor $\mathit{Type} : \mathbf{Env}_{\Theta} \to \mathbf{Prost}$ specifying the non-existential type system for a fixed bound and the substitution for an assignment of variables. There is a choice of finite factorization structure $(\mathcal{E}, \mathcal{M})$ for $\mathbf{Env}_{\Theta}$, preferring $\mathcal{E}$ to be as large as possible. There is a choice of terminal object $\bigstar$ of $\mathbf{Env}_{\Theta}$. The preordered set $\mathit{Type}(\bigstar)$ has joins. There is a functor $\mathit{Gen}_{\bigstar} : \mathbf{Type}(\bigstar)^{op} \to \exists\mathbf{Type}_{\Theta}$. There is a strict natural transformation $\mathit{rep} : \mathit{Gen} \to \mathit{Id}_{\exists\mathbf{Type}_{\Theta}}$.

This may seem like a lot of requirements, but most of them hold for any type system. The key non-trivial requirements are joins for $\mathit{Type}(\bigstar)$, the morphism component of $\mathit{Gen}_{\bigstar}$, and the factorization structure of $\mathbf{Env}_{\Theta}$. There is also one more requirement for practical purposes. There must be a $\Theta \in \mathbf{Env}_{\Theta}$ whose types are the non-existential types, and, for all non-existential types τ in $\mathit{Type}(\Theta)$, $\mathit{rep}_{\langle\Theta,\tau\rangle}$ must belong to $\mathcal{E}$.

The following section provides a concrete construction of a factorization structure. The technical report [19] provides tools for constructing categories with useful factorization structures and other desirable properties. The technical report also has a generalization of this framework using factorization structures for functors [1]. This generalization offers a lot more flexibility and allows generalized types and $\bigstar$ to be defined independently of Θ.

6 Joins in iTal

Joins of structural types in iTal are very simple: produce the structural state type with only the registers mapping to the same structural type in both state types being joined. Subtyping in iTal also has the property that all the locations in supertypes acquire their values from subtypes, since supertypes simply drop registers. The only requirement of the framework left to show is that the category of bounds and substitutions has a useful factorization structure. This section demonstrates how to construct and prove such a factorization structure. Then joins in iTal can be computed using the process defined by the framework, which is what the join algorithm for iTal implements.

A bound in iTal defines a finite set of variables $\mathcal{V}$ and a map $\mathit{bound}_{\mathcal{V}} : \mathcal{V} \to \Theta$. These form the objects $\mathcal{V}$ of $\mathbf{Env}_{\Theta}$. Using $\mathit{bound}_{\mathcal{V}}$, $\mathcal{V}$ and Θ can be combined into a single partially ordered set $\Theta[\mathcal{V}]$. Each valid substitutition defines a map $f : \mathcal{V} \to \Theta[\mathcal{W}]$ with the property that $\forall \alpha \in \mathcal{V}.\ f(\alpha) \ll \mathit{bound}_{\mathcal{V}}(\alpha)$ in $\Theta[\mathcal{W}]$. These form the morphisms f of $\mathbf{Env}_{\Theta}$. f can be extended to $\Theta[\mathcal{V}]$ by mapping elements of Θ to themselves in $\Theta[\mathcal{W}]$.

Here we manually construct the factorization structure of this category. Note that there are many existing constructive theorems on this topic [1, 19] so this process rarely needs to be done by hand. We provide this to give a concrete applied example of the process behind computing and proving joins. The construction relies on the fact that Θ has least common superclasses, which always holds in iTal due to single-inheritance.

$\mathcal{E}$-Morphisms We define $\mathcal{E}$ to be the class of morphisms which are surjective onto variables. More formally, $f : \mathcal{V} \to \mathcal{W}$ belongs to $\mathcal{E}$ whenever $\forall \gamma \in \mathcal{W}.\ \exists \alpha \in \mathcal{V}.\ f(\alpha) = \gamma$. Every variable in $\mathcal{W}$ is mapped to by some variable in $\mathcal{V}$ via f.

$\mathcal{M}$-Sources We define $\mathcal{M}$ to be the conglomorate of mono-sources reflecting subclasses. More formally, $(\mathcal{V} \xrightarrow{f_i} \mathcal{W}_i)_{i\in\mathcal{I}}$ belongs to $\mathcal{M}$ whenever $(\Theta[\mathcal{V}] \xrightarrow{f_i} \Theta[\mathcal{W}_i])_{i\in\mathcal{M}}$ is a mono-source in **Set** and $\forall \alpha \in \mathcal{V}.\ \forall C \in \Theta.\ (\forall i \in \mathcal{I}.\ f_i(\alpha) \ll C \text{ in } \Theta[\mathcal{W}_i]) \Rightarrow \alpha \ll C \text{ in } \Theta[\mathcal{V}]$. The second condition can be interpreted as holding whenever subclassing is defined as strongly as possible in $\mathcal{V}$ while keeping every f_i relation-preserving.

$(\mathcal{E},\mathcal{M})$-Factorizations Given a source $(\mathcal{V} \xrightarrow{f_i} \mathcal{W}_i)_{i\in\mathcal{I}}$ construct the $(\mathcal{E},\mathcal{M})$-factorization as follows. Define the equivalence $\sim$ on $\mathcal{V}$ by $\alpha \sim \beta$ means $\forall i \in \mathcal{I}.\ f_i(\alpha) = f_i(\beta)$. Then $\mathcal{V}/\!\sim$ is the set of $\sim$ equivalence classes of $\mathcal{V}$. Let $\tilde{\mathcal{V}}$ be the subset of $\mathcal{V}/\!\sim$ after removing the equivalence classes which are mapped to the same element of Θ by every f_i (in other words, remove the variables whose assignments are constant across all f_i). Define $\mathit{bound}_{\tilde{\mathcal{V}}}(\langle\alpha\rangle)$ as the least common superclass of $\{\mathit{bound}_{\mathcal{W}_i}(f_i(\alpha)) \mid i \in \mathcal{I}\}$ in Θ.

Define $e : \mathcal{V} \to \Theta[\tilde{\mathcal{V}}]$ as $e(\alpha) = \begin{cases} \langle\alpha\rangle \in \tilde{\mathcal{V}} & \langle\alpha\rangle \\ \langle\alpha\rangle \notin \tilde{\mathcal{V}} & f_i(\alpha) \text{ for arbitrary choice of } i \in \mathcal{I} \end{cases}$.

For each $i \in \mathcal{I}$, define $m_i : \tilde{\mathcal{V}} \to \Theta[\mathcal{W}_i]$ as $m_i(\langle\alpha\rangle) = f_i(\alpha)$. The proofs that these are all proper objects and morphisms, that e belongs to $\mathcal{E}$, and that $(\tilde{\mathcal{V}} \xrightarrow{m_i} \mathcal{W}_i)_{i\in\mathcal{I}}$ belongs to $\mathcal{M}$ are left to the diligent reader.

$(\mathcal{E},\mathcal{M})$-Diagonalizations Now suppose we have $\mathcal{A} \xrightarrow{e} \mathcal{B} \xrightarrow{g_i} \mathcal{D}_i = \mathcal{A} \xrightarrow{f} \mathcal{C} \xrightarrow{m_i} \mathcal{D}_i$ holding for all $i \in \mathcal{I}$ with e in $\mathcal{E}$ and $(\mathcal{C} \xrightarrow{m_i} \mathcal{D}_i)_{i\in\mathcal{I}}$ in $\mathcal{M}$. Construct the function $d : \mathcal{B} \to \Theta[\mathcal{C}]$ by $\forall \alpha \in \mathcal{A}.\ d(e(\alpha)) = f(\alpha)$. d is a well-defined function since e is surjective onto $\mathcal{B}$ and $(\Theta[\mathcal{C}] \xrightarrow{m_i} \Theta[\mathcal{D}_i])_{i\in\mathcal{I}}$ is a mono-source in **Set**.

What still needs to be shown is that $d : \mathcal{B} \to \Theta[\mathcal{C}]$ is valid. Given $\beta \in \mathcal{B}$, then $\forall i \in \mathcal{I}.\ m_i(d(\beta)) = g_i(\beta) \ll \mathit{bound}_{\mathcal{B}}(\beta)$ in $\Theta[\mathcal{D}_i]$. By definition of $\mathcal{M}$, this implies $d(\beta) \ll \mathit{bound}_{\mathcal{B}}(\beta)$ in $\Theta[\mathcal{C}]$. Therefore d is a proper morphism $d : \mathcal{B} \to \mathcal{C}$.

7 Operations in the Framework

Besides subtyping and joining, there are other common operations necessary for working with existential types. Here I formalize the theory behind these operations and why they make sense.

Introducing Variables Often there are types representing nested existential types. In our sample type system, there is the type $\exists\alpha \ll \omega.\mathrm{Rec}(\alpha)$. At certain points the nested existential bound is pulled out into the outer existential. This is done by introducing a new variable bounded by ω. Since this is a common process for existential types, it is useful to express it categorically in the same terms as the framework.

Suppose we have some existential type $\exists \mathit{e}.\tau$. We also have some choice of value $v(\omega)$ for ω in e. We want to build a new bound e' by adding a bounded variable to e. What should hold is that any assignment $a : \mathit{e} \to \Theta$ and any choice of subclass C of $a(v(\omega))$ extends into an assignment $a_C : \mathit{e}' \to \Theta$. The choice of C can be thought of as an assignment $t_C : (\alpha \ll \omega) \to \Theta$ for which $t(\omega) = a(v(\omega))$. This can be phrased categorically as whenever the following commutes:

$$\begin{tikzcd}
(\omega) \arrow[r, "\iota_\omega"] \arrow[d, "v"'] & (\alpha \ll \omega) \arrow[d, "t"] \\
\mathit{e} \arrow[r, "a"'] & \Theta
\end{tikzcd}$$

then the following unique commuting assignment is induced:

$$\begin{tikzcd}
\mathit{e} \arrow[r, "\iota_e"] \arrow[dr, "a"'] & \mathit{e}' \arrow[d, dashed, "a_t"] & v' \arrow[l, "v'"'] \arrow[dl, "t"] \\
 & \Theta &
\end{tikzcd}$$

where ι_e includes e into the extended environment e', and v' assigns ω to $v(\omega)$ and includes α into the extended environment e'.

The commuting square

$$\begin{tikzcd}
(\omega) \arrow[r, "\iota_\omega"] \arrow[d, "v"'] & (\alpha \ll \omega) \arrow[d, "v'"] \\
\mathit{e} \arrow[r, "\iota_e"'] & \mathit{e}'
\end{tikzcd}$$

with this property of inducing assignments is known as a pushout square. So the process of introducing variables takes a template $(\alpha \ll \omega)$ with parameter ω and an instantion v of ω in the bound e and constructs the pushout $\mathit{e}[\alpha \ll v(\omega)]$.

Readers unfamiliar with this concept should consider the following example. Suppose we have $\exists\beta \ll C.\exists\alpha \ll \beta.\mathrm{Rec}(\alpha, \beta)$. Intuitively, this should open into

$\exists\alpha \ll \beta \ll C.\mathrm{Rec}(\alpha,\beta)$. To prove this, convince yourself that the following is a pushout square in $\Theta \downarrow \mathbf{Prost}$ (where the inclusions have the obvious definitions):

$$\begin{tikzcd}
\Theta[\omega] \arrow[r, hook, "\iota_\omega"] \arrow[d, "\omega \mapsto \beta"'] & \Theta[\alpha \ll \omega] \arrow[d, "\omega \mapsto \beta"] \\
\Theta[\beta \ll C] \arrow[r, hook, "\iota"'] & \Theta[\alpha \ll \beta \ll C]
\end{tikzcd}$$

Adding Constraints Often analyses exploit comparisons made in branch conditions to infer properties which must hold in the branch. One particularly important comparison for object-oriented languages is $\mathrm{Runtime}(\alpha) = \mathrm{Runtime}(\beta)$. If this holds, then $\alpha = \beta$, since runtime tags are unique to each class. To encode this in existential types, the existential bound should be adjusted so that α and β become the same variable having the combined properties that α and β had. What should hold is that any assignment a of the old existential bound for which $a(\alpha) = a(\beta)$ should also be an assignment of the new existential bound. This property can be encoded using the categorical concept called coequalizers. Rather than introduce a new concept, there is a more general solution which can also encode new information like $\alpha \ll \beta$. This more general solution is, again, pushouts.

Using intuition much like with introducing new variables, the following forms a pushout in $\Theta \downarrow \mathbf{Prost}$ (where the inclusions have the obvious definitions):

$$\begin{tikzcd}
\Theta[\alpha,\beta] \arrow[r, "{\alpha,\beta \mapsto \gamma}"] \arrow[d, hook] & \Theta[\gamma] \arrow[d, hook] \\
\Theta[A \ll \alpha][\beta \ll C] \arrow[r, "{\alpha,\beta \mapsto \gamma}"'] & \Theta[A \ll \gamma \ll C]
\end{tikzcd}$$

The top map forces α and β to merge in value in the new environment. If the top map were $\Theta[\alpha,\beta] \hookrightarrow \Theta[\alpha \ll \beta]$, then it would force α to extend β in the new environment.

Coequalizers, pushouts, and other devices are very common in naturally constructed categories like $\Theta \downarrow \mathbf{Prost}$.

8 Flexibility of the Framework

This framework is able to handle type systems much more complex than iTal. After requiring generalized subtyping, the real challenge and power of existential types lie in the bounds. Here we investigate how various features can be expressed in the bounds. All the constructions provided below use natural processes which guarantee the presence of a useful factorization structure.

Interfaces Interfaces can be handled extremely simply. Unfortunately, computability of joins presents some major problems because of multiple inheritance. Depending on circumstances, more complex solutions may be necessary.

First, the simple solution. In iTal, every environment is a partially ordered set. For interfaces, we need to add two more unary relations: IsClass and IsInterface. These relations have their obvious interpretations. What may seem counterintuitive, though, is that not every item has either property IsClass or IsInterface. This is due to the fact that a variable α may need to be able to represent both classes and interfaces. If $\text{IsClass}(\alpha)$ holds, then α can only represent classes. With this, we can add additional properties like $\forall \alpha \ll \beta.\ \text{IsClass}(\beta) \Rightarrow \text{IsClass}(\alpha)$. So long as nothing can implement an infinite set of interfaces, then joins are computable because iTal only allows upper-bounds for variables (provided we allow variables to have multiple upper-bounds).

If, however, variables could have lower-bounds as well, computability may not be so simple. If no interface is extended by an infinite set of interfaces or classes, then joins are still computable. If not, then perhaps there is a finite set of interfaces/classes which represents the infinite set. If this finite set is computable, then joins are computable. In fact, if subclassing is co-well-founded, then weakening of bounds is well-founded. If there is no finite representing set, we can use pseudo-interfaces representing sets of interfaces with a common subclass in order to make subclassing have necessary joins. Then, again, joins are computable. In fact, if there is a global bound on the number of interfaces implemented by any class, then weakening of bounds is still well-founded.

Generics and Arrays Interestingly, for class systems where arrays are classes themselves, the solution for handling arrays is about the same as the solution for handling generics. In fact, computing this solution is generally simpler than for interfaces, and it has strong connections to unification.

The fundamental concept is partial mono-algebras. A mono-algebra is a set with some operators and the operators are injective. For example, say there is an operator Array which maps a class to its array class. Distinct classes have distinct array classes, so Array is injective. A partial mono-algebra is like a mono-algebra except its operators may not be defined for all arguments. For example, suppose the SortedList generic is only defined on classes implementing Comparable; then, when considered as an operator on the set of classes, it is only a partial operator.

The fact that the operators are injective is extremely important when considering factorizations. Suppose we want to join $\text{Map}(\text{SortedList}(B), \text{Array}(C))$ and $\text{Map}(\text{SortedList}(X), \text{SortedList}(Y))$. Because these operators are injective, the join will be $\exists \alpha, \beta.\text{Map}(\text{SortedList}(\alpha), \beta)$, which is what we might expect. If SortedList were not injective, then the join would have to be $\exists \alpha, \beta.\text{Map}(\alpha, \beta)$ for reasons we will not go into here. Interestingly, the algorithm for computing this join uses a process much like unification.

Mono-algebras also allow for more powerful rules in your bounds. One such rule which is particularly important for handling covariant arrays is $\forall \alpha, \beta.\ \alpha \ll$

$\text{Array}(\beta) \Rightarrow (\exists\gamma.\ \alpha = \text{Array}(\gamma) \wedge \gamma \ll \beta)$. If Array were not injective and this rule were enforced, then the factorization structure would be lost so joins may not always exist. For the interested reader, these rules are expressed categorically using implications [1], and appropriate implications interact well with factorization structures.

Array Bounds Checking Array bounds checking can be expressed by existential types using the theory of ordered rings. This theory can be rather complex, so often the simpler theory of ordered sets acted on by the integers is used instead. Suppose there is one integer variable n. Then items in the corresponding set look like $a * n + b$, where a and b are integer constants, a very simple form of polynomials. Since this set is ordered, bounds may look something like $0 \leq n \leq \ell - 1$ where ℓ is an integer variable representing the length of some array. Under this bound, it is safe to access the array with length ℓ at index n. A more complex scenario would be the bound $0 \leq 2 * n \leq \ell - 2$, which makes it safe to index by n, $n + 1$, $2 * n$, and $2 * n + 1$, which may be useful when an array has some alternating structure.

The variable ℓ needs to be introduced at some point. Rather than using the type $\text{Int}[]$ to represent arrays of integers, one could use the type $\text{Int}[\ell]$ where ℓ is some integer. Any field, parameter, or return type that originally was $\text{Int}[]$ would be changed to the type $\exists\ell \geq 0.\text{Int}[\ell]$. Then the usual opening process would pull ℓ into the outer existential bound whenever possible in the same way $\exists\alpha \ll C.\text{Ins}(\alpha)$ is handled. The original type of operation *length* : $\text{Int}[] \rightarrow \text{Int}$ would be changed to $\text{Int}[\ell] \rightarrow \ell$. Then on the true branch of the comparison `i < length(a)`, where `i` has type n and `a` has type $\text{Int}[\ell]$, the constraint $n \leq \ell - 1$ is added to the bound using techniques described in the technical report [19].

Fortunately, joins are computable. Unfortunately, weakening of bounds is not well-founded as demonstrated by the sequence $(n \leq 0, n \leq 1, n \leq 2, \ldots)$. In this case, there are domain-specific techniques for making type inference decidable.

9 Shortcomings and Future Research

Although extremely useful, this framework has a few shortcomings. Some are unavoidable while others we believe can be improved upon. Below are the major concerns we have and how we hope to solve them in future research.

Null Pointers A common subtype rule is `null` $\leq \text{Ins}(\alpha)$ for any α. Unfortunately, this does not fit into our framework because the value for α does not come from within the type `null`. This is for good reason though, since adding such a rule makes joins impossible. Consider the following two types: $\tau_\alpha = \exists\alpha.\text{Rec}(\texttt{null}, \alpha, \texttt{null})$ and $\tau_\beta = \exists\beta, \beta'.\text{Rec}(\beta, \texttt{null}, \beta')$. The join would have the structure $\tau_\star = \text{Rec}(\star, \star, \star)$. Now consider the following two supertypes of τ_α and τ_β: $\tau_\gamma = \exists\gamma, \gamma'.\text{Rec}(\gamma, \gamma, \gamma')$ and $\tau_\delta = \exists\delta, \delta'.\text{Rec}(\delta, \delta', \delta')$. The only common subtype of τ_γ and τ_δ with structure $\tau_\star$ is $\tau_\rho = \exists\rho.\text{Rec}(\rho, \rho, \rho)$. But τ_ρ

is not a supertype of τ_β, so τ_α and τ_β cannot have a join. Fortunately, we have always managed to find alternate solutions to rules of this form.

Covariance Another common rule expresses covariant types. Suppose $\mathrm{SubIns}(C)$ represents instances of subclasses of C. Then $\mathrm{SubIns}(\alpha)$ is a subtype of $\mathrm{SubIns}(\beta)$ whenever α is a subclass of β. Again, this rule does not fit into our framework, and again it is for good reason. Using this rule, there are only joins when the inheritance hierarchy has joins, regardless of computability. To model such requirements takes 2-categories. 2-categories are a rather advanced theory, so we have left this to future research. Fortunately, the power of existential types tends to make such subtypings either unnecessary or circumnavigatable.

Computability Although this framework gives a construction of joins it may not always be the case that this construction is computable. This makes sense since typically the construction of joins relies on properties and operations on Θ, so if these operations are not computable then the existential join will not be computable. There are a few potential solutions. One expresses these concepts in terms of the category of computable sets and functions rather than **Set**. Another uses the concept of finitely representable sets and coequalizers. We have left this to future research since this framework provides the construction and proof of the join, which is often the more difficult component to identify. Each instance of the framework, then, can use the full power of the properties specific to that instance when determining how to compute this explicit construction.

High-Level Languages So far all applications we have of this framework have been to low-level languages. However, there is no requirement of the framework that restics it to such domains. There just tends to be more need for existential types in low-level contexts. This framework makes the standard “pack” and “unpack” operations implicit. Removing the need for these cumbersome operations may make existential types more accessible to programmers. In particular, existentially quantifying generics allows for improved type safety and greater modularity of implementation. We hope to combine the above techniques to provide a high-level object-oriented language with existentially quantified generics.

10 Conclusions

We have introduced a category-theoretic framework for existential types. The framework uses factorization structures to generate joins for existential types, so that a dataflow analysis can use the joins to infer types in imperative programs. This inference is powerful enough to infer types in a small object-oriented typed assembly language, iTal, even though each iTal method’s assembly language code omits all type annotations, except for the method’s type signature.

The framework applies to languages beyond iTal. Any type system can exploit the framework if it satisfies these requirements: the types’ underlying structure supports joins, the values in the 'locations' in the supertype come from the values in the subtype, and the bounds and substitutions have factorization structures. Many type systems satisfy these requirements, so that the framework describes many features beyond iTal's features: existential quantification with multiple upper and lower bounds, interfaces, arrays, generics, and array bounds checks. In addition to low-level languages, we hope to explore the impact of this framework on high-level languages.

References

1. J. Adámek, H. Herrlich, and G. Strecker. *Abstract and concrete categories.* Wiley-Interscience, New York, NY, USA, 1990.
2. B. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck. Type-based verification of assembly language for compiler debugging. In *ACM international workshop on Types in languages design and implementation*, pages 91–102, 2005.
3. J. Chen, C. Hawblitzel, F. Perry, M. Emmi, J. Condit, D. Coetzee, and P. Pratikaki. Type-preserving compilation for large-scale optimizing object-oriented compilers. In *ACM conference on Programming language design and implementation*, pages 183–192, 2008.
4. J. Chen and D. Tarditi. A simple typed intermediate language for object-oriented languages. In *ACM Symp. on Principles of Programming Languages*, pages 38–49, 2005.
5. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
6. A. Goldberg. A specification of java loading and bytecode verification. In *ACM conference on Computer and communications security*, pages 49–58, 1998.
7. M. P. Jones. First-class polymorphism with type inference. In *ACM symposium on Principles of programming languages*, pages 483–496, 1997.
8. D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System F. In *ACM International Conference on Functional Programming*, pages 27–38, 2003.
9. C. League, Z. Shao, and V. Trifonov. Type-preserving compilation of Featherweight Java. *ACM Trans. on Programming Languages and Systems*, 24(2):112–152, 2002.
10. D. Leijen. HMF: Simple type inference for first-class polymorphism. In *ACM symp. of the International Conference on Functional Programming*, September 2008.
11. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Sun Microsystems, 2nd edition, 1999.
12. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
13. G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *ACM Workshop on Compiler Support for System Software*, pages 25–35, 1999.
14. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 13(5):957–959, 2003.
15. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *ACM Trans. on Programming Languages and Systems*, volume 21, pages 527–568. ACM Press, 1999.

16. G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Symposium on Operating Systems Design and Implementation*, pages 229–243, 1996.
17. M. Odersky and P. Wadler. Pizza into java: translating theory into practice. In *ACM symposium on Principles of programming languages*, pages 146–159, 1997.
18. B. C. Pierce and D. N. Turner. Local type inference. In *ACM symposium on Principles of programming languages*, pages 252–265, 1998.
19. R. Tate, J. Chen, and C. Hawblitzel. A framework for type inference with existential quantification. Technical report, University of California, San Diego and Microsoft Research, Redmond, http://cs.ucsd.edu/ rtate/existentials/, 2008.
20. J. B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98:111–156, 1999.

A Source Language

The source language is a simple object-oriented language that supports object creation, field fetch, and virtual method invocation. The semantics of the source language is trivial and thus omitted here.

$$
\begin{array}{ll}
\tau & ::= C \mid \mathrm{Int} \\
e & ::= n \mid x \mid e.f \\
s & ::= e_1.f := e_2 \mid x := e \mid e.m(e_1, \ldots, e_n) \mid s_1; s_2 \mid \text{if } (e)\ s_1 \text{ else } s_2 \\
\text{mdecl} & ::= m(x_1 : \tau_1, \ldots, x_n : \tau_n)\{s\} \\
\text{cdecl} & ::= C : B\{l_1 : \tau_1, \ldots, l_n : \tau_n, \text{mdecl}_1, \ldots, \text{mdecl}_m\} \\
H & ::= \ell \mapsto \mathit{ins}(C)\{\{v_1, \ldots, v_n\}\} \\
P & ::= (\text{cdecl}_1, \ldots, \text{cdecl}_n; H; s)
\end{array}
$$

B Translation from Source to iTal

Assume the target language has an infinite set of registers. For each source language variable x, there is a unique correponding register r_x. Different source language variables will not map to the same register in the target language.

Translation of Types. $|\tau|$ means translation of types. A source class name C is translated to an existential type in iTal $\exists\alpha{\ll}C.\ \mathrm{Ins}(\alpha)$.

$$
\begin{array}{ll}
|\mathrm{Int}| & = \mathrm{Int} \\
|C| & = \exists\alpha{\ll}C.\ \mathrm{Ins}(\alpha)
\end{array}
$$

Translation of Programs. The translation of a source program translates each class declaration in the program, builds on the heap functions for the methods and vtables for the classes, and translates the “main” statement. We assume there exist a frame sequence $\overrightarrow{F}$ where “main” returns, which are often provided by the operating system.

$$\frac{\begin{array}{c}|\mathrm{cdecl}_i| = (\mathrm{cdecl}'_i, H_i)\ \forall 1 \le i \le n \quad H'_0 = H_0; H_1, \ldots, H_n \\ |(s, \bullet, \ell_{main}, \bullet)| = (\overrightarrow{b}, \ell, \overrightarrow{\iota s}) \quad \overrightarrow{b}' = \overrightarrow{b}; (\ell : \overrightarrow{\iota s}; \mathrm{ret}) \\ H = H'_0, \ell_{main} \mapsto \exists \bullet.\bullet\{\overrightarrow{b}'\}\end{array}}{|(\mathrm{cdecl}_1, \ldots, \mathrm{cdecl}_n; H_0; s)| = (H, (\bullet; \mathrm{jmp}\ \ell_{main}) :: \overrightarrow{F})}$$

Translation of Class Declarations. The translation of a source class declaration returns a class declaration in iTal and a heap that contains functions for the methods and the vtable for the class. The source class declaration includes the method body for each method, whereas the iTal class declaration includes only method signatures. The method bodies are translated to functions on the heap.

$$\frac{|\mathrm{mdecl}_i| = (\ell_i, \Phi_i, H_i) \forall 1 \le i \le m \quad H = H_1, \ldots, H_m, \ell_C \mapsto \mathit{vtable}(C)\{\ell_1, \ldots, \ell_m\}}{|C : B\{\{l_i : \tau_i\}_{i=1}^n, \mathrm{mdecl}_1, \ldots, \mathrm{mdecl}_m\}| = C : B\{\{|\tau_i|\}_{i=1}^n, \Phi_1, \ldots, \Phi_m\}, H)}$$

Translation of Methods. The translation of a source method returns a label that points to the beginning of the corresponding function on the heap, the method signature, and a heap that maps the label to the function.

$$\frac{|(s, \bullet, \ell_m, \bullet)| = (\overrightarrow{b}, \ell, \overrightarrow{\iota s}) \quad H = \ell_m \mapsto \{x_i : |\tau_i|\}_{i=1}^n\{\overrightarrow{b}, \ell : (\overrightarrow{\iota s}; \mathrm{ret})\}}{|m(\{x_i : \tau_i\}_{i=1}^n)\{s\}| = (\ell_m, \{x_i : |\tau_i|\}_{i=1}^n, H)}$$

Translation of Statements. The translation of a statement takes four inputs: the statement, the collection of basic blocks so far, the label of the current basic block, and the instruction sequence for the current block. It produces three outputs: the new collection of basic blocks, the label for the new current block, and the instruction sequence for the new current block. The translation of most statements simply adds new instructions to the current basic block. The translation of the "if" statement finishes the current block with a branch instruction, creates two blocks for the true branch and the false branch respectively, and creates a block for the merge point, which becomes the new current block.

$$\frac{|e_1|_{r_1} = \overrightarrow{\iota s}_1 \quad |e_2|_{r_2} = \overrightarrow{\iota s}_2}{|(e_1.f := e_2,\ \overrightarrow{b}, \ell, \overrightarrow{\iota s})| = (\overrightarrow{b}, \ell, (\overrightarrow{\iota s}_1; \overrightarrow{\iota s}_2; \text{mov } [r_1 + n_f], r_2))}$$

$$\frac{|e|_{r_x} = \overrightarrow{\iota s}}{|(x := e,\ \overrightarrow{b}, \ell, \overrightarrow{\iota s})| = (\overrightarrow{b}, \ell, \overrightarrow{\iota s})}$$

$$\frac{|e|_r = \overrightarrow{\iota s} \quad |e_i|_{r_i} = \overrightarrow{\iota s}_i\ \forall 1 \leq i \leq n \quad \overrightarrow{\iota s}' = \overrightarrow{\iota s}; \overrightarrow{\iota s}_1; \ldots; \overrightarrow{\iota s}_n; \text{call } r}{|(e.m(e_1, \ldots, e_n),\ \overrightarrow{b}, \ell, \overrightarrow{\iota s})| = (\overrightarrow{b}, \ell, \overrightarrow{\iota s}')}$$

$$\frac{|(s_1,\ \overrightarrow{b}, \ell, \overrightarrow{\iota s})| = (\overrightarrow{b}_1, \ell_1, \overrightarrow{\iota s}_1) \quad |(s_2,\ \overrightarrow{b}_1, \ell_1, \overrightarrow{\iota s}_1)| = (\overrightarrow{b}_2, \ell_2, \overrightarrow{\iota s}_2}{|(s_1; s_2,\ \overrightarrow{b}, \ell, \overrightarrow{\iota s})| = (\overrightarrow{b}_2, \ell_2, \overrightarrow{\iota s}_2)}$$

$$\frac{\begin{array}{c} |e|_r = \overrightarrow{\iota s}' \quad \overrightarrow{b}' = \overrightarrow{b}; (\ell : \overrightarrow{\iota s}; \overrightarrow{\iota s}'; \text{jz } r, \ell_t, \ell_f) \\ |(s_1,\ \overrightarrow{b}', \ell_t, \bullet)| = (\overrightarrow{b}_t, \ell_t', \overrightarrow{\iota s}_t) \quad \overrightarrow{b}'_t = \overrightarrow{b}_t; (\ell_t : \overrightarrow{\iota s}_t; \text{jmp } \ell_m) \\ |(s_2,\ \overrightarrow{b}'_t, \ell_f, \bullet)| = (\overrightarrow{b}_f, \ell_f', \overrightarrow{\iota s}_f) \quad \overrightarrow{b}'_f = \overrightarrow{b}_f; (\ell_f' : \overrightarrow{\iota s}_f; \text{jmp } \ell_m) \end{array}}{|(\text{if } (e)\ s_1 \text{ else } s_2,\ \overrightarrow{b}, \ell, \overrightarrow{\iota s})| = (\overrightarrow{b}'_f, \ell_m, \bullet)}$$

Translation of Expressions. The translation of an expression takes an expression and a register and returns a sequence of instructions that move the result of the expression to the register.

$$\begin{array}{ll} |n|_r & = \text{mov } r, n \\ |x|_r & = \text{mov } r, r_x \\ |e.f|_r & = |e|_{r_e}; \text{mov } r, [r_e + n_f] \end{array}$$

C Transitivity of State Type Subtyping

Theorem 1. *If $\Theta \vdash \Sigma_1 \leq \Sigma_2$ and $\Theta \vdash \Sigma_2 \leq \Sigma_3$, then $\Theta \vdash \Sigma_1 \leq \Sigma_3$.*

Proof. Suppose $\Sigma_1 = \exists \Delta_1.\Gamma_1$, $\Sigma_2 = \exists \Delta_2.\Gamma_2$, and $\Sigma_3 = \exists \Delta_3.\Gamma_3$,

By st-sub, there exists $\theta_1 : dom(\Delta_2) \to dom(\Delta_1) \cup dom(\Theta)$ such that $\Gamma_1(r) = \Gamma_2(r)[\theta_1]$ for all $r \in dom(\Gamma_2)$. And $\Theta; \Delta_1 \vdash \theta_1(\alpha) \ll C$ for all $\alpha \ll C \in \Delta_2$.

Also by st-sub, there exists $\theta_2 : dom(\Delta_3) \to dom(\Delta_2) \cup dom(\Theta)$ such that $\Gamma_2(r) = \Gamma_3(r)[\theta_2]$ for all $r \in dom(\Gamma_3)$. And $\Theta; \Delta_2 \vdash \theta_2(\alpha) \ll C$ for all $\alpha \ll C \in \Delta_3$.

Define $\theta : dom(\Delta_3) \to dom(\Delta_1) \cup dom(\Theta)$ as

$$\begin{array}{ll} \theta(\alpha) = C & \theta_2(\alpha) = C \\ \theta(\alpha) = \theta_1(\theta_2(\alpha)) & \text{otherwise} \end{array}$$

For all $r \in dom(\Gamma_3)$, $\Gamma_2(r) = \Gamma_3(r)[\theta_2]$. Therefore, $\Gamma_2(r)[\theta_1] = \Gamma_3[\theta_2][\theta_1]$. Then $\Gamma_1(r) = \Gamma_2(r)[\theta_1] = \Gamma_3[\theta]$.

For all $\alpha \ll C \in \Delta_3, \Theta; \Delta_2 \vdash \theta_2(\alpha) \ll C$. By Lemma 1, $\Theta; \Delta_1 \vdash (\theta_2(\alpha))[\theta_1] \ll C$, that is, $\Theta; \Delta_1 \vdash \theta(\alpha) \ll C$.

By st-sub, $\Theta \vdash \Sigma_1 \leq \Sigma_3$.

Lemma 1. *If $\Theta; \Delta_2 \vdash \omega_1 \ll \omega_2$, and $\theta_1 : dom(\Delta_2) \to dom(\Delta_1) \cup \Theta$ such that $\Theta; \Delta_1 \vdash \theta_1(\alpha) \ll C$ for all $\alpha \ll C \in \Delta_2$, then $\Theta; \Delta_1 \vdash \omega_1[\theta_1] \ll \omega_2[\theta_1]$.*

Proof. By induction on subclassing rules.

D Soundness of the Target Language

Theorem 2. *If $\Theta; \Psi \vdash P$ and $P \to P'$, then $\Theta; \Psi \vdash P'$.*

Proof. Let $P = (H; (R; \iota s) :: \overrightarrow{F})$.

By the program typing rule, $\vdash \Theta$, and $\Theta \vdash H : \Psi$, and $\Theta; \Psi \vdash R : \Gamma$, and $\Theta; \Psi; \bullet; \Gamma \vdash \iota s$, and $F_i = (R_i, \iota s_i)$, and $\Theta; \Psi \vdash R_i : \Gamma_i$, and $\Theta; \Psi; \bullet; \Gamma_i \vdash \iota s_i$, $\forall 1 \leq i \leq n$.

case jmp: $\iota s =$ jmp ℓ, and $H(\ell) = \iota s_\ell$, and $P' = (H; (R; \iota s_ell) :: \overrightarrow{F})$.

By rule t-jmp and $\Theta; \Psi; \bullet; \Gamma \vdash$ jmp ℓ, $\Theta \vdash \exists \bullet.\Gamma \leq \Psi_c(\ell)$.

By the heap value typing rule, $\Theta; \Psi; \Delta_\ell; \Gamma_\ell \vdash \iota s_\ell$ if $\Psi_c(\ell) = \exists \Delta_\ell.\Gamma_\ell$.

By Lemma 2, $\Theta; \Psi; \bullet; \Gamma \vdash \iota s_\ell$.

By t-prog, $\Theta; \Psi \vdash P'$.

case jeT: $\iota s =$ jz $0, \ell_t, \ell_f$, and $H(\ell_t) = \iota s_t$, and $P' = (H; (R; \iota s_t) :: \overrightarrow{F})$.

By rule t-je, $\Theta \vdash \exists \bullet.\Gamma \leq \Psi_c(\ell_t)$.

By well-formedness of heap, $\Theta; \Psi; \Delta_t; \Gamma_t \vdash H\ell_t$ if $\Psi_c(\ell_t) = \exists \Delta_t.\Gamma_t$.

By Lemma 2, $\Theta; \Psi; \bullet; \Gamma \vdash \iota s_t$.

By t-prog, $\Theta; \Psi \vdash P'$.

case jeF: similar to case jeT.

case bop: $\iota s =$ (bop r, n), $\iota s'$, and $P' = (H; (R'; \iota s') :: \overrightarrow{F})$, and $R' = R[r \mapsto R(r)$ bop $n]$.

By rule t-bop, $\Theta; \Psi; \bullet; \Gamma \vdash r :$ Int, and $\Theta; \Psi; \bullet; \Gamma \vdash \iota s'$.

By operand typing rule, $\Gamma(r) =$ Int.

By register bank typing rule, $\Theta; \Psi \vdash R' : \Gamma$.

By t-prog, $\Theta; \Psi \vdash P'$.

case movR: $\iota s =$ (mov r, v), $\iota s'$, and $P' = (H; (R[r \mapsto v]; \iota s') :: \overrightarrow{F})$.

By rule t-movR, $\Theta; \Psi; \bullet; \Gamma \vdash v : \tau$, and $\bullet; \Gamma \vdash \{r \leftarrow \tau\}\Delta'; \Gamma'$, and $\Theta; \Psi; \Delta'; \Gamma' \vdash \iota s'$.

By the register bank typing rule, $\Theta; H \vdash R[r \mapsto v] : \Gamma[r : \tau]$.

By inpsection of value typing rules, $\tau \neq \exists \alpha {\ll} C.$ Ins(α). By asgn-e, $\Delta' = \Delta$ and $\Gamma' = \Gamma[r : \tau]$.

By t-prog, $\Theta; \Psi \vdash P'$.

case movM: $\iota s =$ (mov $[\ell + m], v$), $\iota s'$, and $P' = (H'; (R; \iota s') :: \overrightarrow{F})$, and $H(\ell) = ins(C)\{v_1, \ldots, v_n\}$, and $1 \leq m \leq n$, and $H' = H[\ell \mapsto ins(C)\{v_1, \ldots, v_{m-1}, v, v_{m+1}, \ldots, v_n\}]$.

By t-movM, $\Theta;\Psi;\bullet;\Gamma \vdash \ell : \text{Ins}(\omega)$, and $\Theta;\Psi;\bullet;\Gamma \vdash [\ell + m] : \tau_m$, and $\Theta;\Psi;\bullet;\Gamma \vdash v : \tau$, and $\Theta;\bullet \vdash \tau_m \hookleftarrow \tau$, and $\Theta;\Psi;\bullet;\Gamma \vdash \iota s'$.

By the operand typing rule and $\Theta;\Psi;\bullet;\Gamma \vdash \ell : \text{Ins}(\omega)$, $\Theta;\bullet \vdash \omega \ll B$, and $\Theta(B) = B : \_\tau_1, \ldots, \tau_n$, and $1 \leq m \leq n$.

By heap value typing rule, $\omega = C$, and $\Theta;\Psi \vdash H'(\ell) : \Psi(\ell)$.

By t-prog, $\Theta;\Psi \vdash P'$.

case call: $\iota s = \text{call } \ell; \iota s'$, and $P' = (H; (R'; \iota s_1) :: (R; \iota s') :: \overrightarrow{F})$, and $H(\ell) = \Phi\ \{\ell_1 : \iota s_1, \ldots, \ell_n : \iota s_n\}$, and $R'(r) = R(r)$, $\forall r \in \mathit{dom}(\Phi)$.

By t-call, $\Theta;\Psi;\bullet;\Gamma \vdash \ell : \text{Code}(\{r_i : \tau_i\}_{i=1}^n)$, and $\Theta;\Psi;\bullet;\Gamma \vdash \iota s'$, and $\Theta;\bullet \vdash \tau_i \hookleftarrow \Gamma(r_i)$, $\forall 1 \leq i \leq n$.

By vt-lbl, t-heap, and hv-func, $\Phi = \{r_i : \tau_i\}_{i=1}^n$, and $\Psi_c(\ell_1) = \mathit{unpack}(\Phi)$, and $\Psi_c(\ell_1) = \exists \Delta_1.\Gamma_1$, and $\Theta;\Psi;\Delta_1;\Gamma_1 \vdash \iota s_1$.

By the definition of R', $\Theta;\Psi \vdash R' : \Gamma'$ where $\Gamma'(r_i) = \Gamma(r_i) \forall 1 \leq i \leq n$.

By Lemma 3 $\Theta \vdash \exists \bullet.\Gamma' \leq \Psi_c(\ell_1)$.

By Lemma 2, $\Theta;\Psi;\bullet;\Gamma' \vdash \iota s_1$. By t-prog, $\Theta;\Psi \vdash P'$.

case ret: $\mathit{instrs} = \text{ret}$, and $P' = (H; \overrightarrow{F})$.

By t-prog, $\Theta;\Psi \vdash P'$.

Lemma 2. *If $\Theta;\Psi;\Delta';\Gamma' \vdash \iota s$ and $\Theta \vdash \exists \bullet.\Gamma \leq \exists \Delta'.\Gamma'$, then $\Theta;\Psi;\bullet;\Gamma \vdash \iota s$.*

Proof. By induction on the instruction typing rules.

case t-ret: trivial.

case t-jmp: $\iota s = \text{jmp } \ell$, and $\Theta \vdash \exists \Delta'.\Gamma' \leq \Psi_c(\ell)$.

By Theorem 1, $\Theta \vdash \exists \bullet.\Gamma \leq \Psi_c(\ell)$.

By t-jmp, $\Theta;\Psi;\bullet;\Gamma \vdash \iota s$.

case t-je: $\iota s = \text{jz } o, \ell_t, \ell_f$, and $\Theta;\Psi;\mathit{kenv}';\Gamma' \vdash o : \text{Int}$, and $\Theta \vdash \exists \Delta'.\Gamma' \leq \Psi_c(\ell_t)$, and $\Theta \vdash \exists \Delta'.\Gamma' \leq \Psi_c(\ell_f)$.

By Lemma 4, $\Theta;\Psi;\bullet;\Gamma \vdash o : \text{Int}$.

By Theorem 1, $\Theta \vdash \exists \bullet.\Gamma \leq \Psi_c(\ell_t)$ and $\Theta \vdash \exists \bullet.\Gamma \leq \Psi_c(\ell_f)$.

By t-je, $\Theta;\Psi;\bullet;\Gamma \vdash \iota s$.

case t-call: $\iota s = \text{call } o; \iota s'$, and $\Theta;\Psi;\Delta';\Gamma' \vdash o : \text{Code}(\{r_i : \tau_i\}_{i=1}^n)$, and $\Theta;\Psi;\Delta;\Gamma' \vdash \iota s'$, and $\mathit{cdecls};\Delta \vdash \tau_i \hookleftarrow \Gamma'(r_i), \forall 1 \leq i \leq n$.

By Lemma 4, $\Theta;\Psi;\bullet;\Gamma \vdash o : \text{Code}(\{r_i : \tau_i\}_{i=1}^n)[\theta]$. $\text{Code}(\{r_i : \tau_i\}_{i=1}^n)[\theta] = \text{Code}(\{r_i : \tau_i[\theta]\}_{i=1}^n)$.

By the induction hypothesis, $\Theta;\Psi;\bullet;\Gamma \vdash \iota s'$.

By Lemma 5, $\Theta;\Psi;\bullet \vdash \tau_i[\theta] \hookleftarrow \Gamma'(r_i)[\theta]\ \forall 1 \leq i \leq n$. By st-sub, $\Gamma'(r_i)[\theta] = \Gamma(r_i)$.

By t-call, $\Theta;\Psi;\bullet;\Gamma \vdash \iota s$.

case t-bop: $\iota s = \text{bop } r, o; \iota s'$, and $\Theta;\Psi;\Delta';\Gamma' \vdash r : \text{Int}$, and $\Theta;\Psi;\Delta';\Gamma' \vdash o : \text{Int}$, and $\Theta;\Psi;\Delta';\Gamma' \vdash \iota s'$.

By Lemma 4, $\Theta;\Psi;\bullet;\Gamma \vdash r : \text{Int}$, and $\Theta;\Psi;\bullet;\Gamma \vdash o : \text{Int}$.

By the induction hypothesis, $\Theta;\Psi;\bullet;\Gamma \vdash \iota s'$.

By t-bop, $\Theta;\Psi;\bullet;\Gamma \vdash \iota s$.

case t-movR: $\iota s = \text{mov } r, o; \iota s'$, and $\Theta;\Psi;\Delta';\Gamma' \vdash o : \tau$, and $\Delta';\Gamma' \vdash \{r \leftarrow \tau\}(\Delta'';\Gamma'')$, and $\Theta;\Psi;\Delta'';\Gamma'' \vdash \iota s'$.

By Lemma 4, $\Theta;\Psi;\bullet;\Gamma \vdash o : \tau[\theta]$.

By Lemma 6, suppose $\bullet; \Gamma \vdash \{r \leftarrow \tau[\theta]\}(\Delta_1; \Gamma_1)$, then $\Theta \vdash \exists \Delta_1.\Gamma_1 \leq \exists \Delta''.\Gamma''$.

By the induction hypothesis, $\Theta; \Psi; \Delta_1; \Gamma_1 \vdash \iota s'$.

By t-movR, $\Theta; \Psi; \bullet; \Gamma \vdash \iota s$.

case t-movM: $\iota s = \text{mov } [r_1 + m], r_2; \iota s'$, and $\Theta; \Psi; \Delta'; \Gamma' \vdash r_1 : \text{Ins}(\omega)$, and $\Theta; \Psi; \Delta'; \Gamma' \vdash [r_1 + m] : \tau_m$, and $\Theta; \Psi; \Delta'; \Gamma' \vdash r_2 : \tau$, and $\Theta; \Delta' \vdash \tau_m \hookleftarrow \tau$, and $\Theta; \Psi; \Delta'; \Gamma' \vdash \iota s'$.

By Lemma 4, $\Theta; \Psi; \bullet; \Gamma \vdash r_1 : \text{Ins}(\omega)[\theta]$, and $\Theta; \Psi; \bullet; \Gamma \vdash [r_1 + m] : \tau_m[\theta]$, and $\Theta; \Psi; \bullet; \Gamma \vdash r_2 : \tau[\theta]$.

By Lemma 5, $\Theta; \bullet \vdash \tau_m[\theta] \hookleftarrow \tau[\theta]$.

By the induction hypothesis, $\Theta; \Psi; \bullet; \Gamma \vdash \iota s'$.

By t-movM, $\Theta; \Psi; \bullet; \Gamma \vdash \iota s$.

Lemma 3. *If $\Phi = \{r_i : \tau_i\}_{i=1}^n$, and $\Gamma' = r_1 : \tau'_1, \ldots, r_n : \tau'_n$, and $\Theta; \bullet \vdash \tau_i \hookleftarrow \tau'_i$, then $\Theta \vdash \exists \bullet.\Gamma' \leq unpack(\Phi)$.*

Proof. We build a substitution $\theta : dom(unpack(\Phi)) \rightarrow dom(\Theta)$ as follows: For all $1 \leq i \leq n$, if $\tau_i = \exists \alpha \ll C.\ \text{Ins}(\alpha)$ and $\tau'_i = \text{Ins}(\omega)$ and $\Theta; \bullet \vdash \omega \ll C$ (rule at-ref), then ω should be a class D. Suppose $unpack(\Phi) = \exists \Delta.\Gamma$. By the definition of $unpack$, Δ has a fresh type variable $\beta \ll C$ and $\Gamma(r_i) = \text{Ins}(()\beta)$. Let $\theta(\beta) = D$. Then $\Gamma(r_i)[\theta] = \Gamma'(r_i)$ and $\Theta; \bullet \vdash \beta[\theta] \ll C$.

By st-sub, $\Theta \vdash \exists \bullet.\Gamma' \leq unpack(\Phi)$.

Lemma 4. *If $\Theta \vdash \exists \bullet.\Gamma \leq \exists \Delta'.\Gamma'$, and θ is the mapping from $dom(\Delta')$ to $dom(\Theta)$ such that $\forall r \in dom(\Gamma'), \Gamma(r) = \Gamma'(r)[\theta]$, and $\forall \alpha \ll C \in \Delta', \Theta; \bullet \vdash \theta(\alpha) \ll C$, and $\Theta; \Psi; \Delta'; \Gamma' \vdash o : \tau$, then $\Theta; \Psi; \bullet; \Gamma \vdash o : \tau[\theta]$.*

Proof. By induction on operand typing rules.

case ot-v: trivial.

case ot-r: $o = r$ and $\tau = \Gamma'(r)$.

By ot-r, $\Theta; \Psi; \bullet; \Gamma \vdash r : \Gamma(r)$.

By $\Gamma(r) = \Gamma'(r)[\theta]$, $\Theta; \Psi; \bullet; \Gamma \vdash o : \tau[\theta]$.

case ot-vtable: $o = [r + 0]$, $\tau = \text{Vtable}(\omega)$, and $\Theta; \Psi; \Delta'; \Gamma' \vdash r : \text{Ins}(\omega)$.

By the induction hypothesis, $\Theta; \Psi; \bullet; \Gamma \vdash r : \text{Ins}(\omega)[\theta]$. $\text{Ins}(\omega)[\theta] = \text{Ins}(\omega[\theta])$.

By ot-vtable, $\Theta; \Psi; \bullet; \Gamma \vdash o : \text{Vtable}(\omega[\theta])$.

$\tau[\theta] = \text{Vtable}(\omega[\theta])$. Therefore, $\Theta; \Psi; \bullet; \Gamma \vdash o : \tau[\theta]$.

case ot-field: $o = [r + j]$, and $\tau = \tau_j$, and $\Theta; \Psi; \Delta'; \Gamma' \vdash r : \text{Ins}(\omega)$, and $\Theta; \Psi; \Delta' \vdash \omega \ll C$, and $\Theta(C) = C : B\{\tau_1, \ldots, \tau_n, \ldots\}$, and $1 \leq j \leq n$.

By the induction hypothesis, $\Theta; \Psi; \bullet; \Gamma \vdash r : \text{Ins}(\omega)[\theta]$. $\text{Ins}(\omega)[\theta] = \text{Ins}(\omega[\theta])$.

By Lemma 1 and $\Theta; \Psi; \Delta' \vdash \omega \ll C$, $\Theta; \Psi; \bullet \vdash \omega[\theta] \ll C$.

By ot-field, $\Theta; \Psi; \bullet; \Gamma \vdash [r + j] : \tau_j$.

$\tau_j[\theta] = \tau_j$ because τ_j does not have free type variables. Therefore, $\Theta; \Psi; \bullet; \Gamma \vdash o : \tau[\theta]$.

case ot-meth: similar to ot-field.

Lemma 5. *If $\Theta \vdash \exists \bullet.\Gamma \leq \exists \Delta'.\Gamma'$, and θ is the mapping from $dom(\Delta')$ to $dom(\Theta)$ such that $\forall r \in dom(\Gamma'), \Gamma(r) = \Gamma'(r)[\theta]$, and $\forall \alpha \ll C \in \Delta', \Theta; \bullet \vdash \theta(\alpha) \ll C$, and $\Theta; \Delta' \vdash \tau \hookleftarrow \tau'$, then $\Theta; \bullet \vdash \tau[\theta] \hookleftarrow \tau'[\theta]$.*

Proof. If $\tau = \tau'$ (rule at-ref), then $\Theta; \bullet \vdash \tau[\theta] \hookleftarrow \tau'[\theta]$.

If $\tau = \exists\alpha{\ll}C.\ \mathrm{Ins}(\alpha)$ and $\tau' = \mathrm{Ins}(\omega)$ and $\Theta; \Delta' \vdash \omega \ll C$ (rule at-pack), then by Lemma 1, $\Theta; \bullet \vdash \omega[\theta] \ll C$. $\tau[\theta] = \tau$ and $\tau'[\theta] = \mathrm{Ins}(\omega[\theta])$. By at-pack, $\Theta; \bullet \vdash \tau[\theta] \hookleftarrow \tau'[\theta]$.

Lemma 6. *If $\Theta \vdash \exists\bullet.\Gamma \leq \exists\Delta'.\Gamma'$, and θ is the mapping from $dom(\Delta')$ to $dom(\Theta)$ such that $\forall r \in dom(\Gamma'), \Gamma(r) = \Gamma'(r)[\theta]$, and $\forall \alpha \ll C \in \Delta', \Theta; \bullet \vdash \theta(\alpha) \ll C$, and $\Delta'; \Gamma' \vdash \{r \leftarrow \tau\}(\Delta''; \Gamma'')$, and $\bullet; \Gamma \vdash \{r \leftarrow \tau[\theta]\}(\Delta_1; \Gamma_1)$, then $\Theta \vdash \exists\Delta_1.\Gamma_1 \leq \exists\Delta''.\Gamma''$.*

Proof. If $\tau = \exists\alpha{\ll}C.\ \mathrm{Ins}(\alpha)$, then $\Delta'' = \Delta'; \beta \ll C$ and $\Gamma'' = \Gamma'[r : \mathrm{Ins}(\beta)]$ where β is a fresh type variable , and $\Delta_1 = \gamma \ll C$ and $\Gamma_1 = \Gamma[r : \mathrm{Ins}(\gamma)]$ where γ is a fresh type variable (rule asgn-e).

Define $\theta' : dom(\Delta'') \rightarrow dom(\Delta_1) \cup \Theta$ as:

$\theta'(\beta) = \gamma$ and $\theta'(\alpha) = \theta(\alpha)$ for all $\alpha \in dom(\Delta')$.

Then $\Gamma_1(r') = \Gamma''(r')[\theta']$ for all $r' \in dom(\Gamma'')$. $\Theta; \Delta_1 \vdash \alpha[\theta'] \ll B$ for all $\alpha \ll B \in \Delta''$.

By st-sub, $\Theta \vdash \exists\Delta_1.\Gamma_1 \leq \exists\Delta''.\Gamma''$.

If $\tau \neq \exists\alpha{\ll}C.\ \mathrm{Ins}(\alpha)$, then $\Delta'' = \Delta'$ and $\Gamma'' = \Gamma'[r : \tau]$, and $\Delta_1 = \bullet$ and $\Gamma_1 = \Gamma[r : \tau[\theta]]$ (rule asgn).

Then $\Gamma_1(r') = \Gamma''(r')[\theta]$ for all $r' \in dom(\Gamma'')$ and $\Theta; \bullet \vdash \alpha[\theta] \ll B$ for all $\alpha \ll B \in \Delta''$.

By st-sub, $\Theta \vdash \exists\Delta_1.\Gamma_1 \leq \exists\Delta''.\Gamma''$.

Theorem 3. *If $\Theta; \Psi \vdash P$, then $\exists P'$ such that $P \rightarrow P'$.*

Proof. By induction on the instruction typing rules and Lemma 7.

Lemma 7. *If $\Theta; \Psi; \bullet; \Gamma \vdash v : int$, then v is an integer.*

If $\Theta; \Psi; \bullet; \Gamma \vdash v : Code(\Phi)$, then v is a label and $H(v) = \Phi\{\ell_1 : \iota s_1, \ldots, \ell_n : \iota s_n\}$.

If $\Theta; \Psi; \bullet; \Gamma \vdash v : Ins(\omega)$, then v is a label and $H(v) = ins(\omega)\{v_1, \ldots, v_n\}$.

Proof. By inspection of value typing rules and heap value typing rules.

Theorem 4. *The iTal type system is sound.*

Proof. By Theorems 2 and 3.