Inferable Object-Oriented Typed Assembly Language

Ross Tate
University of California, San Diego
rtate@cs.ucsd.edu

Juan Chen
Microsoft Research
juanchen@microsoft.com

Chris Hawblitzel
Microsoft Research
chrishaw@microsoft.com

Abstract

A certifying compiler preserves type information through compilation to assembly language programs, producing typed assembly language (TAL) programs that can be verified for safety independently so that the compiler does not need to be trusted. There are two challenges for adopting certifying compilation in practice. First, requiring every compiler transformation and optimization to preserve types is a large burden on compilers, especially when adopting certifying compilation into existing optimizing non-certifying compilers. Second, type annotations significantly increase the size of assembly language programs.

This paper proposes an alternative to traditional certifying compilers. It presents iTalX, the first inferable TAL type system that supports existential types, arrays, interfaces, and stacks. We have proved our inference algorithm is complete, meaning if an assembly language program is typeable with iTalX then our algorithm will infer an iTalX typing for that program. Furthermore, our algorithm is guaranteed to terminate even if the assembly language program is untypeable. We demonstrate that it is practical to infer such an expressive TAL by showing a prototype implementation of type inference for code compiled by Bartok, an optimizing C# compiler. Our prototype implementation infers complete type annotations for 98% of functions in a suite of realistic C# benchmarks. The type-inference time is about 8% of the compilation time. We needed to change only 2.5% of the compiler code, mostly adding new code for defining types and for writing types to object files. Most transformations are untouched. Type-annotation size is only 17% of the size of pure code and data, reducing type annotations in our previous certifying compiler [4] by 60%. The compiler needs to preserve only essential type information such as method signatures, object-layout information, and types for static data and external labels. Even non-certifying compilers have most of this information available.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages, Theory, Verification

Keywords Type inference, Typed assembly language (TAL), Object-oriented compiler, Existential quantification, Certifying compiler

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

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 [17] and Morrisett *et al.* [15] introduced *proof-carrying code* and *typed assembly language* (TAL). 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 checker, rather than trusting an entire compiler.

Before a verifier can check the annotated assembly code, someone 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 [16]. 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 a compiler may require substantial effort. In our previous work, we modified about 19,000 lines of a 200,000-line compiler to implement type preservation [4]. 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 [4, 10, 14–16]. As a result, programmers currently face a trade-off: use a popular existing compiler that does not certify the safety of its output, or use one of the few available experimental type-preserving compilers, which are less optimizing, less documented, and less supported.

This paper proposes another approach to building certifying compilers—a *type inference* system called iTalX. Instead of having each stage explicitly track the type of each variable/register, our type inference algorithm will infer the types of the assembly language after all stages finish. This eases the implementation of both new certifying compilers and those retrofitted from legacy compilers, and vastly reduces the number of type annotations that compilers pass from stage to stage. iTalX requires only two kinds of annotation for instructions: the types of null-pointer literals and the length of jump tables. Null literals and jump tables appear only occasionally in code anyway. All other required type annotations are coarser-grained metadata: function signatures, object-layout information, and types of static data and external labels. Furthermore, there are no special pseudo-instructions, such as pack and unpack.

This paper makes the following contributions. First, we define a practical type system, iTalX, for assembly language, supporting classes, arrays, interfaces, casts, stacks, structs, by-reference arguments, and function pointers. We believe this to be the most extensive type system for object-oriented assembly language code at present.

Second, we prove that inference for iTalX is decidable and complete: if a fully annotated program is well-typed, we can erase the type annotations on basic blocks, and the inference algorithm can still infer valid types for the basic blocks.

It is very difficult to infer a type system with general existential quantification. First-class quantification easily leads to undecidable type inference [22, 23]. To make type inference decidable, we restrict quantification to class variables and integer variables. Nevertheless, our type system still supports expressive subclassing and integer constraints, making it suitable for typing realistic compiled object-oriented code.

Third, we implement type inference for the x86 assembly language, and show that our implementation can completely infer types for 98% of functions compiled from real, large C# benchmarks by the Bartok optimizing compiler. As far as we know, no other systems are able to infer types for real-world x86 benchmarks at a similar scale. Furthermore, omitting basic-block types reduces the size of type annotations in the generated TAL files by 60%. Type inference takes about 8% of the compilation time.

2. Language iTal

We first present iTal, a small inferable typed assembly language which is a stripped down version of iTalX. Although iTal is too small to be directly usable on real-world code, it illustrates the main features of our inference system, such as the treatment of type variables, subtyping, and joins. Section 3 shows how the full-fledged type system iTalX supports real-world features.

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 [12]. 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 inference algorithm uses a forward dataflow analysis, but unlike Java bytecode, it supports quantification over type variables. Such type variables allow us to check individual assembly language instructions for method invocation, array accesses, casts, and other operations (in contrast to Java bytecode, which treats each of these operations as single, high-level bytecode instructions). Consider a class Point with three fields and a virtual method Color that takes an instance of class RGB and returns void. The following code invokes *incorrectly* the Color method on p1:

```
class Point { int x; int y; RGB c;
              virtual void Color(RGB c); }

void Unsafe(Point p1, Point p2) {
    vt = p1.vtable; // fetch p1's vtable
    m = vt.Color; // fetch p1's Color method
    m(p2, p1.c); // call with p2 as "this"
}

class Point3D : Point { int z; }
```

This code above unsafely passes the wrong "this" pointer to p1's Color method: if p1 is an instance of the subclass `Point3D`, Color may refer to fields defined in `Point3D`, which p2 does not necessarily contain, since p2 may not be an instance of `Point3D`. Most type systems for object-oriented typed assembly languages use type variables to distinguish between safe method invocations and unsafe code [3, 4, 10]. LIL$_C$ [3], for example, describes p1's type and p2's type by existentially quantifying over a class variable α: both p1 and p2 have type $\exists\alpha\ll\texttt{Point}.\ \alpha$, which says that p1 and p2 are each instances of some class that is a subclass of `Point`. The virtual methods of the existentially quantified dynamic class α of p1 require the "this" pointer to be an instance of α. The type system conservatively assumes that the α for p1 may differ from the α for p2, ensuring that only p1 can be passed to p1's methods, and only p2 can be passed to p2's methods.

The rest of this section builds ideas from LIL$_C$ into a small class-based object-oriented typed assembly language iTal (inferable Tal), which supports type inference with existential types: type inference can infer all the types inside each function of any typeable iTal program without needing any annotations within the function bodies. It requires no type annotations on basic blocks, no annotations on instructions, and no type-coercion instructions.

We have proved that iTal is sound and that type inference is decidable and complete. The proofs and complete formalization of the semantics of a slight variant of iTal and its join algorithm can be found in our technical report [21].

The purpose of iTal is to shed some light on the more realistic iTalX described in the next section. Both systems use the same mechanisms for subtyping, joining, and inferring existential types, and both systems follow similar restrictions to keep inference decidable.

To make the key ideas stand out, iTal focuses on only core object-oriented features such as classes, single inheritance, object layout, field fetch, and virtual-method invocation. iTalX, however, applies to more expressive languages with arrays, casts, interfaces, stacks, by-reference parameters, and structs. The first three features require more significant changes to the type system. Others are mostly straightforward. The extensions are discussed in Section 3.

2.1 Syntax

iTal borrows ideas from LIL$_C$, a low-level object-oriented typed intermediate language [3]. LIL$_C$ 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 is at the assembly language level.

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

The source language type C is translated to the above existential type. An instance of a subclass of C can be "packed" to have type $\exists\alpha\ll C.\ \text{Ins}(\alpha)$. A value with type $\exists\alpha\ll C.\ \text{Ins}(\alpha)$ can be "unpacked" to have type $\text{Ins}(\beta)$, where β is a fresh class variable (distinct from any existing class variables) indicating the dynamic class of the instance, and the constraint $\beta \ll C$ records the fact that the instance's dynamic class inherits C.

The separation between static and dynamic classes guarantees the soundness of dynamic dispatch (Section 2.3 explains a virtual-method invocation example). A type system without such separation cannot detect the error in the previous unsafe example.

Class 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	$::=$	$\alpha \mid C$
reg type	τ_{Reg}	$::=$	Int $\mid$ Code$(\Phi \to \Phi')$ $\mid$ Ins(ω) $\mid$ Vtable(ω)
term type	τ	$::=$	$\tau_{\text{Reg}} \mid \exists\alpha{\ll}C.\ \text{Ins}(\alpha)$
value	v	$::=$	$n \mid \ell$
operand	o	$::=$	$v \mid r \mid [r+n]$
instr	ι	$::=$	bop r, o $\mid$ mov r, o $\mid$ mov $[r_1{+}n]{,}r_2$ $\mid$ call o
binary op	bop	$::=$	add $\mid$ sub $\mid$ mul $\mid$ div
instrs	ιs	$::=$	jmp ℓ $\mid$ bz o, ℓ_t, ℓ_f $\mid$ ret $\mid$ $\iota; \iota s$
func prec	Φ	$::=$	$\{r_i : \tau_i\}_{i=1}^{n}$
function	f	$::=$	$(\Phi \to \Phi')\ \{\overrightarrow{\ell : \iota s}\}$

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

iTal supports the primitive type Int and the code-pointer type Code$(\Phi \to \Phi')$ where $\Phi \to \Phi'$ describes function signatures with precondition Φ and postcondition Φ'. The other types are object-oriented: the type Ins(ω) describes instances of "exact" ω, and the type Vtable(ω) represents the virtual-method table of class ω. All these types can be used to type registers in basic-block preconditions, and are called register types. Type $\exists\alpha{\ll}C.\ \text{Ins}(\alpha)$ 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. Both register types and the above existential types are called term types.

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 instructions ended with a control-transfer instruction. Instruction jmp ℓ jumps to a block labeled ℓ. Instruction bz 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 $(\Phi \to \Phi')\ \{\overrightarrow{\ell : \iota s}\}$ specifies its signature $\Phi \to \Phi'$ and a sequence of basic blocks, each of which has a label ℓ and a body ιs. The notation $\vec{a}$ means a sequence of items in a.

2.2 Subclassing and Subtyping

We describe selected static semantics of iTal. The static semantics uses the following environments:

class decl	cdecl	$::=$	$C : B\{\vec{\tau}, \overrightarrow{\Phi \to \Phi'}\}$
class decls	Θ	$::=$	$\bullet \mid \Theta, \text{cdecl}$
constr env	Δ	$::=$	$\bullet \mid \Delta, \alpha \ll C$
reg bank type	Γ	$::=$	$\bullet \mid \Gamma, r : \tau_{\text{Reg}}$
state type	Σ	$::=$	$\exists\Delta.\Gamma$

Class declaration $C : B\{\vec{\tau}, \overrightarrow{\Phi \to \Phi'}\}$ introduces a class C with superclass B, fields with types $\vec{\tau}$, and methods with signatures $\overrightarrow{\Phi \to \Phi'}$. It specifies all fields and methods of C, including those from superclasses. Method bodies are translated to functions in the heap. Therefore, only method signatures are included in class declarations, not method bodies. Θ is a sequence of class declarations, which the compiler preserves to iTal.

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 register types.

iTal uses state types Σ, another form of existential types, to represent machine states, including 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 class variable is lifted to the constraint environment of the state type corresponding to the current machine state, and the register is given 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.

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. A class C is a subclass of B if C declares so in its declaration (rule sc-class). A class variable α is a subclass of a class name C if C is α's bound (rule sc-var). Additionally, subclassing is reflexive and transitive.

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

Subtyping between State Types. Subtyping between two state types is used to check control transfer. It is the key to type inference in iTal, allowing subtyping between two state types without first unpacking one type and then packing to the second type. No type coercion is necessary. 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.

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

A state type $\exists\Delta.\Gamma$ is a subtype of $\exists\Delta'.\Gamma'$ if a substitution θ maps each class variable in Δ' to either a class 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 r in $dom(\Gamma')$. The substitution must preserve subclassing in Δ': for each constraint $\alpha \ll C$ in Δ', $\theta(\alpha)$ must 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 the following from st-sub, one case of the traditional "pack" rule for existential types, using a substitution that maps α to C:

$$\Theta \vdash \exists\Delta.(\Gamma, r : \text{Ins}(C)) \leq \exists(\Delta, \alpha \ll C).(\Gamma, r : \text{Ins}(\alpha))$$

Subtyping between state types is reflexive and transitive, as implied by the st-sub rule. Reflexivity can be proved by using the identity substitution. Transitivity can be proved by composing substitutions.

No Subtyping between Term Types. Although iTal includes subtyping between state types Σ, it omits subtyping between term types τ, instead using a weaker notion of assignability. Omitting the subtyping relation $\tau \leq \tau'$ avoids issues of covariant and contravariant subtyping within code-pointer types, which makes it easier to join types. Our larger language iTalX allows subtyping, restricting function arguments to be contravariant to guarantee soundness and nearly invariant to guarantee decidability of inference.

Assignability. Assignability decides if the value in a register can be assigned to a memory location or a formal of a method, both of which can have existential types. Assignability allows a value of type τ to be assigned to a location of type τ. More importantly, it handles "packing" subclass instances to superclass instances (with existential types) by allowing a value of type Ins(ω) to be assigned to a location of type $\exists\alpha{\ll}\omega'.\ \text{Ins}(\alpha)$ whenever $\omega \ll \omega'$ can be inferred from the constraint environment. iTal uses assignability to avoid confusion with subtyping between state types.

2.3 Type Inference and Type Checking

Type inference computes the precondition 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 then uses a forward dataflow 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 "bz"), 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 fixed point. When joining two state types, the result is a supertype of both state types. The type system does not have infinite supertype chains for any given state type, which guarantees termination of type inference.

We use the following code segment to explain type inference and type checking. The example is contrived to show various aspects of type inference. Most compilers would generate better optimized assembly code.

The function Foo takes an instance of the previous class Point (in r1), an instance of Point3D (in r2), and an integer (in r3). Block L0 branches to L1 if r3 equals 0 and to L2 otherwise. L1 and L2 merge at L3, which calls the Color method (at offset 4 of the vtable) on either the instance of Point or the instance of Point3D, depending on the value of the integer. Instructions 3), 4), and 7) are added for the purpose of showing joining of types.

```
//void Foo(r1: Point, r2: Point3D, r3: int)
L0: 1)  bz r3, L1, L2   // branch on r3 equals 0
L1: 2)  mov r4, r1      // true branch
    3)  mov r5, r1
    4)  mov r6, r2
    5)  jmp L3
L2: 6)  mov r4, r2      // false branch
    7)  mov r5, r2
    8)  jmp L3
L3: 9)  mov r6, [r4+12] // get the RGB field
    10) mov r7, [r4+0]  // get vtable from r4
    11) mov r7, [r7+4]  // get the Color method
    12) call r7         // call the Color method
```

The signature of Foo is represented in iTal as $\Phi_{\text{Foo}} \to \{\}$, where Φ_{Foo} is $\{r1 : \exists\alpha{\ll}\text{Point. Ins}(\alpha), r2 : \exists\alpha{\ll}\text{Point3D. Ins}(\alpha), r3 : \text{Int}\}$. The precondition of block L0, Σ_0, is then $\exists\alpha_1 \ll \text{Point}, \alpha_2 \ll \text{Point3D}.\{r1 : \text{Ins}(\alpha_1), r2 : \text{Ins}(\alpha_2), r3 : \text{Int}\}$, by unpacking $r1$ and $r2$ in Φ_{Foo} and lifting the two fresh (and distinct) class variables α_1 and α_2 to the constraint environment.

Block L0 has precondition Σ_0. It has only one control-transfer instruction 1). Type inference checks that $r3$ has type Int and propagates the state type Σ_0 to L1 and L2 since instruction 1) does not change the machine state.

Block L1 now has precondition Σ_0. Checking instruction 2) adds the mapping $r4 : \text{Ins}(\alpha_1)$ to the current state type because $r4$ now contains a value of type $\text{Ins}(\alpha_1)$, and checking 3) and 4) is similar. Now we reach instruction 5) with state type $\Sigma_1' = \exists\alpha_1 \ll \text{Point}, \alpha_2 \ll \text{Point3D}.\{r1 : \text{Ins}(\alpha_1), r2 : \text{Ins}(\alpha_2), r3 : \text{Int}, r4 : \text{Ins}(\alpha_1), r5 : \text{Ins}(\alpha_1), r6 : \text{Ins}(\alpha_2)\}$, which becomes the precondition of the successor L3.

Similarly, checking block L2 produces post condition $\Sigma_2' = \exists\alpha_1 \ll \text{Point}, \alpha_2 \ll \text{Point3D}.\{r1 : \text{Ins}(\alpha_1), r2 : \text{Ins}(\alpha_2), r3 : \text{Int}, r4 : \text{Ins}(\alpha_2), r5 : \text{Ins}(\alpha_2)\}$. The successor L3 has its precondition already, so we need to compute the join of Σ_1' and Σ_2'. The result will be the new precondition of L3.

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 created during the joining process (called generalization variables). Generalization variables are not different from other class variables. We use the special notation to ease the presentation.

The join operation is performed in two steps. The first step generalizes the two register bank types in Σ_1 and Σ_2 to a common register bank supertype. To generalize Γ_1 and Γ_2, for each register r that appears in both Γ_1 and Γ_2, it generalizes $\Gamma_1(r)$ and $\Gamma_2(r)$ to a common supertype and maps r to the supertype in the result register bank type. The generalization omits registers that appear in Γ_1 or Γ_2 but not both.

Generalization recursively goes through the structure of types. Generalizing Int and Int returns Int. Generalizing $\text{Ins}(C)$ and $\text{Ins}(C)$ returns $\text{Ins}(C)$. Otherwise, generalizing $\text{Ins}(\omega_1)$ and $\text{Ins}(\omega_2)$ returns $\text{Ins}(\hat{\alpha})$, where $\hat{\alpha}$ is a fresh class variable (generalization variable). Generalization also records two mappings $\hat{\alpha} \mapsto \omega_1$ and $\hat{\alpha} \mapsto \omega_2$ to track where the new variable is from. The mappings will be used to construct substitutions for the subtyping rule st-sub. The bound of $\hat{\alpha}$ will be computed in the second step of join.

Generalizing $\exists\alpha{\ll}\omega_1.\ \text{Ins}(\alpha)$ and $\exists\beta{\ll}\omega_2.\ \text{Ins}(\beta)$ returns $\exists\gamma{\ll}\hat{\alpha}.\ \text{Ins}(\gamma)$, where $\hat{\alpha} \mapsto \omega_1$ and $\hat{\alpha} \mapsto \omega_2$. Generalizing $\text{Vtable}(\omega_1)$ and $\text{Vtable}(\omega_2)$ returns $\text{Vtable}(\hat{\alpha})$ with similar maps.

Code-pointer types $\text{Code}(\Phi_1 \to \Phi_1')$ and $\text{Code}(\Phi_2 \to \Phi_2')$ are generalized to $\text{Code}(\Phi \to \Phi')$ if Φ_1 and Φ_2 are generalized to Φ and Φ_1' and Φ_2' to Φ', provided no registers are dropped in either case. Our treatment of arguments is sound because iTal does not have subtyping on term types, as explained earlier.

Two types with different structures, such as a code-pointer type and a vtable type, cannot be generalized. If a register has differently structured types in the two register bank types to join, the join result will not contain the register.

For our example, $r4$ has type $\text{Ins}(\alpha_1)$ in Σ_1' and type $\text{Ins}(\alpha_2)$ in Σ_2'. Generalization creates a fresh variable $\hat{\alpha}$ and two mappings $\hat{\alpha} \mapsto \alpha_1$ and $\hat{\alpha} \mapsto \alpha_2$ for Σ_1' and Σ_2' respectively. For $r5$, it creates another fresh variable $\hat{\beta}$, different from $\hat{\alpha}$, and mappings $\hat{\beta} \mapsto \alpha_1$ and $\hat{\beta} \mapsto \alpha_2$. Generalization also creates new generalization variables for $r1$ and $r2$, but for simplicity of presentation, we keep the types of $r1$ and $r2$ unchanged after generalization, since Σ_1' and Σ_2' agree on their types and α_1 (α_2) means the same dynamic type of $r1$ ($r2$) in both Σ_1' and Σ_2'. The result of generalization is then $\{r1 : \text{Ins}(\alpha_1), r2 : \text{Ins}(\alpha_2), r3 : \text{Int}, r4 : \text{Ins}(\hat{\alpha}), r5 : \text{Ins}(\hat{\beta})\}$. Register $r6$ is omitted because it does not exist in Σ_2'.

The second step of the join operation is factorization. The goal is to compute the constraint environment of the result state type, using the least number of generalization variables. This step is done by unifying equivalent generalization variables.

We define an equivalence relation on generalization variables: two variables are equivalent iff they are results of generalizing the same two types. For example, in our example, $\hat{\alpha}$ and $\hat{\beta}$ are equivalent because both come from generalizing α_1 and α_2.

Factorization then creates the final join result $\exists\Delta.\Gamma$, where Δ collects arbitrarily chosen representatives of equivalence classes for generalization variables, and Γ is the register bank type resulting from generalization after substituting each generalization variable with the representative of its equivalence class. The bound of a representative $\hat{\alpha}$ in Δ is the least common superclass name for ω_1 and ω_2, where $\hat{\alpha}$ comes from generalizing ω_1 and ω_2.

For our example, the join of Σ_1' and Σ_2' is $\Sigma_3 = \exists\alpha_1 \ll \text{Point}, \alpha_2 \ll \text{Point3D}, \hat{\alpha} \ll \text{Point}.\{r1 : \text{Ins}(\alpha_1), r2 : \text{Ins}(\alpha_2), r3 : \text{Int}, r4 : \text{Ins}(\hat{\alpha}), r5 : \text{Ins}(\hat{\alpha})\}$. $\hat{\alpha}$ is chosen for the equivalence class $\{\hat{\alpha}, \hat{\beta}\}$ and given bound Point because Point is the least common superclass name for Point (α_1's bound) and Point3D (α_2's bound).

Σ_3 is a supertype of Σ_1' and Σ_2', using the st-sub rule. The joining process generates two maps, $\hat{\alpha} \mapsto \alpha_1$ and $\hat{\alpha} \mapsto \alpha_2$. Using the former map as the substitution in st-sub, we get $\Sigma_1' \le \Sigma_3$. The latter map evidences that $\Sigma_2' \le \Sigma_3$ holds. We have proved that the join process always computes the least upper bound [21].

Continuing our example, we now check block L3 with the new precondition Σ_3. Instruction 9) loads into $r6$ a field of $r4$ (an instance of $\hat{\alpha}$), using the object-layout information provided by the compiler. This is one of the few places where type inference needs hints (metadata here) generated by the compiler. The layout information maps a field offset to the field type. The class variable $\hat{\alpha}$ is not a concrete class, and thus its layout is statically unknown. From the fact that $\hat{\alpha}$ is a subclass of Point, the checker knows that an instance of $\hat{\alpha}$ has at least those fields declared in Point, and at the same offsets as in Point. It looks up the offset 12 (for the field c) in the layout information of Point and finds an existential type $\exists\gamma'{\ll}\text{RGB}.\ \text{Ins}(\gamma')$. When a register is assigned a field with an existential type, it is unpacked automatically. $r6$ is given type $\text{Ins}(\gamma)$, where γ is fresh and $\gamma \ll \text{RGB}$ is added to the constraints.

Instruction 10) loads into $r7$ the first word—the vtable—of $r4$. The type system gives type $\text{Vtable}(\hat{\alpha})$ to $r7$, indicating that $r7$ points to the dynamic type of $r4$'s dynamic type.

Instruction 11) loads into $r7$ the method pointer for Color. the checker uses the layout information of Point to find the code-pointer type $\text{Code}(\{r4 : \text{Ins}(\hat{\alpha}), r6 : \exists\eta{\ll}\text{RGB}.\ \text{Ins}(\eta)\} \to \{\})$. The first parameter is the "this" pointer, whose type is the same as the dynamic type of the object from which the method pointer is fetched. The second parameter is an instance of RGB. The two parameters are consumed by the method, returning nothing.

Instruction 12) calls the method in $r7$. The checker checks if the precondition of the callee is satisfied by the current machine state. Register $r4$ has the required type. Register $r6$ has type $\text{Ins}(\gamma')$ in the current state, and the callee requires an existential type $\exists\eta{\ll}\text{RGB}.\ \text{Ins}(\eta)$. We use assignability rules: register $r6$ can be assigned to the parameter because $r6$ has type $\text{Ins}(\gamma)$ and $\gamma \ll \text{RGB}$ can be inferred from the constraint environment, so $r6$ can be packed to the existential type. After the call, the state type contains the remaining registers and type inference continues.

This example illustrates how the various features of iTal work together to produce an inferable typed assembly language capable of verifying a simplistic object-oriented language. Many of the strategies we use in iTal can be extended to more expressive type systems. The use of existential quantification of class variables, the separation of assignability and subtyping, and the creation of generalization variables with mappings in order to join types are all more broadly applicable, as we demonstrate in the next section.

3. The iTalX Type System

This section describes how our more realistic type system iTalX expresses common language features, such as interior pointers, covariant arrays, type casts, interfaces, and the stack. iTalX is robust with respect to the common optimizations most compilers have, meaning these optimizations can be applied to any typeable program and the resulting program will still be typeable.

iTalX uses the same substitution-based subtyping rules for existentially quantified state types as iTal, and extends iTal's type-inference algorithm. iTalX introduces singleton integer types and integer variables with constraints to handle array-bounds checking. iTalX treats integer variables with constraints the same way as class variables with constraints with respect to subtyping and joining, which demonstrates the flexibility of our type system (Section 3.3).

iTalX uses the same existentially quantified state types for preconditions of basic blocks as iTal does. In fact, this is the only form of existential quantification in iTalX. iTalX uses a type $\text{SubInsPtr}(C)$ to represent the iTal type $\exists\alpha \ll C.\text{Ins}(\alpha)$. iTalX disallows existential quantification in term types—those used to type registers, stack slots, fields in objects, etc. Such quantification in term types is not needed in iTalX. We removed nested existential quantification from iTalX because having two layers of quantification causes complications with the join process. These complications do not arise in iTal because iTal does not have subtyping at the term-type level.

Many of the extensions in iTalX require complex constraints for class variables, whereas iTal needs only one simple subclassing constraint (an upper bound) for each class variable. Covariant arrays may need bounds that are arrays themselves. Type casting needs class variables with lower bounds and with other class variables as bounds. Interfaces have multiple inheritance, which means that class variables may have multiple bounds. iTalX separates class variables from their bounds to allow complex constraints on class variables. One environment collects all class variables (without constraints), and another collects constraints on class variables. iTal uses a single environment Δ for both purposes because a class variable is constrained only by a single upper bound. We use constraints and bounds interchangeably in this section.

3.1 Requirements for Inference

It is challenging to make a type system as expressive as iTalX inferable. For example, if class variables can be bound by other variables, the join of the tuples $\langle$`ArrayList`, `IList`, `IList`$\rangle$ and $\langle$`Hashtable`, `Hashtable`, `IDictionary`$\rangle$ is $\exists\alpha{\ll}\beta{\ll}\gamma.\langle\alpha,\beta,\gamma\rangle$ (where `ArrayList` implements `IList` and `Hashtable` implements `IDictionary`). Even though both types being joined use only two classes each, three variables are required to describe the join. The join also has to recognize the left-to-right inheritance hierarchy. Furthermore, as illustrated in Section 3.8, adding a simple feature such as null pointers can break type inference.

Fortunately, iTalX uses the results of an abstract framework based on category theory (described in our technical report [21]) to guarantee inferability. The framework describes how to construct joins in any existential type system that satisfies three properties, stated informally as follows: (1) if $\tau_1 \le \tau_2$, every free class variable in τ_2 occurs in τ_1; (2) term types have joins disregarding existentially quantified variables; (3) bounds and substitutions have a factorization structure [1], the metatheory behind the factorization process used in constructing joins for iTal. By computing joins, our type inference algorithm is made complete, meaning our inference algorithm will always infer a typing for any typeable program.

For decidability of inference, the main requirements are as follows: (1) instruction typing is monotonic, that is, if $\tau_1 \le \tau_2$, the postcondition when checking an instruction with τ_1 as the precondition should be a subtype of the postcondition when checking the same instruction with τ_2 as the precondition; (2) subtyping is well-founded: there are no infinite chains of strict supertypes.

We have translated both iTal and iTalX into the category-theoretic framework, and proven that they satisfy the requirements above. This proves that they are both inferable.

The rest of this section explains major extensions in iTalX. We introduce the constructs of iTalX as needed.

3.2 Interior Pointers and Records

iTal uses only a simple model of memory: registers may point to objects, which in turn may have fields pointing to other objects. Languages like C# support a more complex memory model, for example, a method may pass a reference to a field of an object as an argument to another method. The compiler represents this reference as an *interior pointer* into the middle of the object, and iTalX must be able to type such pointers. Furthermore, a C# reference may point to not just a single word in the object, but to an entire *struct* of multiple fields embedded directly in the object. In addition, even if the language does not require interior pointers, an optimizing compiler may introduce such pointers (e.g. when iterating through array elements).

To model this, iTalX breaks iTal's named reference type $\text{Ins}(\omega)$ into separate pointer types and name types, and adds a distinct record type where individual field types are made explicit:

$$
\begin{array}{lcl}
name & ::= & \textsc{Ins}(\omega) \mid \textsc{VTable}(\omega) \mid \ldots \\
\tau_{\text{Heap}} & ::= & \texttt{int} \mid \text{HeapPtr}(name) \mid \text{HeapPtr}_{\text{N}}(name) \mid \\
 & & \text{SubInsPtr}(\omega) \mid \ldots \\
rw & ::= & \texttt{R} \mid \texttt{RW} \\
recslot & ::= & \bullet \mid (rw : \tau_{\text{Heap}}) \\
rec & ::= & \{..., recslot, ...\} \\
\tau_{\text{Reg}} & ::= & \text{HeapRecPtr}(name : rec + n) \mid \\
 & & \text{HeapRecPtr}_{\text{N}}(name : rec + n) \mid \\
 & & \text{RecPtr}(rec + n) \mid \ldots
\end{array}
$$

Name types include $\textsc{Ins}(\omega)$ to name a class's instance type and $\textsc{VTable}(\omega)$ to name a class's vtable type.

The $\text{HeapPtr}(name)$ and $\text{HeapPtr}_{\text{N}}(name)$ represent pointers to objects on the heap. The subscript $_{\text{N}}$ denotes that the pointer may be `null`. For example, the iTalX type $\text{HeapPtr}_{\text{N}}(\textsc{Ins}(\omega))$ corresponds to the iTal type $\text{Ins}(\omega)$. There is also $\text{SubInsPtr}(\omega)$, which is equivalent to $\exists\alpha \ll \omega.\text{HeapPtr}_{\text{N}}(\textsc{Ins}(\alpha))$, but it has more restrictive subtypings than general existential quantification. Intuitively, $\text{SubInsPtr}(\omega)$ should be a subtype of $\text{SubInsPtr}(\omega')$ whenever ω inherits ω'; however, this breaks the rule that all variables in a supertype are present in the subtype. We can still allow $\text{HeapPtr}_{\text{N}}(\textsc{Ins}(\omega))$ to be a subtype of $\text{SubInsPtr}(\omega)$, though.

Records. Record types rec are used to represent object layouts. A record type contains zero or more record slots, each with a τ_{Heap} type. Each record slot is either read-write or read-only.

The separation of name types and record types avoids the difficulty of recursive types, such as when a class C has a field with type C. Slots of record types in iTalX do not contain other records or pointers to other record types — they only contain pointers to name types. When the program assigns a pointer-to-name type into a register or stack slot, iTalX automatically opens the name to a record type describing the corresponding layout. For example, when the type $\text{HeapPtr}(name)$ is assigned into a register, it is converted into the type $\text{HeapRecPtr}(name : rec + 0)$, where rec is the record type describing the layout of $name$. $\text{HeapRecPtr}(name : rec+n)$ represents an interior pointer, offset by n, to a record of type rec that is in the heap. A $\text{HeapRecPtr}_{\text{N}}(\textsc{Ins}(\omega) : rec + 0)$ can be assigned to a $\text{SubInsPtr}(\omega')$ field of a record provided ω inherits ω'. Note that this is not subtyping, but simply type-checking an assignment, so it does not break our framework. This reuses the concept of assignability that we used in iTal.

Whereas a HeapRecPtr points to records in the heap, a RecPtr can also refer to the stack of the caller. A $\text{HeapRecPtr}_{\text{N}}$ can forget its heap structure and become a RecPtr. Subtyping for record pointers is primarily inherited from prefix subtyping of their record types, but it can be more flexible to also incorporate offsets. $\text{HeapRecPtr}(name : rec + n)$ and $\text{RecPtr}(rec + n)$ represent interior pointers when the offset n is positive.

When opening a name to a record type, we may not statically know the layout of the entire record, such as when the name refers to the dynamic type α of an object. Opening a name $\textsc{Ins}(\alpha)$ (representing an instance of α), where α is a subclass of C, results in a record type that contains C's fields, except the vtable field has name $\textsc{VTable}(\alpha)$ to guarantee soundness of dynamic dispatch.

3.3 Arrays

Adding support for arrays goes as follows. First, we introduce class types for array classes and adjust how to join existential types accordingly. Second, we add existential quantification of integers and simple arithmetic expressions in order to do array-bounds checking. Third, we add ordering constraints on integers. Fourth, we introduce extended records to encode records (array headers) with a subrecord which repeats an unknown number of times (array elements). Lastly, we allow names to open to existentially quantified extended-record types.

Array Classes. In iTal, a class type is either a class variable or a class constant. Now we add a class constructor Array:

$$\omega ::= \ldots \mid \text{Array}(\omega)$$

$\text{Array}(\omega)$ always extends the `Array` class, per C#'s array classes.

When constructing the join we have to infer when to use array classes. For this, we again use the two maps constructed during generalization in iTal's join algorithm (we refer to them as R_1 and R_2). If R_1 maps a generalization variable $\hat{\alpha}$ to $\text{Array}(\omega_1)$ and R_2 maps $\hat{\alpha}$ to $\text{Array}(\omega_2)$, we introduce a fresh generalization variable $\hat{\beta}$ mapping to ω_1 and ω_2 and substitute $\hat{\alpha}$ with $\text{Array}(\hat{\beta})$. If $\hat{\beta}$ also maps to two array types, we repeat this process recursively.

Existentially Quantifying Integers. To verify array accesses, we add existentially quantified integer variables. By integers, we mean mathematical integers, not 32-bit integers. Existential quantification may introduce integer variables i with simple arithmetic expressions of the form $I = i * a + b$ as constraints, where a and b are (mathematical) integer constants. The type $\text{Int}(I)$ is a singleton integer type representing the single integer value I:

$$\tau_{\text{Reg}} ::= \ldots \mid \text{Int}(I)$$

Joining existentially quantified simple arithmetic expressions poses an interesting challenge. The join of the states $\{r : \text{Int}(4)\}$ and $\{r : \text{Int}(10)\}$ is $\exists i.\{r : \text{Int}(i * 6 + 4)\}$ or equivalently $\exists i.\{r : \text{Int}(i * 6 + 10)\}$. The join of the states:

- $\exists i.\{r : \text{Int}(i * 4 + 4), r' : \text{Int}(i * 8)\}$
- $\exists j.\{r : \text{Int}(j * 6 + 2), r' : \text{Int}(j * 12 - 4)\}$

is $\exists k.\{r : \text{Int}(k{*}2{+}4), r' : \text{Int}(k{*}4)\}$, with substitutions $k \mapsto i{*}2$ and $k \mapsto j * 3 - 1$. More broadly, the join generalizes integer types by introducing variables to represent them, similar to iTal's generalization of class types: $\text{Int}(I_1)$ and $\text{Int}(I_2)$ are generalized to $\text{Int}(\hat{i})$ where $\text{Int}(\hat{i})$ is a fresh generalization variable and $\hat{i} \mapsto I_1$ and $\hat{i} \mapsto I_2$.

As with iTal's join in section 2.3, the join for integer types defines an equivalence relation on generalization variables. Consider, for simplicity, equivalences for types of the form $I = i * a$ where $a \geq 1$. Then the equivalence $\hat{i} \equiv \hat{j}$ holds if:

- $R_1(\hat{i}) = k_1 * a_1$ and $R_1(\hat{j}) = k_1 * b_1$
- $R_2(\hat{i}) = k_2 * a_2$ and $R_2(\hat{j}) = k_2 * b_2$
- $\frac{a_1}{a_2} = \frac{b_1}{b_2}$ (so that $\frac{a_1}{gcd(a_1,a_2)} = \frac{b_1}{gcd(b_1,b_2)}$)

The join then designates a fresh variable k for each equivalence class of generalization variables, then substitutes each generalization variable with an appropriate expression in terms of k. Suppose we have a generalization variable $\hat{i}$ with $R_1(\hat{i}) = k_1 * a_1$, $R_2(\hat{j}) = k_2 * a_2$, and k is the fresh variable designated for $\hat{i}$'s equivalence class; then the join substitutes $\hat{i}$ with $k * gcd(a_1, a_2)$ where gcd is the greatest common divisor. The mappings $k \mapsto$

$k_1 * \frac{a_1}{gcd(a_1,a_2)}$ and $k \mapsto k_2 * \frac{a_2}{gcd(a_1,a_2)}$ serve as evidence that this construction forms a common supertype of the two types being joined. The use of gcd is necessary since a coefficient for k smaller than $gcd(a_1, a_2)$ would fail to yield the best common supertype, while a coefficient larger than $gcd(a_1, a_2)$ would fail to yield a common supertype at all.

The generalization of the join beyond $I = i * a$ to all forms of I is straightforward (see Granger [7] for a thorough discussion of joining integer equalities).

Ordering Integer Expressions. To check array bounds, our existential quantifications also need to include ordering constraints of the form $I_1 <_{32+} I_2$. This constraint means that, viewed as unsigned 32-bit integers, I_1 is strictly less than I_2. Thus, we view the machine as capable of manipulating mathematical integers, but the comparisons are limited to a 32-bit perspective on these mathematical integers. Dereferencing also has a 32-bit perspective, so we can rely on 32-bit ordering constraints to verify array accesses. The reasons for this unusual perspective are contained within our abstract framework. In short, complications arise because $i * 4$ is not an injective operation on 32-bit integers.

iTalX restricts which ordering constraints can be present in an existential type. In particular, iTalX permits the ordering constraint $I_1 <_{32+} I_2$ to be present in an existential type if both I_1 and I_2 occur in the body of the existential type. iTalX also allows the above ordering constraint if I_1 is a constant and I_2 occurs in the body. A constraint is not permitted if it does not satisfy either of these conditions. This restriction bounds the number of constraints present in iTalX's existential types, essentially discarding all constraints that are irrelevant to type checking the program. We found this bound to be important to achieving an efficient implementation.

At present, iTalX does not use any arithmetic inference rules in its subtyping rules, only that $<_{32+}$ is transitive. This prevents any complications with arithmetic overflow, but also prevents iTalX from handling array-bounds-check elimination. The above restriction on constraints, however, would allow us to extend iTalX with arithmetic inference rules that are sound even in the presence of arithmetic overflow, while still keeping the type system well-founded as required by our inference algorithm. Such an extension is considered future work.

Extended Records. An extended-record type is used to represent array layouts. It consists of a record type (array header), a "fixed-length" name (the elements), and an integer expression (the number of elements): $rec \circ name_I$. Although in general a name can actually describe an extended-record type, a fixed-length name must describe a record type of a predetermined length. This allows us to identify which index of an array and which field within that index that an interior pointer is referencing.

Arrays have statically indeterminable lengths. To refer to the length of an array, we allow names to open to existentially quantified record types, although any existentially quantified variables and constraints are immediately pulled into the outer existential bound upon opening the name. For example, the name $\textsc{Ins}(\text{Array}(\alpha))$ opens to the existentially quantified record type

$$\exists \ell. \left\{ \begin{array}{ll} \mathtt{R}: & \text{HeapPtr}(\text{V}\textsc{Table}(\text{Array}(\alpha))) \\ \mathtt{R}: & \text{HeapPtr}(\textsc{Runtime}(\alpha)) \\ \mathtt{R}: & \text{Int}(\ell) \\ & \circ\ \textsc{SubInsPtrRec}(\alpha)_\ell \end{array} \right\}$$

where ℓ is an existentially quantified integer variable indicating the length of the array, and ℓ would be pulled into the environment. We represent the fact that the third field of the header is also the length of the array by using ℓ in both positions. The second field is used to type check covariant arrays using the techniques described next. $\textsc{SubInsPtrRec}(\alpha)$ is the fixed-length name for the record type $\{\mathtt{RW} : \text{SubInsPtr}(\alpha)\}$, representing the elements of the array. Thus, iTalX type checks arrays and array accesses by combining our earlier concepts of existential quantification, records, and names.

3.4 Type Casts and Runtime Types

Type checking downward type casts requires adding class variables with lower bounds and variables bounded by other variables.

Downward type casts test at run time whether an object is an instance of a class. Each class has a unique identifier, called its runtime type. Two runtime types are equal if and only if the corresponding classes are the same. The runtime type of a class points to the runtime type of its immediate superclass, and such pointers form a runtime type chain. A typical implementation of downward type casts walks up the chain to find if a superclass matches the class to which we want to cast.

iTalX uses the name $\textsc{Runtime}(\omega)$ to represent the runtime type of a class ω:

$$name ::= \ldots \mid \textsc{Runtime}(\omega)$$

To represent the pointer to the runtime type of the superclass, we reuse the concept of existentially quantified records that we introduced for arrays. $\textsc{Runtime}(\omega)$ opens to the following existentially quantified record type:

$$\exists \beta \gg \omega.\{\ldots, \mathtt{R} : \text{HeapPtr}_{\text{N}}(\textsc{Runtime}(\beta)), \ldots\}$$

Note that if ω were a class variable α, this would introduce a constraint $\alpha \ll \beta$ between two class variables, justifying the need for more complex constraints in iTalX.

When walking up the runtime type chain to cast an object of class α to a class C, if a (possibly null) runtime type with name $\textsc{Runtime}(\gamma)$ matches the (non-null) runtime type of our target class C, the type inference concludes that $\gamma = C$, and uses that to check the instructions that follows. In particular, the quantification used above would inform us that α inherits γ, so the equality $\gamma = C$ informs us that α inherits C, indicating that the object can be safely treated as an instance of a subclass of C.

3.5 Interfaces

To support interfaces, iTalX distinguishes class variables that can be instantiated with classes from those instantiated with interfaces, using a subscript "C" or "I" on a variable respectively. For example, iTalX uses the constraint α_{C} to indicate that the variable α can only be instantiated with classes. Furthermore, variables can have more than one bound because classes can inherit multiple interfaces.

To compute the join of class variables with such constraints, we again use the two generalization maps R_1 and R_2. A generalization variable $\hat{\alpha}$ will inherit all of the (possibly implicit) constraints on $R_1(\hat{\alpha})$ and $R_2(\hat{\alpha})$. For example, if both $R_1(\hat{\alpha})$ and $R_2(\hat{\alpha})$ are constrained to be classes, the generalization variable $\hat{\alpha}$ will be as well. Similarly, given two generalization variables $\hat{\alpha}$ and $\hat{\beta}$, if $R_1(\hat{\alpha}) \ll R_1(\hat{\beta})$ and $R_2(\hat{\alpha}) \ll R_2(\hat{\beta})$ hold, we add the constraint $\hat{\alpha} \ll \hat{\beta}$. We also have to infer inheritance constraints with respect to class and interface names such as `ArrayList` and `IList`. For upper bounds this poses no problem since any class or interface only inherits a finite number of classes and interfaces. For lower bounds, however, due to multiple inheritance there may be an infinite number of classes and interfaces which inherit both $R_1(\hat{\alpha})$ and $R_2(\hat{\alpha})$. To address this issue, we also allow class variables to be bounded below by a tensor $\otimes$ of class and interface names provided there exists a class or interface that inherits all those in the tensor. This restriction on valid tensors bounds their size in a given program, which keeps our type system well-founded. Finally, if a class or interface inherits all the tensored classes and interfaces bounding α, then that class implicitly inherits α as well. Thus, the join of `ISerializable` and `IList` is $\exists \alpha_{\text{I}} \gg \mathtt{ISerializable} \otimes \mathtt{IList}.\alpha$

and we can infer that `ArrayList` inherits α since `ArrayList` implements both `ISerializable` and `IList`. These techniques grant us an inferable type system capable of casting and even interface-method lookup in a multiple inheritance context (the latter process is described in detail in the Appendix).

3.6 Generics

To support generics, we can reuse many of the same techniques we used to support array classes. If generic class types $C(\omega_1, \ldots, \omega_n)$ and $C(\omega'_1, \ldots, \omega'_n)$ are generalized to a generalization variable $\hat{\alpha}$, we introduce n fresh generalization variables $\hat{\alpha}_i$ mapping to ω_i and ω'_i and substitute $\hat{\alpha}$ with $C(\hat{\alpha}_1, \ldots, \hat{\alpha}_n)$. If any of the $\hat{\alpha}_i$ also map to two similar generics, we repeat this process recursively.

We also have to extend the constraint environment to include constraints of the form $\alpha \ll C(\vec{\omega})$ or $C(\vec{\omega}) \ll \alpha$, where $\vec{\omega}$ may also contain class variables. In fact, the lower bound on an interface variable α may need to be a tensor of generics (with their arguments supplied). However, we never need constraints of the form $C(\vec{\omega}) \ll D(\vec{\omega}')$; because of how inheritance can be specified in C#, constraints of this form can always be simplified. Even with these more complex constraints, type inference is still decidable.

3.7 Using the Stack

iTal has no concept of a stack, an obvious shortcoming since the stack plays such an important role at the assembly level. Here we make simple extensions to iTalX to let it use the stack intraprocedurally and interprocedurally. We represent the stack essentially as a partial map from non-negative integers, marking stack slots in the current stack frame, to register types. We use StackPtr(n) to access and manipulate the stack. By using only non-negative integers in the stack, we prevent a callee from changing the caller's stack. These simple extensions allow iTalX to use the stack intraprocedurally, but we need to refine code-pointer types in order to use the stack interprocedurally.

Code Pointers. A code-pointer type in iTalX, as in iTal, is specified simply as a required input state type and a produced output state type. These state types are a stack type and a register bank type; however, they cannot refer to register types τ_{Reg} (defined in Section 3.2, along with heap types τ_{Heap}). The output state type can only refer to heap types and stack-pointer types. The input state type can only refer to heap types, minus code-pointer types, and stack-pointer types along with two additional types. The first type, ReturnAddress, tells the caller where the return address should be stored. ReturnAddress is also a register type, used to type check the `ret` instruction. The second type is ParamPtr$(name)$, a new pointer type which can be used only as an input type. Unlike the other input pointer types HeapPtr and SubInsPtr, ParamPtr need not refer to the heap. In the callee, a ParamPtr$(name)$ will be translated into a RecPtr of the record type that $name$ opens to; thus, the caller can pass any pointer whose referenced space will look like the appropriate record type for the duration of the call. This allows the caller to pass even a stack pointer, provided the referenced portion of the stack is appropriately typed at the time of the call and does not overlap with the callee's stack frame. ParamPtrs allow us to pass references to local variables, fields, and array indices per the pass-by-reference semantics of C#'s `ref` keyword.

Callee-Save Registers A common calling convention requires the callee to ensure that the values of certain registers upon entering the function are the same upon exiting the function. This convention is known as callee-save registers. We incorporate this into iTalX by having each code-pointer type declare the set of registers whose values will be preserved. Subtyping of code-pointer types allows this set to be smaller in supertypes. We type check callee-save registers in the usual manner: each callee-save register is given its own type variable at the beginning of the function body and must have that same type variable upon returning from the function.

3.8 Null Pointers

The extensions above capture most features of C# except one seemingly unremarkable feature: null pointers. Although iTalX has nullable heap pointers, it does not have an explicit `null` type. We do this for a very good reason: null pointers break joins in the presence of existential quantification, breaking the inference process. Our framework even suggests this, since the α in the simple rule $\texttt{null} \leq \text{HeapPtr}(\textsc{Ins}(\alpha))$ is not used in the subtype `null`. Without changing `null` to already refer to α, there is no way to resolve this problem. Fortunately, we can illustrate the problem concretely and concisely using some shorthand. Take the two existentially quantified triples $\tau_\alpha := \exists\alpha.\langle\texttt{null}, \alpha, \texttt{null}\rangle$ and $\tau_\beta := \exists\beta, \beta'.\langle\beta, \texttt{null}, \beta'\rangle$. The join of τ_α and τ_β cannot contain `null`. The two existentially quantified triples $\tau_\gamma := \exists\gamma, \gamma'.\langle\gamma, \gamma, \gamma'\rangle$ and $\tau_\delta := \exists\delta, \delta'.\langle\delta, \delta', \delta'\rangle$ are both supertypes of both τ_α and τ_β. Their only common subtype without `null` is $\tau_\rho := \exists\rho.\langle\rho, \rho, \rho\rangle$, but τ_ρ is not a supertype of τ_β. Thus, τ_α and τ_β have no join. This example illustrates how some of the most intuitive types can break an inference algorithm. Although C# has `null`, it always occurs where the class that it is a null pointer of can be easily discerned. In the lowering stage, when we replace `null` with 0, we include an annotation indicating that that occurrence of 0 has type NullPtr(ω), where ω is the class or interface associated with that use of `null`. NullPtr(ω) is a subtype of both $\text{HeapPtr}_{\text{N}}(\text{Ins}(\omega))$ and $\text{Int}(0)$.

3.9 Theorems

We have proven the following properties for inference of iTalX:

Decidability. The inference algorithm described in Section 2.3 extended to iTalX halts.

Completeness. If an iTalX function is typeable, the inference algorithm described in Section 2.3 extended to iTalX infers a valid typing of that function.

These theorems result primarily from our categorical framework for existential types described in our technical report [21]. The proof strategy extends the strategies used for iTal. The proofs for a slight variant of iTal can also be found in our technical report.

4. Implementation

Here we present our prototype implementation of a type inference engine for iTalX on the output of a large-scale object-oriented optimizing C# compiler called Bartok. We show that: (1) it is practical to infer types for an expressive TAL such as iTalX; (2) type inference needs much less effort from the compiler and much fewer type annotations, compared to traditional certifying compilation.

Our base compiler Bartok compiles Microsoft .NET bytecode programs to standalone x86 executables. It is not a just-in-time compiler. The compiler has about 200,000 lines of code, mostly written in C#, and is fully self-hosting. Performance of Bartok's generated code is comparable to performance under the Microsoft Common Language Runtime (CLR). According to the benchmarks tested, programs compiled by Bartok are 0.94 to 4.13 times faster than the CLR versions, with a geometric mean of 1.66. Throughout this evaluation we compare against our previous work [4] in which we built a traditional certifying compiler, also based on Bartok, by making every compilation phase preserve types.

For our benchmarks, about 98% of methods are inferable. As far as we know, no other systems are able to infer types for real-world x86 benchmarks at a similar scale.

We changed about 2.5% of the compiler code, about 5,000 lines of code out of 200,000 lines. Among the 5,000 lines, about 4,500 lines are for adding new code to define iTalX types and to write metadata such as the class hierarchy, record layouts, and function signatures into the object files in terms of iTalX's type system. The compiler transformations and optimizations are mostly untouched. Our previous traditional certifying compiler changed about 10% (19,000 lines) of the compiler code. It required every transformation and optimization to preserve types, and therefore modified many more compilation phases. Although our experience showed that most optimizations can be made to preserve types easily, changing 19,000 lines of code is still a large burden on the compiler writers, especially figuring out where changes are needed and what types in the complex TAL type system to use. Compiler writers who build certifying compilers from scratch also have to think about maintaining the right type information in every optimization if they follow the traditional type-preserving approach.

Our type inference engine mainly consists of the iTalX definitions (5,000 lines of C# code), an x86 disassembler (4,700 lines), and the type inference (about 4,100 lines). The main differences between our type inference engine and our previous traditional certifying compiler's type checker are the definitions of state types and the computation of joins, which add up to about 1,300 lines of code. We chose to increase the trusted computing base slightly to relieve the compiler from full-blown certifying compilation. In order to reduce the trusted computing base, we could separate type inference and type checking into two phases so that only the type checking phase would be trusted.

Type annotation needed by our type inference implementation is about 60% less than that required by our previous traditional certifying compiler. Size of type annotations required by inference is only about 17% of the size of *pure code and data* in object files, compared with 36% for the previous certifying compiler. It indicates that type-annotation size may no longer be a big obstacle for adopting certifying compilation.

Our implementation supports allocating C# structs on the stack, without annotations specifying the struct type during allocation. Type inference supports initializing a struct field by field and then using a pointer to the first field as a pointer to the whole struct. It even supports passing structs as parameters on the stack. We are unaware of any other systems with similarly flexible stack support.

The implementation also supports jump tables (a more efficient way to compile switch statements) by disassembling the data section where the jump tables are stored to figure out the jump targets. The compiler only needs to annotate the jump instruction with the length of the jump table. We also extend our permitted integer constraints to include expressions bounded above by constants less than or equal to the length of the largest jump table in the function. This way we can ensure the assembly code is accessing the jump table correctly, while still keeping our type system inferable.

The implementation extends iTalX slightly to address features such as type arguments for polymorphic methods. For polymorphic method calls, we infer the type arguments instead of relying on type annotations. Type inference for type arguments of polymorphic functions is in general undecidable [19], but currently we support only polymorphic methods for type casts and memory allocation, where inferring type arguments is simple in these special cases.

Our implementation does not support exceptions or delegates. Those are considered future work. Our framework can handle generics, but our prototype does not include it because Bartok fully instantiates generics before code generation. We have not yet added type annotations for null literals, as discussed in Section 3.8. This would only cause our type inference to report null-related type errors when it should not. We expect that the missing annotations would have little impact on type-annotation size.

Measurement. Here we describe measurement of type-annotation size and type-inference time on our benchmarks. We chose the seven major benchmarks used in our previous work, which range from 54KB to 21MB in object file size not including libraries (see Table 1). We compile the benchmarks separately from the libraries, to focus on the user programs. The object files include type annotations for type inference. We compile with all of Bartok's standard optimizations (more than 40 of them) turned on, except for three optimizations that our inference cannot yet handle: array-bounds-check elimination, redundant-type-test elimination, and inlined memory allocation.

Name	**Description**	**Obj. Size** (B)
ahcbench	Adaptive Huffman Compression.	54,922
asmlc	A compiler for ASML.	21,276,036
bartok	An older version of Bartok itself.	10,051,871
lcscbench	The front end of a C# compiler.	9,623,384
mandelform	Mandelbrot set computation.	78,544
sat_solver	a SAT solver written in C#.	369,797
zinger	A model checker for the zing model.	1,167,567

Table 1. Benchmarks.

Benchmarks	**Succ.**	**Total**	**Succ./Total** (%)
ahcbench	67	67	100.0
asmlc	15,820	16,462	96.1
bartok	7,037	7,222	97.4
lcscbench	5,718	5,860	97.6
mandelform	12	12	100.0
sat_solver	274	274	100.0
zinger	1,125	1,189	94.6
		Geomean	97.9

Table 2. Number of Successfully Inferred Methods

Benchmarks	**Infer.**	**TAL**	**Infer./TAL** (%)
ahcbench	3,320	5,936	55.9
asmlc	644,937	2,745,130	23.5
bartok	334,590	1,448,867	23.1
lcscbench	256,116	911,308	28.1
mandelform	3,732	4,716	79.1
sat_solver	13,560	22,828	59.4
zinger	45,411	114,084	39.8
		Geomean	39.8

Table 3. Type-Annotation Size (in bytes)

About 98% of methods in the benchmarks are inferable (see Table 2). All methods in the small benchmarks (ahcbench, sat_solver, and mandelform) are inferable. For the large benchmarks (asmlc, bartok, lcscbench, and zinger), type inference fails on a small number of methods because the methods use unsupported language features, such as delegates, or interact with unsafe code, say by using Platform Invocation Services (PInvoke).

Table 3 compares the type-annotation sizes: type inference needs about 23%-79% of the type annotation required by our previous certifying compiler, with a geometric mean of 40%, which is about a 60% reduction on the type-annotation size. We see more size reduction on large benchmarks than on small ones because small benchmarks do not have many annotations to begin with and types for static data and function signatures are more dominating than those in large benchmarks.

Benchmarks	Infer.	Comp.	Infer./Comp. (%)
ahcbench	0.1	4.8	1.9
asmlc	21.6	135.2	16.0
bartok	24.2	69.6	34.7
lcscbench	8.5	61.3	13.9
mandelform	0.1	10.5	1.3
sat_solver	0.6	6.7	9.3
zinger	2.1	15.2	13.7
		Geomean	8.2

Table 4. Type-Inference Time vs. Compilation Time (in seconds)

Table 4 shows the type-inference time compared with compilation time (including writing metadata to file). The numbers were measured on a PC running Windows Vista with a 3GHz quad core CPU and 4GB of memory. Type inference in our current implementation is slower than type checking in the previous certifying compiler: type inference takes about 1%-35% of compilation time, with a geometric mean of 8%, whereas type checking in the previous certifying compiler takes less than 3% of the compilation time.

The difference is mainly because type inference is more sensitive to the control flow structures of methods. Straight-line code is easy to infer; type inference scans code only once and thus can be as efficient as type checking. For methods with complex loops, type inference sometimes takes much longer to reach a fixed point for preconditions of basic blocks without the guidance of type annotations. The type checker with full annotations needs to scan code only once no matter how complex the code structure is, because a basic block at each control merge point is annotated with its precondition. The Bartok benchmark is an outlier for type-inference time. It has more than 7,000 methods. The largest 23 methods in the benchmark have complex control flow graphs and together take about half of the total type-inference time.

One approach to getting more efficient type inference even with complex control flow structures is to ask for slightly more type annotation from the compiler, such as loop invariants, so that the type inference engine can reach the fixed point faster. We consider this approach future work.

One lesson we learned from our implementation experience is that memoizing large types does not pay off when we do not compare those types for equality often. State types in iTalX are large and complicated because they model machine states. Our first implementation memoized state types, which required substitutions (because state types are quantified types) and structural equality, and the type-inference time took about 3%-323% of the compilation time (with a geometric mean of 36%). With no memoization of state types and a few other fine-tunings, our current implementation is much more efficient.

5. Related Work

Hindley-Milner type inference [13] is used by the ML and Haskell programming languages. The Hindley-Milner algorithm 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 [23]. Extensions to the Hindley-Milner approach supporting first-class quantified types require some type annotations [9, 11] or pack/unpack annotations [8]. Alternatives to the Hindley-Milner approach, such as local type inference [20], also require some type annotations. Although these extended and alternative algorithms [8, 9, 11, 20] were developed for functional languages, they could be applied to a typed assembly language like iTalX 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. In order to accomplish type inference, iTalX restricts what quantified variables may represent. In this respect, our work is most similar to the Pizza language's internal type system [18], whose existential types quantify over named classes rather than over all types. Like iTalX, Pizza's internal type system defines a join operation over existential types. However, to the best of our understanding, the operation computes the join of $\exists\alpha.\mathtt{IList}(\alpha)$ with itself as a type of the form $\exists\alpha \ll \vec{C}.\alpha$, where $\vec{C}$ is a set of classes not containing α. Regardless of the contents of $\vec{C}$, this type cannot be equivalent to $\exists\alpha.\mathtt{IList}(\alpha)$, and therefore cannot be the join. This complication is simply a demonstration of how challenging inference of existential types can be.

SpecialJ [5] is a certifying compiler for Java that uses a proof generator to create proofs of safety for assembly language. However, SpecialJ's proof generation relies on compiler-generated loop invariants, whereas iTalX infers loop invariants automatically.

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, corresponding to existentially quantified variables in iTalX's state types. Coolaid is more specialized towards a particular source language and a particular compilation strategy than most typed assembly languages are, whereas iTalX encodes objects and classes using more standard general-purpose types (namely existential 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". We envision iTalX as exactly such an object-oriented TAL.

6. Conclusions

We have formalized and implemented type inference for iTalX, a typed assembly language capable of verifying optimized assembly code compiled from object-oriented languages like C# and Java. Currently, the implementation completely infers the types for about 98% of functions in our benchmark suite. Inferring most of the remaining 2% appears to be a matter of engineering the inference implementation to recognize idioms such as Bartok's implementation of delegates. It may also require modifications to the compiler, such as propagating types of null-pointer literals, but such modifications are minor compared to the effort of implementing type preservation throughout a large compiler. Based on this, it appears feasible to use inference as the primary mechanism for generating TAL types from a large optimizing compiler, only rarely disabling optimizations or falling back to a smaller type-preserving compiler.

Although our type system is not yet able to support all optimizations (e.g. array-bounds-check elimination), it supports the common optimizations essential to generating good code from object-oriented languages. Only 3 out of more than 40 optimizations in Bartok are not supported. Based on the abstract framework underlying our type system, we believe that inference can readily be adjusted to accommodate new language features. We are currently investigating adding null-dereference checking and more powerful array-bounds checking directly to our type system by expanding the capabilities of our existential bounds. As languages like Java and C# evolve, so will our inferable typed assembly language.

Acknowledgements. We would like to thank Francesco Logozzo for discussions about numerical abstract domains, as well as our anonymous reviewers for their insightful feedback.

References

[1] J. Adámek, H. Herrlich, and G. E. 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 *TLDI*, pages 91–102, 2005.

[3] J. Chen and D. Tarditi. A simple typed intermediate language for object-oriented languages. In *POPL*, pages 38–49, 2005.

[4] 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 *PLDI*, pages 183–192, 2008.

[5] C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *PLDI*, pages 95–107, 2000.

[6] A. Goldberg. A specification of java loading and bytecode verification. In *Computer and Communications Security*, pages 49–58, 1998.

[7] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT*, volume 1, pages 169–192, 1991.

[8] M. P. Jones. First-class polymorphism with type inference. In *POPL*, pages 483–496, 1997.

[9] D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System F. In *ICFP*, pages 27–38, 2003.

[10] C. League, Z. Shao, and V. Trifonov. Type-preserving compilation of Featherweight Java. *TOPLAS*, 24(2):112–152, 2002.

[11] D. Leijen. HMF: Simple type inference for first-class polymorphism. In *ICFP*, pages 283–294, 2008.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, 2nd edition, 1999.

[13] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[14] 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.

[15] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *TOPLAS*, 21(3):527–568, 1999.

[16] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *JFP*, 13(5):957–959, 2003.

[17] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI*, pages 229–243, 1996.

[18] M. Odersky and P. Wadler. Pizza into java: translating theory into practice. In *POPL*, pages 146–159, 1997.

[19] F. Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2):185–199, 1993.

[20] B. C. Pierce and D. N. Turner. Local type inference. In *POPL*, pages 252–265, 1998.

[21] R. Tate, J. Chen, and C. Hawblitzel. A framework for type inference with existential quantification. Technical report, `http://research.microsoft.com/pubs/78684/tr.pdf`, 2008.

[22] S. Wehr and P. Thiemann. On the decidability of subtyping with bounded existential types. In *APLAS*, pages 111–127, 2009.

[23] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98:111–156, 1998.

Appendix. Interface-Method Lookup in Detail

Here we describe in detail how we type check the common but surprisingly challenging process of looking up an interface method implementation. Suppose we have an instance of some class α implementing the interface `IList`, and we want to invoke the method `getCount` declared in `IList`. A typical implementation first loads the vtable for α from the instance. Then it must load α's interface table from the vtable. This table is an array of all the interfaces implemented by α. Each entry in the interface table corresponds to an interface implemented by α and consists of two pieces of information about that interface: the runtime type as well as the offset of the appropriate interface-method table from the beginning of α's vtable. So, the program has to go through each index of α's interface table until it finds an interface matching `IList`. It then adds the offset at that index to the vtable to get α's interface-method table for `IList`. From that we finally retrieve α's implementation of `getCount`. This is all accomplished by the assembly code in Figure 1 (for simplicity we assume that the important fields for this example are always at offset 0).

Now we wish to type check this assembly code. First, we start at the beginning of the block. There is an instance of `IList` at stack offset -4, corresponding to the first stack slot, so we give that slot the type $\text{HeapPtr}(\textsc{Ins}(\alpha))$, where α is an existentially quantified class variable. We use the variable α to refer to the exact class of this instance. We know α is a class, so we have the constraint α_c. Furthermore, we know the instance implements `IList`, so we also have the constraint $\alpha \ll \texttt{IList}$. Since we have a HeapPtr in a stack slot, we automatically open the name $\textsc{Ins}(\alpha)$ to the record type $\{\texttt{R} : \text{HeapPtr}(\text{V}\textsc{Table}(\alpha)), \dots\}$ (using the metainformation provided by the compiler). Thus, the first stack slot is given the following type:

$$\text{HeapRecPtr}(\textsc{Ins}(\alpha) : \{\texttt{R} : \text{HeapPtr}(\text{V}\textsc{Table}(\alpha)), \dots\} + 0)$$

This way we may both use the instance as an object in the heap and have access to the fields of its record. Now, instruction 1) simply copies this type into register `EAX`.

Instruction 2) replaces the type in `EAX` with the type of the first field of the instance's record: $\text{HeapPtr}(\text{V}\textsc{Table}(\alpha))$. Once again, we automatically open the name type to the following record type:

$$\{\texttt{R} : \text{HeapPtr}(\text{I}\textsc{Table}(\alpha)), \dots\}$$

$\text{I}\textsc{Table}(\alpha)$ is the name for α's interface table. Instruction 3) saves the vtable for later use by copying `EAX` to the third slot on the stack.

Instruction 4) replaces the type in `EAX` with the type of the first field of α's vtable: $\text{HeapPtr}(\text{I}\textsc{Table}(\alpha))$. Once again, we automatically open the name type $\text{I}\textsc{Table}(\alpha)$ to a record type. Because $\text{I}\textsc{Table}(\alpha)$ represents an array, this actually opens to the following existentially quantified extended-record type:

$$\exists \ell.\{\texttt{R} : \text{Int}(\ell)\} \circ \text{I}\textsc{TableEntry}(\alpha)_\ell$$

The existentially quantified variable ℓ (the length of the array) is automatically pulled into the environment. $\text{I}\textsc{TableEntry}(\alpha)$ is a fixed-length name representing the length-2-record type comprising each entry of the interface table, which we will examine in more detail later. Instruction 5) loads the first field of α's interface table. This field is the singleton integer type representing the length of the array. Comparing with this value will enable us to ensure that the assembly code is accessing the array with a valid index. Instruction 6) is an optimization which adds 4 to `EAX` so that `EAX` points to the first element of the interface table's array. We can handle this optimization since our HeapRecPtr and RecPtr types have an offset. Specifically, `EAX` will have the following type:

$$\text{HeapRecPtr}(\text{I}\textsc{Table}(\alpha) : \{\texttt{R} : \text{Int}(\ell)\} \circ \text{I}\textsc{TableEntry}(\alpha)_\ell + 4)$$

Note the offset $+4$, and that ℓ is now in the environment.

$$
\exists\ \begin{array}{c}\textbf{Variables}\\ \\ \alpha,\beta\\ i,\ell\end{array}\ :\ \begin{array}{c}\textbf{Constraints}\\ \alpha_{\mathrm{C}}\\ \beta_{\mathrm{I}}\\ \alpha\ll\texttt{IList}\\ \alpha\ll\beta\\ i<_{32+}\ell\end{array}\ .\
\begin{array}{c|lll}
\textbf{Registry} & \texttt{EAX} & \mapsto & \mathrm{HeapRecPtr}(\mathrm{ITABLE}(\alpha):\{\texttt{R}:\mathrm{Int}(\ell)\}\circ\mathrm{ITABLEENTRY}(\alpha)_\ell[\mathrm{last}:i\mapsto\beta]+4)\\
 & \texttt{EBX} & \mapsto & \mathrm{Int}(\ell)\\
 & \texttt{ECX} & \mapsto & \mathrm{Int}(i)\\
 & \texttt{EDX} & \mapsto & \mathrm{HeapRecPtr}(\mathrm{RUNTIME}(\beta):\{\dots\}+0)\\
\hline
\textbf{Stack} & 1^{\mathrm{st}} & \mapsto & \mathrm{HeapRecPtr}(\mathrm{INS}(\alpha):\{\texttt{R}:\mathrm{HeapPtr}(\mathrm{VTABLE}(\alpha)),\dots\}+0)\\
 & 2^{\mathrm{nd}} & \mapsto & \mathrm{HeapRecPtr}(\mathrm{RUNTIME}(\texttt{IList}):\{\dots\}+0)\\
 & 3^{\mathrm{rd}} & \mapsto & \mathrm{HeapRecPtr}(\mathrm{VTABLE}(\alpha):\{\texttt{R}:\mathrm{HeapPtr}(\mathrm{ITABLE}(\alpha)),\dots\}+0)\\
\hline
\textbf{CC} & \multicolumn{3}{l}{\mathrm{HeapRecPtr}(\mathrm{RUNTIME}(\texttt{IList}):\{\dots\}+0)\ \text{with}\ \mathrm{HeapRecPtr}(\mathrm{RUNTIME}(\beta):\{\dots\}+0)}
\end{array}
$$

Figure 2. An existentially quantified state type inferred during interface-method lookup

```
// stack offset -4 contains an instance of IList
// stack offset -8 contains runtime type of IList
   1)  mov EAX, [ESP-4]       // load instance into reg
   2)  mov EAX, [EAX]         // load vtable
   3)  mov [ESP-12], EAX      // store vtable on stack
   4)  mov EAX, [EAX]         // load interface table
   5)  mov EBX, [EAX]         // load table's length
   6)  addi EAX, #4           // move to head of array
   7)  movi ECX, #0           // start at index 0
L0:8)  cmp EBX, ECX           // compare length and index
   9)  jbe L1                 // break if length <= index
   10) mov EDX, [EAX+ECX*8]   // retrieve runtime type
   11) cmp EDX, [ESP-8]       // compare with IList
   12) je L2                  // break from loop if equal
   13) addi ECX, #1           // increment index
   14) jmp L0                 // continue looping
L1:15) throw exception        // not an instance of IList
L2:16) mov EDX, [EAX+ECX*8+4]// load offset from table
   17) add EDX, [ESP-12]      // add offset to vtable
   18) mov EAX, [EDX]         // load getCount
// stack offset -4 contains an instance of IList
// EAX contains getCount implementation for the instance
```

Figure 1. The assembly code for looking up an interface method

Instruction 7) loads the constant 0 into register ECX (which tracks the index into the interface table), so that ECX has the singleton integer type $\mathrm{Int}(0)$. The block starting at L0 iterates over each interface-table entry until it finds the one (if any) corresponding to IList. Notice that instruction 14) jumps back to L0 for the next iteration. Normally, we would proceed to type the instructions with ECX having type $\mathrm{Int}(0)$ until we get to instruction 14), at which point we would join that type with the current type and repeat the whole process. In the interest of saving time, we will simply replace the type of ECX with $\mathrm{Int}(i)$, where i is a fresh existentially quantified integer variable representing the index of the current iteration.

Instruction 8) compares EBX, which has type $\mathrm{Int}(\ell)$, with ECX, which has type $\mathrm{Int}(i)$. The state type is augmented to note that the condition code results from comparing $\mathrm{Int}(\ell)$ with $\mathrm{Int}(i)$. If $\ell \leq_{32+} i$ holds, instruction 9) jumps to instruction 15) to throw a runtime exception as there is no IList entry. Otherwise, we can only proceed to instruction 10) if $i <_{32+} \ell$ holds, so we add this constraint to the environment when type checking instruction 10).

Instruction 10) accesses the interface array. Since 4 was added to the address of the interface table earlier, EAX already points to the first element of the array. Instruction 10) accesses this array at offset $\texttt{ECX} * 8$. Since ECX has type $\mathrm{Int}(i)$, we know the value of this offset is $i * 8$. Since the name $\mathrm{ITABLEENTRY}(\alpha)$ always refers to a record with 2 fields, amounting to 8 bytes of data total, we can deduce that offset $i * 8$ is accessing the first field of the i^{th} index of the array. The constraint $i <_{32+} \ell$ is in the environment due to the earlier comparison, so we can ensure that this is a safe access into the array. In order to load the first field of the i^{th} index, we must open the name $\mathrm{ITABLEENTRY}(\alpha)$, which results in the following existentially quantified record type:

$$\exists\beta : \beta_{\mathrm{I}}, \alpha \ll \beta.\left\{\begin{array}{ll}\texttt{R}: & \mathrm{HeapPtr}(\mathrm{RUNTIME}(\beta))\\ \texttt{R}: & \mathrm{IMTableOffset}(\alpha,\beta)\end{array}\right\}$$

β represents the interface corresponding to that index, and the variable is pulled out into the environment. The constraint β_{I} indicates that β is an interface, and the constraint $\alpha \ll \beta$ indicates that α implements β. The first field has type $\mathrm{HeapPtr}(\mathrm{RUNTIME}(\beta))$, indicating it is the runtime type of β. Instruction 10) copies this type into EDX and opens the name $\mathrm{RUNTIME}(\beta)$ to a record type as usual. Furthermore, we tag the extended-record type for the interface table with $[\mathrm{last} : i \mapsto \beta]$, indicating that the last index accessed was i and β is the interface corresponding to that index. This is a feature of iTalX (not mentioned earlier) used specifically for any extended-record types whose fixed-length name opens to an existentially quantified type, the above case being the most important example. The purpose of this feature will be demonstrated later.

Instruction 11) compares the runtime type of IList with the runtime type in EDX corresponding to interface β. The state type is augmented to note that the condition code results from comparing these two types. Figure 2 shows the entire existentially quantified state type after instruction 11). Instruction 12) breaks from the loop if the two runtime types are the same. Otherwise, we proceed to instruction 13). No useful information is gained from knowing that the runtime types differ, so the state type stays the same except that the condition code is forgotten. Instruction 13) adds 1 to ECX, so ECX is given the type $\mathrm{Int}(i + 1)$. Instruction 14) jumps back to instruction 8), so we simply check that whether the current type is a subtype of the type before instruction 8), which is the case due to our earlier shortcut.

Instruction 12) could also break from the loop and proceed to instruction 16). This can only happen if the runtime type of IList matches the runtime type of β. From this, iTalX infers that these two class types are equal and merges them, essentially substituting all uses of β in the state type with IList. In particular, the tag we added earlier to the interface table's extended-record type becomes $[\mathrm{last} : i \mapsto \texttt{IList}]$, so that we know that index i corresponds to IList. Thus, when instruction 16) loads the second field of index i into EDX, the type of that field is $\mathrm{IMTableOffset}(\alpha, \texttt{IList})$. This type represents the integer offset which, when added to $\mathrm{HeapRecPtr}(\mathrm{VTABLE}(\alpha) : \{\dots\} + 0)$, results in $\mathrm{HeapPtr}(\mathrm{IMTABLE}(\alpha, \texttt{IList}))$, which is precisely the effect of instruction 17). $\mathrm{IMTABLE}(\alpha, \texttt{IList})$ is the name of the record containing α's implementations of IList's methods, all of which expect an instance of a subclass of α as the "this" pointer. Instruction 18) fetches the first field from this record, which our metainformation informs us corresponds to getCount. Thus, after all this effort, iTalX is finally able to type check a call to getCount using the original instance as the "this" pointer.