Typing a Multi-Language Intermediate Code

Article *in* ACM SIGPLAN Notices · December 2000

DOI: 10.1145/373243.360228 · Source: CiteSeer

2 authors:

Andrew D. Gordon
Microsoft
117 PUBLICATIONS **9,248** CITATIONS
SEE PROFILE

Don Syme
Microsoft
105 PUBLICATIONS **1,198** CITATIONS
SEE PROFILE

Typing a Multi-Language Intermediate Code

Andrew D. Gordon Don Syme

Microsoft Research

Abstract

The Microsoft .NET Framework is a new computing architecture designed to support a variety of distributed applications and web-based services. .NET software components are typically distributed in an object-oriented intermediate language, Microsoft IL, executed by the Microsoft Common Language Runtime. To allow convenient multi-language working, IL supports a wide variety of high-level language constructs, including class-based objects, inheritance, garbage collection, and a security mechanism based on type safe execution.

This paper precisely describes the type system for a substantial fragment of IL that includes several novel features: certain objects may be allocated either on the heap or on the stack; those on the stack may be boxed onto the heap, and those on the heap may be unboxed onto the stack; methods may receive arguments and return results via typed pointers, which can reference both the stack and the heap, including the interiors of objects on the heap. We present a formal semantics for the fragment. Our typing rules determine well-typed IL instruction sequences that can be assembled and executed. Of particular interest are rules to ensure no pointer into the stack outlives its target. Our main theorem asserts type safety, that well-typed programs in our IL fragment do not lead to untrapped execution errors.

Our main theorem does not directly apply to the product. Still, the formal system of this paper is an abstraction of informal and executable specifications we wrote for the full product during its development. Our informal specification became the basis of the product team's working specification of type-checking. The process of writing this specification, deploying the executable specification as a test oracle, and applying theorem proving techniques, helped us identify several security critical bugs during development.

1 Introduction

This paper describes typing and evaluation rules, and a type safety theorem, for a substantial fragment of the intermediate language (IL) executed by Microsoft's Common Language Runtime. The rules are valuable because they succinctly and precisely account for some unusual and subtle features of the type system.

Background: IL The Common Language Runtime is a new execution environment with a rich object-oriented class library through which software components written in diverse languages may interoperate. Using the Visual Studio .NET development environment, .NET components can be written in the new object-oriented language C$^{\#}$ [HW00], as well as Visual Basic, Visual C++, and the scripting languages VBScript and JScript. Furthermore, prototype .NET compilers exist for COBOL, Component Pascal, Eiffel, Haskell, Mercury, Oberon, Ocaml, and Standard ML.

Type-checking of .NET components implemented in IL has already proved useful for finding code generation bugs. Moreover, the .NET security model assumes type-safe behaviour; type-checking is therefore useful for handling untrusted components. Given these and other applications, the IL type system is worthy of formal specification.

Background: Executable Specifications This paper is one outcome of a research project to evaluate and develop formal specification techniques for describing and analyzing type-checkers in general. Specifically, we applied these techniques to the study of IL. We began by writing a detailed specification of type-checking method bodies. This was an informal document in the style of most language references. Eventually, this document was adopted by the product team as the basis of their detailed specification of type-checking. In parallel, following a methodology advocated by Syme [Sym98], we wrote formal specifications for various IL subsets suitable for comparative testing and formal proof. The executable part of these specifications is in a functional fragment of ML, the rest in higher order logic (HOL). We can compile and run the executable part as an IL type-checker. Since it is purely functional code, we may also interpret it as HOL and use it for theorem proving in DECLARE [Sym98]. In principle, this strategy allows the same source code to serve both as an oracle for testing actual implementations and as a model for formal validation. So far, we have built an ML type-checker for a largely complete subset of the IL type system, but have formally verified only a rather smaller fragment.

As is well known [Coh89], even formal proof cannot guarantee the absence of implementation defects, simply because one has to abstract from details of the environment when writing formal models. We found that developing a test suite that used our formal model as an oracle was an important way of making our model consistent with the runtime. Our suite included about 30,000 automatically generated tests. Our experience was that testing remains the only viable way of relating a specification to software of the complexity we were considering. One of our slogans: *if you specify, you must test.* Writing a formal specification without generating tests may be viable once a design has been frozen, but is simply not effective during the design of a new system. Eventually, we handed over our suite to the test team, who maintain it, and who have found bugs using it.

This Paper: An IL Fragment The main part of the paper concerns an IL fragment based on reference, value, and pointer types.

At its core, the fragment is a class-based object-oriented language with field update and simple imperative control structures. This core is comparable to the imperative object calculus [AC96, GHL99] and to various fragments of Java [DE97, IPW99]. An item of a reference type is a pointer to a heap-allocated object.

Moreover, our fragment includes value and pointer types:

- An item of a value type is a sequence of machine words representing the fields of the type. Value types support the compilation of C-style structs, for instance. Value types may be stack-allocated and passed by value. A `box` instruction turns a value type into a heap-allocated object by copying, and an `unbox` instruction performs the inverse coercion. Hence, when convenient, value types may be treated as ordinary heap-allocated objects.
- An item of pointer type is a machine address referring either to a heap-allocated object or to a variable in the call stack or to an interior field of one of these. The main purpose of pointer types is to allow methods to receive arguments or return results by reference.

We selected these types because they are new constructs not previously described by formal typing rules, and because their use needs to be carefully limited to avoid type loopholes. In particular, we must take care that stack pointers do not outlive their targets.

For the sake of clarity, our presentation of the semantics differs from the ML code in our executable specifications in two significant ways:

- First, we adopt the standard strategy of presenting the type system as logical inference rules. Such rules are succinct, but not directly executable; we found it better to write executable ML when we initially wrote our specifications in order to help with testing. Still, typing rules are better than code for presenting a type system and for manual proof.
- Second, we adopt a new, non-standard strategy of assuming that each method body has been parsed into a tree-structured applicative expression. Each expression consists of an IL instruction applied to the subexpressions that need to be evaluated to compute the instruction's arguments. This technique allows us to concentrate on specifying the typing conditions for each instruction, and to suppress the algorithmic details of how a type-checker would compute the types of the arguments to each instruction. These algorithmic details are important in any implementation, but they are largely irrelevant to specifying type safety.

Finally, in the spirit of writing specifications to support testing, our applicative expressions use the standard IL assembler syntax. Hence, any method body that is well-typed according to our typing rules can be assembled and tested on the running system.

In summary, the principal technical contributions made by this paper are the following:

- New typing and evaluation rules for value and pointer types, together with a type safety result, Theorem 1.
- The idea that the essence of a low-level intermediate language can be presented in an applicative notation.

Future Challenges: As we have discussed, this project is a successful demonstration of the value of writing executable, formal specifications during product development.

On the other hand, the main theorem of this paper does not apply to the full product; type safety bugs may well be discovered. An unfulfilled ambition of ours is to prove soundness of the typing rules for the full language through mechanized theorem-proving. So a future challenge is to further develop scalable and maintainable techniques for mechanized reasoning. A soundness proof for the whole of IL would be an impressive achievement. To apply theorem proving during product development, scalability and maintainability of proof scripts are important. Scripts should be scalable in the sense that human effort is roughly linear in the size of the specification (with a reasonable constant factor), or else proof construction cannot keep up with new features as they are added. Scripts should be maintainable in the sense that they are robust in the face of minor changes to the specification, or else proof construction cannot keep up with the inevitable revisions of the design.

In the meantime, another challenge is to develop systematic techniques for test case generation.

A third challenge is to integrate executable specifications, such as our ML type-checker, into the product itself. The .NET Framework, like other component models, itself contributes to this goal, in that its support for multi-language working would easily allow a critical component to be written in ML, say, even if the rest of the product is not.

The remainder of the paper proceeds as follows. Section 2 presents the typing and evaluation rules for our IL fragment, and states our main theorem. Section 3 explains a potentially useful liberalisation of the type system. Section 4 summarizes the omissions from our IL fragment. Section 5 discusses related work. Section 6 concludes.

Proofs omitted from this conference paper appear in a technical report [GS00].

2 A Formal Analysis of BIL, a Baby IL

This section makes the main technical contributions of the paper. We present a substantial fragment of IL that includes enough detail to allow a formal analysis of reference, value, and pointer types, but omits many features not related to these. We name this fragment Baby IL, or BIL for short.

Section 2.1 describes the type structure of BIL. In Section 2.2, we specify the instructions that may appear in method bodies of BIL, and explain their informal semantics. In Section 2.3, we specify a formal memory model for BIL, and a formal semantics for the evaluation of method bodies. In Section 2.4, we specify a formal type system for type-checking method bodies. Section 2.5 introduces conformance relations that express when intermediate states arising during evaluation are type-correct. Finally, Section 2.6 concludes this analysis by stating our Type Safety Theorem.

2.1 Type Structure and Class Hierarchy

All BIL methods run in an execution environment that contains a fixed set of classes. Each class specifies types for a set of field variables, and signatures for a set of methods. Each object belongs to a class. The memory occupied by each object consists of values for each field specified by its class. Methods are shared between all objects of a class (and possibly other classes). Objects of all classes may be stored boxed in a heap, addressed by heap references. Objects of certain classes—known as value classes—may additionally be stored unboxed in the stack or as fields embedded in other objects.

Formally, we assume three sets, *Class*, *Field*, and *Meth*, the sets of class, field, and method names, respectively, and a set $\mathit{ValueClass} \subseteq \mathit{Class}$ of value class names. We assume a distinguished class name `System.Object` such that `System.Object` $\notin \mathit{ValueClass}$.

Classes, Fields, Methods:

$c \in \mathit{Class}$	class name
$vc \in \mathit{ValueClass} \subseteq \mathit{Class}$	value class name
`System.Object` $\in \mathit{Class} - \mathit{ValueClass}$	root of hierarchy
$f \in \mathit{Field}$	field name
$\ell \in \mathit{Meth}$	method name

Types describe objects, the fields of objects, the arguments and results of methods, and the intermediate results arising during evaluation of method bodies.

Types:

$A, B \in \mathit{Type} ::=$	type
`void`	no bits
`int32`	32 bit signed integer
`class` c	boxed object
`value class` vc	unboxed object
$A\&$	pointer to A

The type `void` describes the absence of data, no bits; `void` is only used for the results of methods or parts of method bodies that return no actual result.

The type `int32` describes a 32 bit integer; BIL uses integers to represent predicates for conditionals and while-loops but includes no primitive arithmetic operations. (IL features a rich selection of numeric types and arithmetic operations.)

A *reference type* `class` c describes a pointer to a boxed object (heap-allocated, subject to garbage collection).

A *value type* `value class` vc describes an unboxed object—a sequence of words representing the fields of the value class vc, akin to a C struct. The associated reference type, `class` vc describes a pointer to a boxed object—a heap-allocated representation of the fields.

Finally, a *pointer type* $A\&$ describes a pointer to data of type A, which may be stored either in the heap or the stack.

To avoid dangling pointers—pointers that outlive their targets—our type system restricts pointers as follows. An important use for pointers in IL is to allow arguments and results to be passed by reference. The following are sufficient conditions to type-check this motivating usage while preventing dangling pointers. The following are not necessary conditions; we explain a useful and safe liberalisation in Section 3.

BIL Pointer Confinement Policy:

(1) No field may hold a pointer.
(2) No method may return a pointer.
(3) No pointer may be stored indirectly via another pointer.

(IL itself follows a slightly stricter policy that bans pointers to pointers altogether.) Each of the conditions prevents a way of creating a dangling pointer. If a field could hold a pointer, a method could store a pointer into its stack frame in an object boxed on the heap. If a method could return a pointer, a method could simply return a pointer into its stack frame. If a pointer could be stored indirectly, a method could store a pointer into its stack frame through a pointer to an object boxed on the heap or to an earlier stack frame. In each case, the pointer would outlive its target as soon as the method had returned.

The following predicate identifies types containing no pointers.

Whether a Type Contains No Pointer:

$\mathit{pointerFree}(A) \Leftrightarrow \neg(A = B\&$ for some $B)$

Next, a *method signature* $B\ \ell(A_1, \ldots, A_n)$ refers to a method named ℓ that expects a vector of arguments with types $A_1, \ldots, A_n$, and whose result has type B. No two methods in a given class may share the same signature, though they may share the same method name.

Method signature:

$\mathit{sig} \in \mathit{Sig} ::= B\ \ell(A_1, \ldots, A_n)$	method signature

We assume the execution environment organises classes into an inheritance hierarchy. We write c *inherits* c' to mean that c inherits from c'. We induce a *subtype relation*, $A <: B$, from the inheritance hierarchy. Our type system supports *subsumption*: if $A <: B$ an item of type A may be used in a context expecting an item of type B. The only non-trivial subtyping is between reference types. The subtype relation is the least to satisfy the following rules.

Subtype Relation: $A <: B$

(Sub Refl) $\dfrac{}{A <: A}$ (Sub Class) $\dfrac{c \mathit{\ inherits\ } c'}{\texttt{class}\ c <: \texttt{class}\ c'}$

We assume that the relation c *inherits* c' is transitive, and therefore so is the relation $A <: B$.

The IL assembler recognises a fairly standard notation for single inheritance that allows a class to inherit methods and fields from a single superclass. One might define the inheritance relation by formalizing such a syntax and type-checking rules. Instead, since our focus is type-checking the

BIL instruction set, it is easier and more concise to simply axiomatize the intended properties of the hierarchy. (Although the IL syntax disallows multiple inheritance, it happens that our axioms allow a class to inherit from two superclasses that are incomparable according to the inheritance relation.)

Formally, we assume there is an *execution environment* consisting of three components—a function $\mathit{fields}(c)$, a function $\mathit{methods}(c)$, and an inheritance relation c *inherits* c'—that satisfy the following axioms:

Execution Environment: $(\mathit{fields}, \mathit{methods}, \mathit{inherits})$

$\mathit{fields} \in \mathit{Class} \to (\mathit{Field} \overset{\text{fin}}{\to} \mathit{Type})$	fields of a class
$\mathit{methods} \in \mathit{Class} \to (\mathit{Sig} \overset{\text{fin}}{\to} \mathit{Body})$	methods of a class
$\mathit{inherits} \subseteq \mathit{Class} \times \mathit{Class}$	class hierarchy
$c\ \mathit{inherits}\ c$	(Hi Refl)
$c\ \mathit{inherits}\ c' \wedge c'\ \mathit{inherits}\ c'' \Rightarrow c\ \mathit{inherits}\ c''$	(Hi Trans)
$c\ \mathit{inherits}\ c' \wedge c'\ \mathit{inherits}\ c \Rightarrow c = c'$	(Hi Antisymm)
$c\ \mathit{inherits}$ `System.Object`	(Hi Root)
$c\ \mathit{inherits}\ d \wedge f \in \mathit{dom}(\mathit{fields}(d)) \Rightarrow f \in \mathit{dom}(\mathit{fields}(c)) \wedge \mathit{fields}(c)(f) = \mathit{fields}(d)(f)$	(Hi *fields*)
$c\ \mathit{inherits}\ d \Rightarrow \mathit{dom}(\mathit{methods}(d)) \subseteq \mathit{dom}(\mathit{methods}(c))$	(Hi *methods*)
$c\ \mathit{inherits}\ vc \Rightarrow c = vc$	(Hi Val)
$\mathit{pointerFree}(\mathit{fields}(c)(f))$	(Good *fields*)
$B\ \ell(A_1, \ldots, A_n) \in \mathit{dom}(\mathit{methods}(c)) \Rightarrow \mathit{pointerFree}(B)$	(Good *methods*)

For any class c, $\mathit{fields}(c) \in \mathit{Field} \overset{\text{fin}}{\to} \mathit{Type}$, the set of finite maps from field names to types. If $\mathit{fields}(c) = f_i \mapsto A_i{}^{i \in 1..n}$, the class c has exactly the set of fields named $f_1, \ldots, f_n$ with types $A_1, \ldots, A_n$, respectively.

(The notation $f_i \mapsto A_i{}^{i \in 1..n}$ exemplifies our notation for finite maps in general. We let $\mathit{dom}(f_i \mapsto A_i{}^{i \in 1..n}) = \{f_1, \ldots, f_n\}$. We assume that the f_i are distinct. Let $(f_i \mapsto A_i{}^{i \in 1..n})(f) = A_i$ if $f = f_i$ for some $i \in 1..n$, and otherwise be undefined.)

For any class c, $\mathit{methods}(c) \in \mathit{Sig} \overset{\text{fin}}{\to} \mathit{Body}$, the set of finite maps from method signatures to method bodies. We define the set *Body* of method bodies—instruction sequences—in the next section. If $\mathit{methods}(c) = \mathit{sig}_i \mapsto b_i{}^{i \in 1..n}$, the class c has exactly methods with signatures $\mathit{sig}_1, \ldots, \mathit{sig}_n$, implemented by the bodies $b_1, \ldots, b_n$, respectively.

A binary relation on classes, *inherits*, formalizes the inheritance hierarchy. Axioms (Hi Refl) and (Hi Trans) guarantee it is reflexive and transitive. (Hi Antisymm) asserts it is anti-symmetric, that is, there are no cycles in the hierarchy. According to (Hi Root), every class inherits from `System.Object`, the root of the hierarchy.

Suppose that c is a subclass of d, that is, c *inherits* d. By subsumption, an object of the subclass c may be used in a context expecting an object of the superclass d. Accordingly, (Hi *fields*) asserts that every field specified by d is also present in the subclass c. The axiom (Hi *methods*) asserts that every method signature implemented by d is also implemented by the subclass c, though not necessarily by the same method body.

In order to implement a method invocation on an object, we need to know the class of the object. In general, we cannot statically determine the class of an object from its type, since by subsumption it may in fact be a subclass of the class named in its type. Therefore, each boxed object is tagged in our formal memory model with the name of its class. On the other hand, for the sake of space efficiency, unboxed objects include no type information. Therefore, we must rely on statically determining the class of an unboxed object from its type. For this to be possible, axiom (Hi Val) prevents any other class from inheriting from a value class. So the actual class of any unboxed object is the same as the class named in its type.

Axioms (Good *fields*) and (Good *methods*) implement points (1) and (2) of the Pointer Confinement Policy.

We end this section by exemplifying how value and pointer types provide possibly more efficient alternatives to reference types for returning multiple results. Suppose there is a class `Point` $\in \mathit{ValueClass}$ such that $\mathit{fields}(\texttt{Point}) = \texttt{x} \mapsto \texttt{int32}, \texttt{y} \mapsto \texttt{int32}$, that is, a class with two integer fields. Here are three alternative signatures for returning a `Point` from a method named `mouse`:

- As a boxed object: `class Point mouse` ().
- As an unboxed object: `value class Point mouse` ().
- In a pre-allocated unboxed object passed by reference: `void mouse` (`value class Point&`).

2.2 Syntax of Method Bodies

BIL is a deterministic, single-threaded, imperative, class-based object-oriented language. For the sake of simplicity, we omit constructs for error or exception handling. This section specifies the instruction set as tree-structured applicative expressions, most of which represent an application of an instruction to a sequence of argument expressions. Since each applicative expression is in a postfix notation, it can also be read as a sequence of atomic instructions. We have chosen our syntax carefully so that, subject to very minor editing, this sequence of atomic instructions can be parsed by the IL assembler (as well as our own IL type-checker).

We express the syntax of our conditional and iteration constructs using assembler labels, ranged over by L.

A *method reference* $B\ c{::}\ell(A_1, \ldots, A_n)$ refers to the method with signature $B\ \ell(A_1, \ldots, A_n)$ in class c.

Inspired by FJ [IPW99], we assume for simplicity that each class has exactly one constructor, whose arguments are the initial values assumed by the fields of the new object. The *constructor reference* for a class c takes the form `void` c::`.ctor`$(A_1, \ldots, A_n)$. Constructors are only called to create a new object; `.ctor` $\notin \mathit{Meth}$.

Method and Constructor References:

L	assembler label
$M ::= B\ c{::}\ell(A_1, \ldots, A_n)$	method reference
$K ::= \texttt{void}\ c{::}\texttt{.ctor}(A_1, \ldots, A_n)$	constructor reference

Applicative Expressions for Method Bodies:

$i4$	32 bit signed integer
$a, b \in \mathit{Body} ::=$	method body
`ldc.i4` $i4$	load integer
a `brtrue` $L_1\ b_0$ `br` $L_2\ L_1{:}b_1\ L_2{:}$	conditional
$L_1{:}\ a$ `brfalse` $L_2\ b$ `br` $L_1\ L_2{:}$	while-loop
$a\ b$	sequencing
a `ldind`	load indirect

$a\,b$ `stind`	store indirect
`ldarga` j	load argument address
a `starg` j	store into argument
$a_1 \cdots a_n$ `newobj` K	create new object
$a_0\, a_1 \cdots a_n$ `callvirt` M	call on boxed object
$a_0\, a_1 \cdots a_n$ `call instance` M	call on unboxed object
a `ldflda` $A\ c{::}f$	load field address
$a\,b$ `stfld` $A\ c{::}f$	store into field
a `box` vc	copy value to heap
a `unbox` vc	fetch pointer to value

Conditionals and while-loops are not primitive instructions in IL, but it is worthwhile to make them primitive in BIL to allow a simple format for evaluation and typing rules. We have carefully chosen a syntax for these constructs by assembling suitable IL branch instructions and labels. We assume that the assembler labels in these expressions do not appear in any of their subexpressions. The result is a syntax that is a little cryptic but that does produce IL instruction sequences with the appropriate semantics. These abbreviations are more readable:

Abbreviations for Conditionals and While-Loops:

$a\ b_0\ b_1\ \mathit{cond} \triangleq a$ `brtrue` $L_1\ b_0$ `br` $L_2\ L_1{:}b_1\ L_2{:}$

$a\ b\ \mathit{while} \triangleq L_1{:}\ a$ `brfalse` $L_2\ b$ `br` $L_1\ L_2{:}$

The technique of representing assembly language in an applicative syntax works for this paper because it can express all the operations on reference, value, and pointer types. We express structured control flow like conditionals or while-loops in this style by treating an assembly of IL branch instructions as a primitive BIL instruction. Still, the technique may not scale well to express control flow such as arbitrary branching within a method or exception handling.

IL includes primitive instructions `ldfld` and `ldarg` to load the contents of an object field or an argument. Instead of taking these as primitives in BIL, we can derive them as follows:

Derived Instructions:

a `ldfld` $A\ c{::}f \triangleq a$ `ldflda` $A\ c{::}f$ `ldind`

a `ldarg` $j \triangleq a$ `ldarga` j `ldind`

2.3 Evaluating Method Bodies

The memory model consists of a heap of objects and a stack of method invocation frames, each of which is a vector of arguments. Our semantics abstracts away from the details of evaluation stacks or registers.

We assume a collection of heap references, p, q, pointing to boxed objects in the heap.

A *pointer* takes one of three forms. A pointer p refers to the boxed object at p. A pointer (i, j) refers to argument j of stack frame i. A pointer $\mathit{ptr}.f$ refers to field f of the object referred to by ptr.

A *result* is either void $\mathbf{0}$, an integer $i4$, a pointer ptr, or an unboxed object $f_i \mapsto u_i\ ^{i \in 1..n}$, a finite map consisting of a sequence of results $u_1, \ldots, u_n$ corresponding to the fields $f_1, \ldots, f_n$, respectively.

References, Pointers, Results:

p, q	heap reference	
$\mathit{ptr} ::=$	pointer	
p		pointer to boxed object p
(i, j)		pointer to argument j of frame i
$\mathit{ptr}.f$		pointer to field f of object at ptr
$u, v ::=$	result	
$\mathbf{0}$		void
$i4$		integer
ptr		pointer
$f_i \mapsto u_i\ ^{i \in 1..n}$		value: unboxed object

Next, we formalize our memory model. A *heap* is a finite map from references to boxed objects, each taking the form $c[f_i \mapsto u_i\ ^{i \in 1..n}]$, where c is the class of the object, and $f_i \mapsto u_i\ ^{i \in 1..n}$ is its unboxed form. A *frame*, fr, is a vector of arguments writen as $.\mathtt{args}(u_0, \ldots, u_n)$: u_0 is the self parameter; $u_1, \ldots, u_n$ are the computed arguments. A *stack*, s, is a list of frames $\mathit{fr}_1 \cdots \mathit{fr}_n$. Finally, a *store* is a heap paired with a stack.

Memory Model:

$o ::= c[f_i \mapsto u_i\ ^{i \in 1..n}]$	boxed object
$h ::= p_i \mapsto o_i\ ^{i \in 1..n}$	heap
$\mathit{fr} ::= .\mathtt{args}(u_0, \ldots, u_n)$	frame: vector of arguments
$s ::= \mathit{fr}_1 \cdots \mathit{fr}_n$	stack (grows left to right)
$\sigma ::= (h, s)$	store

The example heap $h = p \mapsto c[f_1 \mapsto 0, f_2 \mapsto (g \mapsto 1)]$ consists of a single boxed object $c[f_1 \mapsto 0, f_2 \mapsto (g \mapsto 1)]$ at heap reference p. The boxed object is of class c and consists of fields named f_1 and f_2. The first field contains the integer 0. The second field contains the unboxed object $g \mapsto 1$, which itself consists of a field named g containing the integer 1.

The example stack $s = .\mathtt{args}(p, p.f_2.g).\mathtt{args}(p, (1, 1))$ consists of two frames. The bottom of the stack is the frame $.\mathtt{args}(p, p.f_2.g)$, consisting of two arguments, a reference to the boxed object at p, and a pointer to field g of field f_2 of the same object. The top of the stack is the frame $.\mathtt{args}(p, (1, 1))$, consisting of two arguments, a reference to the boxed object at p, and the pointer $(1, 1)$, which refers to argument 1 of frame 1, that is, the pointer $p.f_2.g$.

We rely on two auxiliary partial functions for dereferencing and updating pointers in a store:

Auxiliary Functions for Lookup and Update:

$\mathit{lookup}(\sigma, \mathit{ptr})$	lookup ptr in store σ
$\mathit{update}(\sigma, \mathit{ptr}, v')$	update store σ at ptr with result v'

We explain the intended meaning of store lookup and update by example. Let store $\sigma = (h, s)$ where h and s are the heap and stack examples introduced above. Then $\mathit{lookup}(\sigma, (1, 0))$ is the reference p stored in argument 0 of frame 1, and $\mathit{lookup}(\sigma, p.f_2.g)$ is the integer 1 stored in field g of the unboxed object stored in field f_2 of the boxed object at p. The outcome of $\mathit{update}(\sigma, (2, 0), 1)$ is to update σ by replacing the reference p in argument 0 of frame 2 with 1. Similarly, the outcome of $\mathit{update}(\sigma, p.f_1.g, 0)$ is to update σ by replacing the integer 1 in field g of field f_1 of the boxed object at p with the integer 0.

A little functional programming suffices to define these two functions; we give the full definitions in the Appendix.

Our operational semantics of method bodies is a formal judgment $\sigma \vdash b \leadsto v \cdot \sigma'$ meaning that in an initial store

σ, the body b evaluates to the result v, leaving final store σ'. (A "judgment" is simply a predicate defined by a set of inference rules.)

Evaluation Judgment:

$\sigma \vdash b \leadsto v \cdot \sigma'$ given σ, body b returns v, leaving σ'

Our semantics takes the form of an interpreter. The rest of this section presents the formal rules for deriving evaluation judgments, interspersed with informal explanations.

Evaluation Rules for Control Flow:

(Eval `ldc`)

$$\frac{}{\sigma \vdash \texttt{ldc.i4}\ \mathit{i4} \leadsto \mathit{i4} \cdot \sigma}$$

(Eval Seq)

$$\frac{\sigma \vdash a \leadsto u \cdot \sigma' \quad \sigma' \vdash b \leadsto v \cdot \sigma''}{\sigma \vdash a\ b \leadsto v \cdot \sigma''}$$

(Eval Cond) (where $j = 0$ if $\mathit{i4} = 0$, otherwise $j = 1$)

$$\frac{\sigma \vdash a \leadsto \mathit{i4} \cdot \sigma' \quad \sigma' \vdash b_j \leadsto v \cdot \sigma''}{\sigma \vdash a\ b_0\ b_1\ \mathit{cond} \leadsto v \cdot \sigma''}$$

(Eval While 0)

$$\frac{\sigma \vdash a \leadsto 0 \cdot \sigma'}{\sigma \vdash a\ b\ \mathit{while} \leadsto \mathbf{0} \cdot \sigma'}$$

(Eval While 1) (where $\mathit{i4} \neq 0$)

$$\frac{\sigma \vdash a \leadsto \mathit{i4} \cdot \sigma' \quad \sigma' \vdash b \leadsto v \cdot \sigma'' \quad \sigma'' \vdash a\ b\ \mathit{while} \leadsto u \cdot \sigma'''}{\sigma \vdash a\ b\ \mathit{while} \leadsto u \cdot \sigma'''}$$

The expression `ldc.i4` $\mathit{i4}$ evaluates to the integer $\mathit{i4}$.

The expression $a\ b$ evaluates a, returning void (that is, nothing). The result of the whole expression is then the result of evaluating b.

The expression $a\ b_0\ b_1\ \mathit{cond}$ evaluates a to an integer $\mathit{i4}$. The result of the whole conditional is then the result of evaluating b_0 if $\mathit{i4} = 0$, and evaluating b_1 otherwise.

The expression $a\ b\ \mathit{while}$ evaluates a to an integer $\mathit{i4}$. If $\mathit{i4} = 0$ evaluation terminates, returning void. Otherwise, the body b is evaluated, returning void, and then evaluation of $a\ b\ \mathit{while}$ repeats.

Evaluation Rules for Pointer Types:

(Eval `ldind`)

$$\frac{\sigma \vdash a \leadsto \mathit{ptr} \cdot \sigma'}{\sigma \vdash a\ \texttt{ldind} \leadsto \mathit{lookup}(\sigma', \mathit{ptr}) \cdot \sigma'}$$

(Eval `stind`)

$$\frac{\sigma \vdash a \leadsto \mathit{ptr} \cdot \sigma' \quad \sigma' \vdash b \leadsto v \cdot \sigma''}{\sigma \vdash a\ b\ \texttt{stind} \leadsto \mathbf{0} \cdot \mathit{update}(\sigma'', \mathit{ptr}, v)}$$

The expression a`ldind` evaluates a to a pointer, and then returns the outcome of dereferencing the pointer.

The expression $a\ b$ `stind` evaluates a to a pointer, stores the result of evaluating b in the (heap or stack) location addressed by the pointer, and returns void.

Evaluation Rules for Arguments:

(Eval `ldarga`)

$$\frac{\sigma = (h, \mathit{fr}_1 \cdots \mathit{fr}_i)}{\sigma \vdash \texttt{ldarga}\ j \leadsto (i, j) \cdot \sigma}$$

(Eval `starg`)

$$\frac{\sigma \vdash a \leadsto u \cdot \sigma' \quad \sigma' = (h', \mathit{fr}_1 \cdots \mathit{fr}_i)}{\sigma \vdash a\ \texttt{starg}\ j \leadsto \mathbf{0} \cdot \mathit{update}(\sigma', (i, j), u)}$$

The expression `ldarga` j returns a pointer to argument j in the current stack frame.

The expression a `starg` i evaluates a, stores the result in argument i in the current stack frame, then returns void.

Evaluation Rules for Reference Types Only:

(Eval `newobj`) (where $K = \texttt{void}\ c\texttt{::.ctor}(A'_1, \ldots, A'_m)$)

$$\frac{\begin{array}{l} c \notin \mathit{ValueClass} \\ \mathit{fields}(c) = f_i \mapsto A_i\ ^{i \in 1..n} \quad \sigma_i \vdash a_i \leadsto v_i \cdot \sigma_{i+1} \quad \forall i \in 1..n \\ \sigma_{n+1} = (h, s) \quad p \notin \mathit{dom}(h) \quad h' = h, p \mapsto c[f_i \mapsto v_i\ ^{i \in 1..n}] \end{array}}{\sigma_1 \vdash a_1 \cdots a_n\ \texttt{newobj}\ K \leadsto p \cdot (h', s)}$$

(Eval `callvirt`) (where $M = B\ c\texttt{::}\ell(A_1, \ldots, A_n)$)

$$\frac{\begin{array}{l} \sigma_0 \vdash a_0 \leadsto p_0 \cdot (h_1, s_1) \quad h_1(p_0) = c'[f_i \mapsto u_i\ ^{i \in 1..m}] \\ (h_i, s_i) \vdash a_i \leadsto v_i \cdot (h_{i+1}, s_{i+1}) \quad \forall i \in 1..n \\ \mathit{methods}(c')(B\ \ell(A_1, \ldots, A_n)) = b \\ (h_{n+1}, s_{n+1}.\texttt{args}(p_0, v_1, \ldots, v_n)) \vdash b \leadsto v' \cdot (h', s'\ \mathit{fr}') \end{array}}{\sigma_0 \vdash a_0\ a_1 \cdots a_n\ \texttt{callvirt}\ M \leadsto v' \cdot (h', s')}$$

The expression $a_1 \cdots a_n$ `newobj` K, where K is the constructor for a class $c \notin \mathit{ValueClass}$, allocates a boxed object whose fields contain the results of evaluating $a_1, \ldots, a_n$, and returns the new reference.

The expression $a_0\ a_1 \cdots a_n$ `callvirt` M, where M refers to $B\ \ell(A_1, \ldots, A_n)$ in class c, evaluates a_0 to a reference to a boxed object of class c' (expected to inherit from c), locates the method body for $B\ \ell(A_1, \ldots, A_n)$ in class c', and returns the result of evaluating this method body in a new stack frame whose argument vector consists of the reference to the boxed object (the self pointer) together with the results of $a_1, \ldots, a_n$. The result of this evaluation is the store $(h', s'\ \mathit{fr}')$, where fr' is the final state of the new stack frame. Once evaluation of the method is complete, the stack is popped, to leave (h', s') as the final store.

Evaluation Rules for Reference and Value Types:

(Eval `ldflda`)

$$\frac{\sigma \vdash a \leadsto \mathit{ptr} \cdot \sigma'}{\sigma \vdash a\ \texttt{ldflda}\ A\ c\texttt{::}f \leadsto \mathit{ptr}.f \cdot \sigma'}$$

(Eval `stfld`)

$$\frac{\sigma \vdash a \leadsto \mathit{ptr} \cdot \sigma' \quad \sigma' \vdash b \leadsto v \cdot \sigma''}{\sigma \vdash a\ b\ \texttt{stfld}\ A\ c\texttt{::}f \leadsto \mathbf{0} \cdot \mathit{update}(\sigma'', \mathit{ptr}.f, v)}$$

The expression a `ldflda` $A\ c\texttt{::}f$ evaluates a to a pointer to a boxed or unboxed object, then returns a pointer to field f of this object.

The expression $a\ b$ `stfld` $A\ c\texttt{::}f$ evaluates a to a pointer to a boxed or unboxed object, updates its field f with the result of evaluating b, and returns void.

Evaluation Rules for Value Types Only:

(Eval `newobj`) (where $K = \texttt{void}\ vc\texttt{::.ctor}(A'_1, \ldots, A'_m)$)

$$\frac{\mathit{fields}(vc) = f_i \mapsto A_i\ ^{i \in 1..n} \quad \sigma_i \vdash a_i \leadsto v_i \cdot \sigma_{i+1} \quad \forall i \in 1..n}{\sigma_1 \vdash a_1 \cdots a_n\ \texttt{newobj}\ K \leadsto (f_i \mapsto v_i\ ^{i \in 1..n}) \cdot \sigma_{n+1}}$$

(Eval `call`) (where $M = B\ vc\texttt{::}\ell(A_1, \ldots, A_n)$)

$$\frac{\begin{array}{l} \sigma_0 \vdash a_0 \leadsto \mathit{ptr} \cdot (h_1, s_1) \\ (h_i, s_i) \vdash a_i \leadsto v_i \cdot (h_{i+1}, s_{i+1}) \quad \forall i \in 1..n \\ \mathit{methods}(vc)(B\ \ell(A_1, \ldots, A_n)) = b \\ (h_{n+1}, s_{n+1}.\texttt{args}(\mathit{ptr}, v_1, \ldots, v_n)) \vdash b \leadsto v' \cdot (h', s'\ \mathit{fr}') \end{array}}{\sigma_0 \vdash a_0\ a_1 \cdots a_n\ \texttt{call instance}\ M \leadsto v' \cdot (h', s')}$$

(Eval $\texttt{box}$) (where $p \notin dom(h')$)

$$\frac{\sigma \vdash a \rightsquigarrow ptr \cdot (h', s') \quad lookup((h', s'), ptr) = f_i \mapsto v_i \;{}^{i \in 1..n}}{\sigma \vdash a \; \texttt{box} \; vc \rightsquigarrow p \cdot ((h', p \mapsto vc[f_i \mapsto v_i \;{}^{i \in 1..n}]), s)}$$

(Eval $\texttt{unbox}$)

$$\frac{\sigma \vdash a \rightsquigarrow p \cdot \sigma'}{\sigma \vdash a \; \texttt{unbox} \; vc \rightsquigarrow p \cdot \sigma'}$$

The expression $a_1 \cdots a_n$ $\texttt{newobj}$ K, where K is the constructor for a value class vc, returns an unboxed object whose fields contain the results of evaluating $a_1, \ldots, a_n$.

The expression $a_0 \, a_1 \cdots a_n$ $\texttt{call instance}$ M where M refers to the signature $B \, \ell(A_1, \ldots, A_n)$ in value class vc, evaluates a_0 to a pointer to an unboxed object (expected to be of class vc), locates the method body for $B \, \ell(A_1, \ldots, A_n)$ in class vc, and returns the result of evaluating this method body in a new stack frame whose argument vector consists of the pointer to the unboxed object (the self pointer) together with the results of $a_1, \ldots, a_n$.

The expression a $\texttt{box}$ c evaluates a to a pointer to an unboxed object, allocates it in boxed form in the heap, and returns the fresh heap reference.

The expression a $\texttt{unbox}$ c evaluates a to a heap reference to a boxed object, and returns this reference as its result.

2.4 Typing Method Bodies

This section describes a type system for method bodies such that evaluation of well-typed method bodies cannot lead to an execution error. What is perhaps most interesting here is the implementation of the Pointer Confinement Policy of Section 2.1.

Let a *type frame*, *Fr*, take the form $.\texttt{args}(A_0, \ldots, A_n)$, a description of the types of the results in the current (top) stack frame. Our typing judgment, $Fr \vdash b : B$, means if the current stack frame matches Fr, the body b evaluates to a result of type B.

Type Frames and Typing Judgment:

$Fr ::= .\texttt{args}(A_0, \ldots, A_n)$	frame: types of arguments
$Fr \vdash b : B$	given Fr, body b returns type B

We make the additional assumption about our execution environment that every method body (b below) conforms to its signature:

Additional Assumptions:

$c \notin ValueClass \wedge$ (Ref *methods*)
$methods(c)(B \, \ell(A_1, \ldots, A_n)) = b \;\Rightarrow$
$.\texttt{args}(\texttt{class}\ c, A_1, \ldots, A_n) \vdash b : B$

$vc \in ValueClass \wedge$ (Val *methods*)
$methods(vc)(B \, \ell(A_1, \ldots, A_n)) = b \;\Rightarrow$
$.\texttt{args}(\texttt{value class}\ vc\&, A_1, \ldots, A_n) \vdash b : B$

Next, we give typing rules to define $Fr \vdash b : B$.

Typing Rule for Subsumption:

(Body Subsum)

$$\frac{Fr \vdash b : B \quad B <: B'}{Fr \vdash b : B'}$$

This standard rule allows an expression of a subtype B to be used in a context expecting a supertype B'.

Typing Rules for Control Flow:

(Body $\texttt{ldc}$)

$$\frac{}{Fr \vdash \texttt{ldc.i4}\ i4 : \texttt{int32}}$$

(Body Seq)

$$\frac{Fr \vdash a : \texttt{void} \quad Fr \vdash b : B}{Fr \vdash a \, b : B}$$

(Body Cond)

$$\frac{Fr \vdash a : \texttt{int32} \quad Fr \vdash b_0 : B \quad Fr \vdash b_1 : B}{Fr \vdash a \, b_0 \, b_1 \; cond : B}$$

(Body While)

$$\frac{Fr \vdash a : \texttt{int32} \quad Fr \vdash b : \texttt{void}}{Fr \vdash a \, b \; while : \texttt{void}}$$

The rule (Body Seq) uses the type $\texttt{void}$ to guarantee that the first part of a sequential composition returns no results.

The rules (Body Cond) and (Body While) use the type $\texttt{int32}$ to guarantee the predicate expression a returns an integer.

Typing Rules for Pointer Types:

(Body $\texttt{ldind}$)

$$\frac{Fr \vdash a : A\&}{Fr \vdash a \; \texttt{ldind} : A}$$

(Body $\texttt{stind}$) (where $pointerFree(A)$)

$$\frac{Fr \vdash a_1 : A\& \quad Fr \vdash a_2 : A}{Fr \vdash a_1 \, a_2 \; \texttt{stind} : \texttt{void}}$$

The rule (Body $\texttt{stind}$) implements rule (3) of the Pointer Confinement Policy; without the condition $pointerFree(A)$, $\texttt{stind}$ could copy a pointer to the current stack frame further back the stack.

Typing Rules for Arguments:

(Body $\texttt{ldarga}$)

$$\frac{j \in 0..n}{.\texttt{args}(A_0, \ldots, A_n) \vdash \texttt{ldarga}\ j : A_j\&}$$

(Body $\texttt{starg}$)

$$\frac{.\texttt{args}(A_0, \ldots, A_n) \vdash a : A_j \quad j \in 0..n}{.\texttt{args}(A_0, \ldots, A_n) \vdash a \; \texttt{starg}\ j : \texttt{void}}$$

These rules check that the argument index j exists. Since $\texttt{starg}$ only writes within the current frame, we can safely allow A_j to be a pointer.

Typing Rules for Reference Types:

(Ref $\texttt{newobj}$) (where $K = \texttt{void}\ c::\texttt{.ctor}(A_1, \ldots, A_n)$ and $fields(c) = f_i \mapsto A_i \;{}^{i \in 1..n}$)

$$\frac{Fr \vdash a_i : A_i \quad \forall i \in 1..n \quad c \notin ValueClass}{Fr \vdash a_1 \cdots a_n \; \texttt{newobj}\ K : \texttt{class}\ c}$$

(Ref $\texttt{callvirt}$) (where $B \, \ell(A_1, \ldots, A_n) \in dom(methods(c))$)

$$\frac{Fr \vdash a_0 : \texttt{class}\ c \quad Fr \vdash a_i : A_i \quad \forall i \in 1..n}{Fr \vdash a_0 \, a_1 \cdots a_n \; \texttt{callvirt}\ B\ c::\ell(A_1, \ldots, A_n) : B}$$

(Ref $\texttt{ldflda}$) (where $fields(c) = f_i \mapsto A_i \;{}^{i \in 1..n}$)

$$\frac{Fr \vdash a : \texttt{class}\ c \quad j \in 1..n}{Fr \vdash a \; \texttt{ldflda}\ A_j\ c::f_j : A_j\&}$$

(Ref $\texttt{stfld}$) (where $fields(c) = f_i \mapsto A_i \;{}^{i \in 1..n}$ and $pointerFree(A_j)$)

$$\frac{Fr \vdash a : \texttt{class}\ c \quad Fr \vdash b : A_j \quad j \in 1..n}{Fr \vdash a \, b \; \texttt{stfld}\ A_j\ c::f_j : \texttt{void}}$$

These are fairly standard rules for operations on boxed objects. Recall that the axiom (Good *fields*) guarantees every field is pointer-free. So the *pointerFree*(−) condition on the rule (Ref `stfld`) is redundant. Still, it is not redundant in a variation of our type system considered in Section 3, that allows value classes to include pointers.

Typing Rules for Value Types:

(Val `newobj`) (where $K = \texttt{void}\ vc{::}\texttt{.ctor}(A_1, \ldots, A_n)$ and $\mathit{fields}(vc) = f_i \mapsto A_i\ ^{i \in 1..n}$)

$$\frac{Fr \vdash a_i : A_i \quad \forall i \in 1..n}{Fr \vdash a_1 \cdots a_n\ \texttt{newobj}\ K : \texttt{value class}\ vc}$$

(Val `call`) (where $B\ \ell(A_1, \ldots, A_n) \in \mathit{dom}(\mathit{methods}(vc))$)

$$\frac{Fr \vdash a_0 : \texttt{value class}\ vc\& \quad Fr \vdash a_i : A_i \quad \forall i \in 1..n}{Fr \vdash a_0\ a_1 \cdots a_n\ \texttt{call instance}\ B\ vc{::}\ell(A_1, \ldots, A_n) : B}$$

(Val `ldflda`) (where $\mathit{fields}(vc) = f_i \mapsto A_i\ ^{i \in 1..n}$)

$$\frac{Fr \vdash a : \texttt{value class}\ vc\& \quad j \in 1..n}{Fr \vdash a\ \texttt{ldflda}\ A_j\ vc{::}f_j : A_j\&}$$

(Val `stfld`) (where $\mathit{fields}(vc) = f_i \mapsto A_i\ ^{i \in 1..n}$ and $\mathit{pointerFree}(A_j)$)

$$\frac{Fr \vdash a : \texttt{value class}\ vc\& \quad Fr \vdash b : A_j \quad j \in 1..n}{Fr \vdash a\ b\ \texttt{stfld}\ A_j\ vc{::}f_j : \texttt{void}}$$

(Val `box`) (where $\mathit{pointerFree}(\texttt{value class}\ vc)$)

$$\frac{Fr \vdash a : \texttt{value class}\ vc\&}{Fr \vdash a\ \texttt{box}\ vc : \texttt{class}\ vc}$$

(Val `unbox`)

$$\frac{Fr \vdash a : \texttt{class}\ vc}{Fr \vdash a\ \texttt{unbox}\ vc : \texttt{value class}\ vc\&}$$

These are similar to the typing rules for operations on boxed objects, except we refer to the object via a pointer type instead of a reference type. Like (Ref `stfld`), the rules (Val `stfld`) and (Val `box`) bear *pointerFree*(−) conditions that are redundant in the current system, but not in the system of Section 3.

2.5 Typing the Memory Model

In this section, we present predicates, known as conformance judgments, that confer types on our memory model. In the next, we show that these predicates are invariants of computation, that is, are preserved by method evaluation.

We begin by introducing types for the components of our memory model. A *heap type* $p_i \mapsto c_i\ ^{i \in 1..n}$ determines the actual class of each boxed object. A *stack type* $Fr_1 \cdots Fr_n$ determines frame types for each frame in the stack. A *store type* $\Sigma = (H, S)$ determines a heap type H and stack type S.

Heap, Stack, and Store Types:

$H ::= p_i \mapsto c_i\ ^{i \in 1..n}$	heap type
$S ::= Fr_1 \cdots Fr_n$	stack type
$\Sigma ::= (H, S)$	store type

Our first conformance judgment, $\Sigma \models u : A$, means that in a store matching the store type Σ, the result u is well-formed and has type A. We define what it means for a store to match a store type through other conformance judgments, defined later.

Conformance Judgment for Results (Including Pointers):

$\Sigma \models u : A$	in Σ, result u has type A

Conformance Rules for References and Pointers:

(Res Ref)

$$\frac{H(p) = c \quad c\ \mathit{inherits}\ c'}{(H, S) \models p : \texttt{class}\ c'}$$

(Ptr Ref)

$$\frac{H(p) = vc}{(H, S) \models p : \texttt{value class}\ vc\&}$$

(Ptr Arg)

$$\frac{i \in 1..m \quad Fr_i = \texttt{.args}(A_0, \ldots, A_n) \quad j \in 0..n}{(H, Fr_1 \cdots Fr_m) \models (i, j) : A_j\&}$$

(Ptr Field) (where $A = \texttt{class}\ c$ or $A = \texttt{value class}\ c\&$)

$$\frac{\Sigma \models \mathit{ptr} : A \quad \mathit{fields}(c) = f_i \mapsto A_i\ ^{i \in 1..n} \quad j \in 1..n}{\Sigma \models \mathit{ptr}.f_j : A_j\&}$$

The rule (Res Ref) assigns a reference type $\texttt{class}\ c'$ to a heap reference p, so long as c' is a superclass of the actual class of the object referred to by p.

The rule (Ptr Ref) assigns a pointer type to a heap reference p that refers to a value that is boxed on the heap.

These two rules can assign both a reference type and a pointer type to a heap reference to a value class. If $H(p) = vc$, then we have $(H, S) \models p : \texttt{class}\ c$ by (Res Ref), but also $(H, S) \models p : \texttt{value class}\ c\&$ by (Ptr Ref). We need (Res Ref) to type references constructed by the `box` instruction. We need (Ptr Ref) to type pointers constructed by the `unbox` instruction.

The rule (Ptr Arg) assigns a pointer type to a stack pointer (i, j) that refers to argument j of frame i.

The rule (Ptr Field) assigns a pointer type to a pointer referring to the field f_j of the object referred to by *ptr*. The base pointer *ptr* may either be of type $\texttt{class}\ c$ or $\texttt{value class}\ c\&$. The first case is needed for a pointer to a field of a heap object that is not in a value class. The second case is needed for a pointer to a field of a heap or stack object in a value class.

Conformance Rules for Other Results:

(Res Void)

$$\frac{}{\Sigma \models \mathbf{0} : \texttt{void}}$$

(Res Int)

$$\frac{}{\Sigma \models i4 : \texttt{int32}}$$

(Res Value)

$$\frac{\mathit{fields}(vc) = f_i \mapsto A_i\ ^{i \in 1..n} \quad \Sigma \models v_i : A_i \quad \forall i \in 1..n}{\Sigma \models f_i \mapsto v_i\ ^{i \in 1..n} : \texttt{value class}\ vc}$$

The rules (Res Void) and (Res Int) assign the `void` and `int32` types to void and integer values, respectively.

The rule (Res Value) assigns a value type $\texttt{value class}\ vc$ to a value. By axiom (Hi Val), the inheritance hierarchy is flat for value types. So (Res Value), unlike (Res Ref), does not allow vc to be a proper superclass of the actual class of the value.

Other Conformance Judgments:

$H \models o : c$	in H, object o has class c
$H \models h$	heap h conforms to H

$\Sigma \models fr : Fr$ — frame fr conforms to Fr
$\Sigma \models \sigma$ — store σ conforms to Σ

Conformance Rule for Objects:

(Con Object) (where $fields(c) = f_i \mapsto A_i \,^{i \in 1..n}$)

$$\frac{(H, \varnothing) \models v_i : A_i \quad \forall i \in 1..n}{H \models c[f_i \mapsto v_i \,^{i \in 1..n}] : c}$$

This rule defines when a heap object $c[f_i \mapsto v_i \,^{i \in 1..n}]$ is well-typed. The preconditions $(H, \varnothing) \vdash v_i : A_i$ require that the fields v_i be typed with an empty stack type. It follows that no field v_i contains a stack pointer, since the rule (Ptr Arg) for typing stack pointers assumes a non-empty stack type.

Conformance Rule for Heaps:

(Con Heap) (where $H = p_i \mapsto c_i \,^{i \in 1..n}$)

$$\frac{H \models o_i : c_i \quad \forall i \in 1..n}{H \models p_i \mapsto o_i \,^{i \in 1..n}}$$

This rule defines when a heap $p_i \mapsto o_i \,^{i \in 1..n}$ conforms to the heap type $p_i \mapsto c_i \,^{i \in 1..n}$. The heap type contains the actual class c_i of each object o_i.

Conformance Rule for Frames:

(Con Frame)

$$\frac{\Sigma \models u_i : A_i \quad \forall i \in 0..n}{\Sigma \models \texttt{.args}(u_0, \ldots, u_n) : \texttt{.args}(A_0, \ldots, A_n)}$$

This rule defines when a frame conforms to a frame type.

Conformance Rule for Stores:

(Con Store)

$$\frac{H \models h \quad (H, Fr_1 \cdots Fr_i) \models fr_i : Fr_i \quad \forall i \in 1..n}{(H, Fr_1 \cdots Fr_n) \models (h, fr_1 \cdots fr_n)}$$

This rule defines when a store $(H, Fr_1 \cdots Fr_n)$ conforms to a store type $(h, fr_1 \cdots fr_n)$. It asks that the heap h conform to the heap type H, and that each stack frame fr_i conform to the corresponding frame type Fr_i, but after removing from the store type any higher—shorter lived—stack frames. Hence, there may be pointers from a higher to a lower stack frame, but not the other way round.

2.6 Evaluation Respects Typing

We use standard proof techniques to show the consistency of the BIL evaluation semantics with its type system. The following is the main type safety result of the paper. If a program satisfies the restrictions on type structure imposed in Section 2.1 and the typing rules for method bodies in Section 2.4 then its evaluation according to the rules in Section 2.3 can lead only to conformant intermediate states as defined in Section 2.5. Let $H \leq H'$ mean that $dom(H) \subseteq dom(H')$ and $H(p) = H'(p)$ for all $p \in dom(H)$.

Theorem 1 *If $(H, S\ Fr) \models \sigma$ and $Fr \vdash b : B$ and $\sigma \vdash b \rightsquigarrow v \cdot \sigma^\dagger$ then there exists a heap type $H^\dagger$ such that $H \leq H^\dagger$ and $(H^\dagger, S\ Fr) \models v : B$ and $(H^\dagger, S\ Fr) \models \sigma^\dagger$.*

Proof By induction on the derivation of $\sigma \vdash b \rightsquigarrow v \cdot \sigma^\dagger$. We omit the details. See the Appendix for the main lemmas about the type system needed in the proof. □

As usual, such a theorem is vacuous if there is no $\sigma^\dagger$ such that $\sigma \vdash b \rightsquigarrow v \cdot \sigma^\dagger$ holds, which happens either because the computation would diverge, or because it gets stuck (if there is no applicable evaluation rule). Stuck states correspond to execution errors, such as calling a non-existent method, or attempting to de-reference an integer or a dangling pointer. As discussed by Abadi and Cardelli [AC96], we conjecture it would be straightforward to adapt the proof of Theorem 1 to show that no stuck state is reachable.

3 Variation: Allowing Pointers in Fields of Value Classes

To avoid dangling pointers, the IL type system prevents the fields of all objects, whether boxed on the heap or unboxed on the stack, from holding pointers. In fact, as pointed out by Fergus Henderson, a more liberal type system that allows unboxed objects to contain pointers is useful for compiling nested functions.

When compiling a language with nested functions (for example, Pascal or Ada), each invocation of a nested function needs access to the activation records (that is, the arguments and local variables) of the lexically enclosing functions. A standard technique is to pass the function a display [ASU86], an array of pointers to these activation records. One strategy is to implement an activation record (containing those arguments and local variables referred to by nested functions) as a value class on the stack, and to implement the display by pointers to the value classes representing the activation records. Since arguments may be passed by reference, this scheme works only if we allow value classes to hold pointers. Otherwise, we need to pay the cost of boxing these activation records on the heap.

If we allow fields of value classes to hold pointers, the following more liberal policy still avoids dangling pointers.

A More Liberal Pointer Confinement Policy:

(1) No field of a boxed object may hold a pointer.
(2) No method may return a result containing a pointer.
(3) No result containing a pointer may be stored indirectly via another pointer.

Though this policy helps compile nested functions, we lose the possibly useful fact that every value class may be boxed, and hence treated as a subtype of `class System.Object`.

To formalize this policy, we amend BIL as follows.

- Change the definition of $pointerFree(A)$ to hold if and only if (1) A is not itself a pointer type, and (2) if A is a value class then the type B of each field satisfies $pointerFree(B)$. (The only change is the insertion of clause (2).)
- Change axiom (Good *fields*) to read: $c \notin ValueClass \Rightarrow pointerFree(fields(c)(f))$. (The only change is the insertion of the $c \notin ValueClass$ precondition.)

To see the effect of these changes, recall there are four typing rules that mention the $pointerFree(-)$ predicate: (Ref `stfld`), (Body `stind`), (Val `stfld`), and (Val `box`).

Previously, any value could be stored via (Body `stind`), and the pointer-free conditions on the other three rules were redundant. Now, these rules prevent the export of values containing pointers to the heap or further back the stack. Now, (Ref `stfld`) prevents a pointer being stored into a boxed value class with a pointer field. In fact, no such boxed value classes can even be allocated, given the *pointerFree*(−) condition on (Val `box`).

Our proof of Theorem 1, outlined in the Appendix, is in fact for this more liberal system. Type safety for the original system is a corollary of type safety for this more liberal system, since any method body typed by the original system remains typable.

Implementation of the new scheme remains future work.

4 IL Features Omitted From BIL

To give a flavour of the full intermediate language, we briefly enumerate the main features omitted from BIL. The IL Assembly Programmer's Reference Manual [Mic00] contains a complete informal description of IL.

We omit all discussion of IL metadata, such as how classes, static data and method headers are described. We omit any discussion of the on-disk format, the specification of linkage information, and assemblies, the unit of software deployment.

Our object model omits null objects, global fields and methods, static fields and methods, non-virtual methods, single dimensional and multidimensional covariant arrays, and object interfaces. Our instruction set omits local variables, arithmetic instructions, arbitrary branching, jumping, and tail calls. Tail calls require care, because the type system must prevent pointers to the current stack frame being passed as arguments. The current IL policy is to prevent the passing of any pointers via a tail call.

We omit delegates (that is, built-in support for anonymous method invocation), typed references (that is, a pointer packaged with its type, required for Visual Basic), attributes, native code calling conventions, interoperability with COM, remoting (object distribution) and multithreading. We also omit exception handling, a fairly elaborate model that permits a unified view of exceptions in C++, C$^{\#}$, and other high-level languages.

5 Related Work

The principle of formalizing type-checking via logical inference rules is a long-standing topic in the study of progamming languages [Car97]. Formal typing rules have been developed for several high-level languages, including SML [MTHM97], Haskell [PW92], and for subsets of Java [DE97, IPW99]. Formal typing rules have also been developed for several low-level languages, including TAL [MWCG99] and for subsets of the JVM [SA98, Qia99, Yel99, FM00]. The properties established by proof-carrying code [Nec97] can be viewed as typing derivations for native code. The idea of formalizing a type system via an executable type-checker has recently been advocated for Haskell [Jon99]. Our use of an executable specification as an oracle is an instance of the standard software engineering principle of multi-version prototyping. Proofs of soundness of several programming language type systems have been partially mechanised in theorem provers [Van96, Nor98, Sym99, vN99].

Several compilers, such as GHC [PHH^{+}93], TIL [TMC^{+}96], FLINT [Sha97], and MARMOT [FKR^{+}00], use a typed intermediate language internally. One [MWCG99] in particular translates all the way from System F, a polymorphic λ-calculus, down to a typed assembly language, TAL. The idea of writing a type-checker for a textual assembly format (like our type-checker for IL) appears in connection with TAL: the TALx86 type-checker accepts input in a typed form of the IA32 assembly language that can also be processed by the standard MASM assembler.

Reference types for heap-allocated data structures akin to the reference types of the type system of Section 2 appear in all of these intermediate languages. What is new about our type system is its inclusion of value and pointer types.

- Value types describe the unboxed stack-allocated form of a class. The `box` and `unbox` instructions coerce between stack and heap forms of a class. Types for boxed and unboxed non-strict data structures [PL91] and automatic type-based coercions between boxed and unboxed forms [Ler92] have been studied previously. Other approaches include region analysis [TT97] and escape analysis [PG92]. Still, the idea and formalization of types to differentiate between unboxed and boxed forms of class-based objects appears to be new.

- Pointer types describe pointers to either stack or heap allocated items. A risk with a stack pointer is that it may dangle, if its lifetime exceeds the lifetime of its target. The stack-based form of TAL [MCGW98] includes a type constructor for describing pointers into the stack; the parameter to the type constructor is a stack type that ensures the target is still live when the pointer is dereferenced. Instead, the Pointer Confinement Policy of Section 2 avoids dangling pointers via various syntactic restrictions. IL's pointer types are easier to integrate with high-level languages like Visual Basic with rather simple type systems than a more sophisticated solution using stack types, as found in TAL.

6 Conclusions

One of the innovations in Microsoft's Common Language Runtime is support for typed stack pointers, for passing arguments and results by reference, for example. We presented formal typing rules and a type safety result for a substantial fragment of the Common Language Runtime intermediate language. Our treatment of value types and pointer types appears to be new. These rules were devised through our writing informal and executable specifications of the full intermediate language. This effort clarified the design and helped find bugs, but further research is needed on machine support for formal reasoning and on test case generation. We exploited our formal model to validate a liberalisation of the IL policy that allows object fields to contain stack pointers.

Acknowledgements This paper is one outcome of a collaborative project between Microsoft Research Cambridge and the .NET Common Language Runtime product team. We thank the Common Language Runtime team, and especially S. Dasan, J. Forbes, J. Miller, and V. Morrison, for their help during our collaboration. We thank our research interns P. Hankin, J. Matthews, and C. Pitcher for their work on the project. M. Abadi, C.A.R. Hoare, A. Jeffrey, and C. Skalka suggested improvements to a draft of this paper.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Car97] L. Cardelli. Type systems. In A.B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

[Coh89] A. Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.

[DE97] S. Drossopoulou and S. Eisenbach. Java is type safe—probably. In *Proceedings ECOOP'97*, June 1997.

[FKR$^+$00] R. Fitzgerald, T.B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. *Software: Practice and Experience*, 30(3), 2000.

[FM00] S. Freund and J.C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 2000. To appear.

[GHL99] A.D. Gordon, P.D. Hankin, and S.B. Lassen. Compilation and equivalence of imperative objects. *Journal of Functional Programming*, 9(4):373–426, 1999.

[GS00] A.D. Gordon and D. Syme. Typing a multi-language intermediate code. Technical Report MSR–TR–2000–106, Microsoft Research, 2000.

[HW00] A. Hejlsberg and S. Wiltamuth. C# Language Reference. Available at *http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp*, 2000.

[IPW99] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999.

[Jon99] M.P. Jones. Typing Haskell in Haskell. In *Proceedings Haskell Workshop, Paris*, 1999. Available at *http://www.cse.ogi.edu/~mpj/thih*.

[Ler92] X. Leroy. Unboxed objects and polymorphic typing. In *19th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 177–188. ACM Press, 1992.

[MCGW98] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52. Springer Verlag, 1998.

[Mic00] Microsoft Corporation. *Microsoft IL Assembly Programmer's Reference Manual*, July 2000. Part of the *.NET Framework Software Development Kit*, distributed on CD at the Microsoft Professional Developers Conference, Orlando, Florida, July 11–14, 2000.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[MWCG99] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.

[Nec97] G. Necula. Proof-carrying code. In *24th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[Nor98] M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[PG92] Y.G. Park and B. Goldberg. Escape analysis on lists. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–127. ACM Press, 1992.

[PHH$^+$93] S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings UK Joint Framework for Information Technology (JFIT) Technical Conference*, pages 249–257. 1993.

[PL91] S. Peyton Jones and J. Launchbury. Unboxed values as first class citizens. In *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666. Springer Verlag, 1991.

[PW92] S. Peyton Jones and P. Wadler. A static semantics for Haskell. Unpublished draft, Department of Computing Science, University of Glasgow. Available at *http://research.microsoft.com/users/simonpj*, 1992.

[Qia99] Z. Qian. A formal specification of Java^{TM} virtual machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1532 of *Lecture Notes in Computer Science*, pages 271–312. Springer Verlag, 1999.

[SA98] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proceedings POPL'98*, pages 149–160. ACM Press, 1998.

[Sha97] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

[Sym98] D. Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, 1998.

[Sym99] D. Syme. Proving Java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1532 of *Lecture Notes in Computer Science*, pages 83–119. Springer Verlag, 1999.

[TMC$^+$96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. PLDI'96*, pages 181–192, 1996.

[TT97] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[Van96] M. VanInwegen. *The Machine-Assisted Proof of Programming Language Properties.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, May 1996.

[vN99] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1532 of *Lecture Notes in Computer Science*, pages 119–156. Springer Verlag, 1999.

[Yel99] P.M. Yelland. A compositional account of the Java$^{\text{TM}}$ Virtual Machine. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 57–69. ACM Press, 1999.

A Facts Needed in the Proof of Theorem 1

This appendix encompasses the main lemmas needed in the proof of the main type safety theorem of the paper. Our proofs are for the definitions of (Good *fields*) and *pointerFree*$(-)$ described in Section 3. The proofs can trivially be adapted for the original definitions in Section 2. Appendix A.1 covers basic lemmas about the subtype and conformance relations. Appendix A.2 presents an alternative characterisation of the pointer conformance judgement $\Sigma \models ptr : A\&$. Finally, Appendix A.3 presents the definitions and typing properties of the store lookup and update functions.

A.1 Basic Lemmas

We begin with two lemmas about the subtype relation. Subtyping is trivial for all types except reference types. Only reference types can be supertypes of other reference types.

Lemma 1 *Assume $B \neq$ `class` c for all c. If $A <: B$ or $B <: A$ then $A = B$.*

Lemma 2 *If `class` $c <: A$ then there exists c' such that $A =$ `class` c' and c inherits c'.*

Although a subsumption rule is not part of the definition of the result conformance relation $\Sigma \models v : A$, it is derivable.

Lemma 3 *If $\Sigma \models v : A$ and $A <: A'$ then $\Sigma \models v : A'$.*

The next three lemmas concern how varying the size of the stack affects conformance.

Lemma 4 states that a pointer-free result well-formed in a store type (H, S) is also well-formed in the store type $(H, \varnothing)$. This justifies moving pointer-free results from the current frame to the heap.

Lemma 5 states that any result well-formed in a store type (H, S) is also well-formed in the store type $(H, S\ Fr)$. This justifies passing results from the current frame into the frame of a called method.

Lemma 6 states that a pointer-free result well-formed in a store type $(H, S\ Fr)$ is also well-formed in the store type (H, S). This justifies returning pointer-free results from a called frame to the previous frame.

Lemmas 4 and 6 do not apply to pointer results because if the result is a pointer into the top stack frame it is not well-formed in a smaller stack.

Lemma 4 *If $(H, S) \models v : A$ and pointerFree(A) then $(H, \varnothing) \models v : A$.*

Lemma 5 *If $(H, S) \models v : A$ then $(H, S\ Fr) \models v : A$.*

Lemma 6 *If $(H, S\ Fr) \models v : A$ and pointerFree(A) then $(H, S) \models v : A$.*

Next, we have two lemmas concerned with method call and return.

Lemma 7 says that a frame is well-formed in the store $(H, S Fr)$ if it is well-formed in the store (H, S). This justifies passing an argument frame to a called method.

Lemma 8 says that a store (h, s) conforms to the store type (H, S) if the store $(h, s\ fr)$ conforms to a store type $(H, S\ Fr)$. This justifies returning from a method.

The proof of Lemma 8 depends on showing that no pointer in the final store (h, s) refers to the frame *fr*.

Lemma 7 *If $(H, S) \models fr : Fr$ then $(H, S\ Fr) \models fr : Fr$.*

Lemma 8 *If $(H, S\ Fr) \models (h, s\ fr)$ then $(H, S) \models (h, s)$.*

Recall that we state Theorem 1 in terms of a relation $H \leq H'$ defined to mean that $dom(H) \subseteq dom(H')$ and $H(p) = H'(p)$ for all $p \in dom(H)$. We may call this the *heap extension* relation. Heap extension is a partial order.

Lemma 9 *The relation $H \leq H'$ is reflexive and transitive (that is, for all H, H', and H'', $H \leq H$, and, if $H \leq H'$ and $H' \leq H''$ then $H \leq H''$).*

The next three lemmas state that heap extension preserves the conformance relations for results, objects, and frames.

Lemma 10 *If $(H, S) \models v : A$ and $H \leq H'$ then $(H', S) \models v : A$.*

Lemma 11 *If $H \models o : c$ and $H \leq H'$ then $H' \models o : c$.*

Lemma 12 *If $(H, S) \models fr : Fr$ and $H \leq H'$ then $(H', S) \models fr : Fr$.*

The final lemma of this section justifies boxing of results. If the heap h and the object o both conform to the heap type H, and p is a fresh reference, then the extended heap obtained by allocating o at p is well-formed.

Lemma 13 *If $H \models h$ and $p \notin dom(h)$ and $H \models o : c$ then $H, p \mapsto c \models h, p \mapsto o$.*

A.2 Another Formulation of Pointer Conformance

In the next section we present the recursive definitions of the *lookup* and *update* functions on pointers. To show properties of these functions, it is convenient to present in this section a reformulation of the pointer conformance relation $\Sigma \models ptr : A\&$. Essentially, we show that every well-formed pointer takes the form of either (1) a pointer to an argument in a frame, followed by a possibly empty path of field selections, or (2) a reference to a boxed object of a value class, followed by a possibly empty path of field selections, or (3) a reference to a boxed object (not necessarily of a value class) followed by a non-empty path of field selections.

This reformulation begins with a notion of a path, a possibly empty sequence of field names.

Path Within an Object:

$\vec{f} ::= f_1 \cdots f_n$ sequence of fields (written ϵ if $n = 0$)

Next, we define a relation $A \overset{\vec{f}}{\Longrightarrow} B$ to mean that either the sequence $\vec{f}$ is empty and $A = B$, or that A is a value class, and selecting the fields in the series $\vec{f}$ in order yields the type B. This is defined in terms of $A \xrightarrow{f} B$, an auxiliary single step relation.

Actions of Fields on Types: $A \xrightarrow{f} B$ and $A \overset{\vec{f}}{\Longrightarrow} B$

$A \xrightarrow{f} B$ if and only if $A =$ `value class` vc and
$\quad$ $\mathit{fields}(vc) = f_i \mapsto A_i\ ^{i \in 1..n}$ and $f = f_j$ and $B = A_j$.
$A \overset{\epsilon}{\Longrightarrow} B$ if and only if $A = B$.
$A \overset{f_1 \cdots f_n}{\Longrightarrow} B$ if and only if $A \xrightarrow{f_1} \cdots \xrightarrow{f_n} B$ (where $n > 0$).

Given these notations, we reformulate pointer conformance as follows.

Lemma 14 *The judgment $\Sigma \models ptr : A\&$ holds if and only if either:*

(1) *there exist (i,j), $\vec{f}$, and B such that $ptr = (i,j).\vec{f}$ and $\Sigma \models (i,j) : B\&$ and $B \overset{\vec{f}}{\Longrightarrow} A$, or*

(2) *there exist p, $\vec{f}$, and vc such that $ptr = p.\vec{f}$ and $\Sigma \models p :$ `value class` $vc\&$ and `value class` $vc \overset{\vec{f}}{\Longrightarrow} A$, or*

(3) *there exist p, f_j, $\vec{f}$, and c such that $ptr = p.f_j.\vec{f}$ and $\Sigma \models p :$ `class` c and $A_j \overset{\vec{f}}{\Longrightarrow} A$, where $\mathit{fields}(c) = f_i \mapsto A_i\ ^{i \in 1..n}$ and $j \in 1..n$.*

We use this lemma to prove the typing properties of store lookup and update functions stated in the next section.

A.3 Facts about Lookup and Update

We omitted the definitions of functions for store lookup $\mathit{lookup}(\sigma, ptr)$ and store update $\mathit{update}(\sigma, ptr, v')$ from the main body of the paper.

The store lookup function is defined in terms of an auxiliary function, result lookup $\mathit{lookup}(v, f_1 \cdots f_n)$, that given the result v, returns the outcome of applying each of the field selections $f_1, \ldots, f_n$ in turn. Here is the definition of this auxiliary function, followed by a typing lemma.

Result Lookup: $\mathit{lookup}(v, f_1 \cdots f_n)$

$\mathit{lookup}(v, \epsilon) \triangleq v$
$\mathit{lookup}(f_i \mapsto u_i\ ^{i \in 1..n}, f_j\ \vec{f}) \triangleq \mathit{lookup}(u_j, \vec{f})$ where $j \in 1..n$

Lemma 15 *If $\Sigma \models v : A$ and $A \overset{\vec{f}}{\Longrightarrow} B$ then $\Sigma \models \mathit{lookup}(v, \vec{f}) : B$.*

Next, we present the definition of store lookup, followed by a typing lemma.

Store Lookup via Pointer: $\mathit{lookup}(\sigma, ptr)$

$\mathit{lookup}((h,s), p.\vec{f}) \triangleq \mathit{lookup}(f_i \mapsto u_i\ ^{i \in 1..n}, \vec{f})$
$\quad$ where $h(p) = c[f_i \mapsto u_i\ ^{i \in 1..n}]$
$\mathit{lookup}((h,s), (i,j).\vec{f}) \triangleq \mathit{lookup}(v_j, \vec{f})$
$\quad$ where $s = \mathit{fr}_1 \cdots \mathit{fr}_i \cdots \mathit{fr}_m$ with $i \in 1..m$,
$\quad$ and $\mathit{fr}_i =$ `.args`$(v_0, \ldots, v_n)$ with $j \in 0..n$

Lemma 16 *If $\Sigma \models \sigma$ and $\Sigma \models ptr : A\&$ then $\Sigma \models \mathit{lookup}(\sigma, ptr) : A$.*

The store update function is defined in terms of an auxiliary function, result update $\mathit{update}(v, f_1 \cdots f_n, v')$, that given the result v, returns the outcome of updating the field indicated by the field selections $f_1, \ldots, f_n$ with the result v'. Here is the definition, together with a typing lemma.

Result Update: $\mathit{update}(v, f_1 \cdots f_n, v')$

$\mathit{update}(v, \epsilon, v') \triangleq v'$
$\mathit{update}(f_i \mapsto u_i\ ^{i \in 1..n}, f_j\ \vec{f}, v') \triangleq$
$\quad (f_j \mapsto \mathit{update}(u_j, \vec{f}, v'), f_i \mapsto u_i\ ^{i \in (1..n)-\{j\}})$ for $j \in 1..n$

Lemma 17 *If $\Sigma \models u : A$ and $A \overset{\vec{f}}{\Longrightarrow} B$ and $\Sigma \models v : B$ then $\Sigma \models \mathit{update}(u, \vec{f}, v) : A$.*

Given the previous auxiliary function, here is the definition of store update.

Store Update via Pointer: $\mathit{update}(\sigma, ptr, v')$

$\mathit{update}((h,s), p.\vec{f}, v') \triangleq$
$\quad (((h - p), p \mapsto c[\mathit{update}(f_i \mapsto u_i\ ^{i \in 1..n}, \vec{f}, v')]), s)$
$\quad\quad$ where $h(p) = c[f_i \mapsto u_i\ ^{i \in 1..n}]$
$\mathit{update}((h,s), (i,j).\vec{f}, v') \triangleq$
$\quad (h, \mathit{fr}_1 \cdots$ `.args`$(v_0, \ldots, \mathit{update}(v_j, \vec{f}, v'), \ldots, v_n) \cdots \mathit{fr}_m)$
$\quad\quad$ where $s = \mathit{fr}_1 \cdots \mathit{fr}_i \cdots \mathit{fr}_m$ with $i \in 1..m$,
$\quad\quad$ and $\mathit{fr}_i =$ `.args`$(v_0, \ldots, v_n)$ with $j \in 0..n$

Finally, we state two typing lemmas for store update. They are essential facts in the proof of type safety for BIL: the proof of Theorem 1 uses Lemma 18 and Lemma 19 to show that evaluations of `stind` and `starg`, respectively, are type safe.

Lemma 18 *If $\Sigma \models \sigma$ and $\Sigma \models ptr : A\&$ and $\Sigma \models v : A$ and $\mathit{pointerFree}(A)$ then $\Sigma \models \mathit{update}(\sigma, ptr, v)$.*

Lemma 19 *If $\Sigma \models \sigma$ and $\Sigma \models (i,j) : A\&$ and $\Sigma \models v : A$ and $\sigma = (h, \mathit{fr}_1 \cdots \mathit{fr}_i)$ then $\Sigma \models \mathit{update}(\sigma, (i,j), v)$.*