Generics in LWorld

alla JVM Explorations

Created by Karen Kinnear, last modified on Mar 01, 2019

Goals and Assumptions

New Capability: Fast generics over value types and primitives

- Allow Classes and Methods which currently support generics over object classes to
    - have parameterized types which span identifyful classes, value types and primitives
    - to support bounds which include identityful classes and value types (not primitives?)
- Support generic class evolution to support null-free types
- Deliver faster performance for generics over value types and primitives than generics over Objects today
    - without requiring manual coding of methods to handle each value type, and support dynamically added value types
    - still allow manual coding of methods to optimize for a given value type or primitive (primitives are hand-coded today)

User Model Requirements

Backward Compatibility Requirements - Erased Generics Behavior Today

- Erased Generics: List<T> always treated as List<Object> today (actually to List<Bound> at the language level)
    - List<Object> today erased in classfile to the class LList; vm never sees List<Object>
    - javac tracks the actual generic type and issues warning/errors to prevent heap pollution
    - Addition of specialized generics should not change the semantics of existing erased generics
        - existing clients and subtypes that work with erased generics should continue to work unchanged
- Raw Types: originally for migration from pre-generics to erased generics
    - "List" in source is a raw type
    - "List" is erased to LList; in the class file, which represents the class List.
    - Use of List in source, or LList; in the class file, allows any existing erased List<T>, so List<Object>, List<Interface>
    - "new List" translates in the class file today as new LList; and is instantiable
    - Supertypes of a raw type are erased. The type of an instance field/method declared in a raw type is the raw type that corresponds to the erasure of its type in the generic declaration.
- Wildcards:
    - Can be used in field or method declarations
    - Can not use wildcards in instantiation
    - Forms:
        - List<? extends X> ; source allows List<A> if A <: X
        - List<? super X>   ; contravariant: e.g. for a return type ; source allows List<B> if X <: B
        - List<?>
    - Erases to LList; in class file
    - VM does not see any difference between a raw type and a wildcard
- Recursive types:
    - List<List<T>>
    - All erase to LList; today
    - See open issues
- Subtyping Promises - Current basic assumptions
    - Class Subtyping
        - Given: A <: B, a:A, b:B
        - Promises: b = a // no cast required, (B)a // safe cast, (a instanceof B) == true, b.getClass().isAssignableFrom(a.getClass())

        - Current Array Covariance: Also given as:A[], bs:B[]
            - Promises: bs = as // no cast required, (B[])a[] // safe cast, (as instanceof B[]) == true, b[].getClass().isAssignableFrom(a[].getClass())
            - NOTE: we are exploring Covariance requirements with arrays of value types and of primitives as value types
- Relationships between generics today
    - source level:
        - class Foo<T> extends Bar<T> is explicitly declared (resp. class implements interface and interface extends interface)
            - implies raw Foo; is a subclass of raw Bar; (resp. subtype)
            - implies Foo<T> is a subclass of raw Bar; (resp. subtype)
    - class file level:
        - LFoo; subclass of LBar; (resp. subtype with interface(s))
    - source level invariance: If Integer <: Number, Foo<Integer> has no relationship to Foo<Number>:
        - javac will inject checkcasts for parameterized types
        - implies: jvm just needs to consider LFoo; handling for raw types and erased types but we do not need to allow invalid behaviors for specialized types
- Class file Signature Attribute:
    - JVMS 4.7.8.1 states that javac is required to emit this attribute for any ClassFile, field_info or method_info whose declaration uses type variables or parameterized types
    - The JVM does not use these, they are used by platform libraries such as class.getGenericSuperClass() and jlreflect.Executable.toGenericString()
    - My translation is that this information is available for declarations but not use sites.
- Generics Type Witness - also called NonWildTypeArguments - source explicit typing when a type can not be inferred, e.g. for a constructor, a method parameter or for a return type.
    - This is information for the static compiler in source and does not impact this exercise

Migration Constraints

- Support Incremental Transitions
    - Step 1: support erased generics over identity objects and null-tolerant value types
        - Assumption is that erased generics will continue to assume null-tolerance
    - Step 2: support generic specialization
        - explicit author opt-in to support specialized generics which handle null-free types
        - no semantic change for users of erased generics as generics become specialized
        - If we can support specialized generics over primitives wrapped as null-free value types, then we can make this a single migration step to support value types and primitives
- Migration Goal is to allow independent incremental support for value types and for generic specialization in any order for source and binary compatibility for each of the following:
    - Authors
        - evolve a value-based-class to a null-tolerant value type
        - opt-in to generic specialization
            - e.g. evolve APIs to also support null-free value types
        - these two steps can be performed in either order
    - Subtypes
        - no change
        - opt-in to support specialized generics over null-free value types
    - Exporters (better name?) - their APIs are defined in terms of a generic class which is changing its semantics
        - no change
        - explicit opt-in to support specialized generics over null-free value types
    - Clients - reference members: constructors, fields, methods
    - Clients of Subtypes (I believe there are cases in which this also needs to be examined)
        - Can they take advantage of generic specialization prior to subtypes opting in to generic specialization?
- Value-based class migration
    - Erased generics over existing value-based-classes which migrate to value classes
        - Existing uses will continue to be erased (these are null-tolerant value types)
        - Uses with specialized generics may require null-free value types
    - Generic value-based-class migration to generic value classes (e.g. Optional<T>
        - Existing uses will continue to use erasure

Requirements for Specialized Generics

- Define a species as a specialization of a generic class based on a fully concrete set of type parameters
    - The type parameters must be visible to the client of the specialization, but not required to be visible to the generic class
- All species must share the same java.lang.Class, raw type, version, package/module/loader/protection domain
- All species must extend all superclasses and implement at least all interfaces of the root class
    - there may be a new relationship between a species and the root class, and therefore the superclasses/super-interfaces of the root class
- raw-reachable: reachable by a raw type
- Language requirement to allow declaration of species-specific components
    - species-scope components include
        - species-super interfaces
        - species-static members
        - species-instance members
        - species-nested classes
    - species-specific members are not reachable via existing raw types
- Conditional class components
    - In order to allow optimizations for species, the java language is adding the concept of conditional class components, possibly represented by "when"
        - possibly dependent on parameter bounds
- Members of a species
    - class-members
        - class-statics
            - static members of the generic class are non-parameterized and shared across all species
            - all static fields and methods of the raw type are shared across all specializations unchanged, including flags and attributes
            - synchronization on static methods is on the java.lang.Class and remains unchanged
            - class-statics are raw-reachable
        - class-instance
            - non-parameterized instance members of the class
                - these are the same across all specializations including signature, flags and all attributes
            - class-instance members are raw-reachable
    - species-members
        - no species members are shared with any other species
        - species-static members
            - these can be explicitly declared as species-specific or conditional members
            - species-static members are not raw-reachable
            - which can be parameterized, because you must have concrete type parameters to specialize the type before accessing a species-static
            - shared and synchronized across the species
        - species instance members
            - species-specific instance members
                - explicitly declared species-instance members or conditional members
                - species-specific members are not raw-reachable
            - specialized instance members
                - each species has specialized class-members for all parameterized class type members
                    - if the parameter is erased, the signature must be an exact match with the raw type member
            - erased parameterized instance members
                - parameterized type members are raw-reachable
                    - erased parameterized class-member mechanism (e.g. virtual members e.g. bridge, forwarding)
- Member scope
    - class-static members can refer to class-static members only
    - species-static members can refer to species-static and class-static members only
    - class-instance members can refer to class-instance and class-static members only
    - species-instance members can refer to all names
- Resolution & Selection Search Orders: "most-specific principle"
    - For each lookup scope (local, superclass, superinterfaces), the species members should be searched before

    the class members
        - TBD whether the class (or template-class) is a new kind of species, or whether it is considered at the same lookup scope
    - Method overriding should have species members override class members at a given lookup scope

- Specialized instance members raw-reachability: for Raw, Wildcard and Erased client references
    - FieldReference or MethodReference via a raw List, can operate on any receiver which is a specialization of the generic List
        - REFC today is always LList; receiver can be List, List<Object>, List<Point>, List<ValInt>
        - field "T f1" itself may be specialized to a field declaration of bound, Object, Point, Valint respectively in terms of object layout
        - method m()T itself may be specialized to a parameter/return declaration of bound, Object, Point, Valint respectively
        - For backwards compatibility with raw generics, you must always be able to access any member common across specializations
            - as if we have a virtual field of the erased type
            - as if we have a virtual method with a parameter/return of the erased type
- Reflection Support
    - getClass()
        - expect user level view as a single Class
    - getSpecies()
        - additional detail available for new code

Generic Value Classes

- new value classes
- migration of generic value-based-classes to value classes - author opt-in to value class migration
    - may wish to benefit from value class optimizations via specialization - author opt-in to specialization

Subtyping Relationships for specialized generics

    - class Foo<T> extends Bar<T>
        - Subtyping relationships we want: - must obey Subtyping Promises above
            - Foo<Valint> <: Bar<Valint> // propose this as a precise superclass relationship
            - Foo<Valint> <: Foo<> / raw LFoo; TODO - double-check - is this part of the reason for distinguishing Foo<erased> and raw LFoo;
            - No relationship between Foo<Valint> and Foo<erased>
    - source level invariance: If Integer <: Number, Foo<Integer> has no relationship to Foo<Number>:
        - javac will inject checkcasts for parameterized types
        - implies: jvm just needs to consider LFoo; handling for raw types and erased types but we do not need to allow invalid behaviors for specialized types
- Generic types are still invariant:
    - If Foo <: Bar:
        - There is NO expected relationship between List<Foo> and List<Bar>

Access Control Requirements

- Access Control Requirement across species
    - All species of the same class Foo should have access to all shared members
    - all are nest mates - for private member access (potential partial solution)
    - Individual species may add their own members - access control requirements (see questions below)

Open Questions on User Model Requirements

- Relationship between LFoo; and LFoo<erased>
    - Proposed non-instantiable *caveat class vs. erased specialization
    - Must have exact match on members
    - TODO: details of when we want the same behavior and when we want different

- In existing class files, can the vm tell the difference between a raw type, an erased type and a wildcard type since they are all represented by LFoo;
    - No.
    - Note: Wildcard type can not be instantiated today, so never the subject of "new" and never the type of a live type
- recompilation of existing client class files
    - use of existing generics over Object - will javac translate all of these to List<erased> ?
        - OPEN ISSUE
    - use of a raw type in source: will javac translate to List<erased> ? Or leave as LList; assume generate erased?
        - A: Brian: will leave as LList; for raw type
    - Challenge 1: see chart below: class LFoo; is instantiable today, that code needs to continue to work
        - Open Issue: possibly request that new LFoo; is translated "magically" by the vm prior to verification to new LFoo<erased>
    - Challenge 2: if recompilation changes class file from LList; to List<erased>; then e.g. field/method reference through a class LList; → List<erased> is now translated differently and restricts the potential receivers.
    - Challenge 3: If a FieldReference or MethodReference contains an erased generic parameter- e.g. m(Foo<erased>)
        - will javac generate this?
        - Will this only support a receiver of Foo<erased> and subtypes, i.e. no other specializations of Foo?
        - Does this imply that the signatures will use the raw type rather than the erased specialization when the goal is operating on any specialization?
        - Does the author have a way to distinguish the intended scope?
- Conditional superinterfaces, methods, instances, nested classes - is there anything else that could be per-species?
    - Can you specify a per-species superclass?
        - Many possible vm mechanisms will depend on the superclass for a given specialization to be the equivalent specialization of the superclass of the root type
        - per-species random replacement of the superclass could break assumptions
    - Are there any attributes of members shared across all species or specialized across species which can vary by species?
    - Can you have conditional class components defined for the erased species? Can we assume those are not accessible via the raw type, or do we default to erased parameters when accessing List.speciesstatic1 ?
- How are recursive type (List<List<...>> limitations handled today by javac and how will they be handled going forward?
- Custom specialization - e.g. hand-crafted
    - What are the rules the verifier should use to ensure a custom specialization is "complete"? It is required to support specializations for all existing raw class members, correct? What about supporting members that are defined for conditions the parameters for this specialization meet? Concern about opening up the specialization capabilities to random authors.
- Requirements from jvm for generic method support
- Incremental migration
    - author opts-in to specialized generics
    - chart of client/subtype/client of subtype migration

Proposed Answers on User Model Requirements

- Serviceability
    - How would breakpoints see the species?
        - Initial proposal: breakpoints would be added across all species - and fit the user's model of single source
    - How would redefineclasses see the species?
        - Initial proposal - single source class, so redefine all species
        - John also proposed ability to redefine/CFLH for each species as well as for the template. OUCH

Specialized Generics in LWorld Proposed Alternatives

ATLASSIAN

ATLASSIAN