Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Tesis de Licenciatura

Anotación automática de tipos para colecciones en ambientes dinámicos

Autores

De Sousa Bispo, Mariano Edgardo
LU: 389/08
marian_sabianaa@hotmail.com

Felisatti, Ana Laura
LU: 335/10
anafelisatti@gmail.com

Director

Lic. Hernán Wilkinson

Anotación automática de tipos para colecciones en ambientes dinámicos

Las colecciones son el principal ejemplo de tipos paramétricos o genéricos y son, a su vez, utilizadas ampliamente en el desarrollo de software, dándole una importancia aún mayor a su correcto tipado. Este trabajo presenta una implementación para la anotación automática de tipos sobre dichos objetos, basada en los tipos añadidos a sus instancias y extendiendo el soporte de LiveTyping.

LiveTyping es un sistema de anotación automática de tipos con el fin de mejorar la experiencia de desarrollo en ambientes dinámicos, donde la falta de tipado estático dificulta la implementación de herramientas. Su funcionamiento se basa en la intervención a nivel VM de primitivas de asignación y ejecución de métodos, puntos donde se inspeccionan y recolectan los tipos observados. Al funcionar sobre un ambiente de desarrollo vivo donde la misma VM se utiliza tanto para ejecutar código como para ejecutar la IDE, la información recolectada es inmediatamente puesta a disposición de las herramientas de desarrollo.

La investigación se centra en extender este sistema a colecciones, donde la información de tipos sólo puede ser recolectada de manera diferida sobre las instancias, a medida que los objetos son añadidos a las mismas. En este trabajo discutiremos los cambios sobre la VM e imagen realizados para ello, los cambios sobre el ambiente de desarrollo para beneficiarse de esta nueva información disponible, así como los desafíos enfrentados.

Como conclusión, se presenta una generalización de la implementación realizada y se discuten los cambios requeridos para extender LiveTyping de manera de soportar todo tipo paramétrico.

Palabras claves: Virtual Machine, LiveTyping, Tipado Estatico, Tipado Dinámico, Tipos Paramétricos, Generics, Parametrización Polimórfica, Live Development Environments

Agradecimientos

- A nuestras familias, por despertar en nosotros la curiosidad por la ciencia, por enseñarnos la importancia del pensamiento crítico y por el inconmensurable esfuerzo que fue acompañarnos hasta este punto.
- A Exactas y la UBA por darnos aquellas herramientas que cultivan el pensamiento científico y abren las puertas del conocimiento.
- A Hernán Wilkinson por contagiarnos la pasión por la profesión, el diseño con objetos, y el enseñar, cambiando para siempre nuestra actividad profesional.
- A mi par en este trabajo de tesis, y mi par para el resto de mi vida.

Índice

Introducción

Tipado

Los lenguajes de programación pueden depender de un sistema de tipos. Llamaremos *type binding* al proceso de asignar un tipo a un valor y *type checking* al proceso de verificar si los operandos de una operación tienen tipos compatibles. Los errores de tipos ocurrirán cuando se aplique un operando de tipo incorrecto a una operación. (Ebraert et al., 2005)

Es importante distinguir estos conceptos aunque dependan el uno del otro. Ambos pueden ocurrir en tiempo de compilación, estáticamente, o en tiempo de ejecución, dinámicamente, o en cualquier combinación. (Ebraert et al., 2005)

Luego, un *type binding* estático significa que el sistema de tipos determina en tiempo de compilación el tipo de cada variable, requiriendo de información de tipos provista por el desarrollador en forma de anotación de tipos explícitas o por inferencia de tipos. Esto permite al sistema de tipos verificar en tiempo de compilación si una operación es compatible con los tipos de sus operandos. Esta verificación se llama *type checking* estático. Los sistemas de tipos que combinan *type binding* estático con *type checking* estático se conocen como estáticamente tipados. (Ebraert et al., 2005)

Al contrario, un *type binding* dinámico determina el tipo de una variable basándose en su valor en tiempo de ejecución. Por lo tanto, la verificación de compatibilidad entre los tipos de operandos y una operación sólo es posible en tiempo de ejecución, dinámicamente. La combinación de un *type binding* dinámico y *type checking* dinámico es conocida como tipado dinámico. (Ebraert et al., 2005)

Algunos ejemplos de lenguajes dinámicamente tipados son JavaScript, Smalltalk, Ruby o Python. Otros como Java y C#, por su parte, combinan *type binding* y *type checking* dinámico y estático, mientras que los lenguajes Assembly no usan ninguno de los 2 procesos (Ebraert et al., 2005) (Tratt, 2009).

Los esfuerzos por entender el impacto del sistema de tipos en la usabilidad de los lenguajes no han sido concluyentes hasta ahora (Koeppe, 2018). Sin embargo, podemos destacar una serie de estudios donde se observó beneficios en la productividad del desarrollador con el uso de lenguajes estáticamente tipados respecto a la mantenibilidad de un sistema (Kleinschmager et al., 2012) y el uso de APIs (Petersen et al., 2014) (Endrikat et al., 2014) (Fischer & Hanenberg, 2015). Estas tareas están impactadas por las herramientas que el entorno de desarrollo provee para ganar conocimiento sobre el sistema y el código fuente, por ejemplo, la navegación, la compleción de código y los *refactorings*. Como se observa en los estudios sobre entornos de desarrollo modernos (Petersen et al., 2014) (Fischer & Hanenberg, 2015), las capacidades en el caso de lenguajes dinámicamente tipados aún no se equiparan a

las de sus contrapartes debido en gran parte a la falta de información de tipos que pueda ser fácilmente explotada (Dolby, 2005).

Resulta crítico entonces acceder a la información de tipos y proveer herramientas de desarrollo para lenguajes dinámicamente tipados que asistan en la comprensión del comportamiento de un programa, prerrequisito para tareas de mantenimiento o extensión de un sistema (Röthlisberger et al. 2008).

Varias estrategias pueden utilizarse en lenguajes dinámicos para obtener acceso a la información de tipos en tiempo de desarrollo:

- Inferencia de tipos, la asignación automática basada en análisis estático.
- Anotación manual, mediante la extensión del lenguaje o el uso de anotaciones de preprocesamiento
- Monitoreo de tipos, la captura de tipos en ejecuciones específicas.

La inferencia de tipos dependerá mucho del alcance contextual que pueda obtenerse y su costo pero es la opción menos invasiva ya que no requiere intervención del usuario. Dado que la capacidad de evitar las declaraciones constantes de tipos es una de las virtudes de los lenguajes dinámicamente tipados, la anotación manual resulta más una transformación del lenguaje que una estrategia en sí misma. El monitoreo de tipos, por su lado, requiere una ejecución particular sobre un modelo alterado para poder capturar los tipos y limita el uso de estos tipos a ese ambiente de debugging (Wilkinson, 2019).

En este sentido, es interesante considerar las posibilidades que un ambiente vivo y unificado de desarrollo y ejecución ofrecen. Herramientas como Hermion (Röthlisberger et al., 2008) y *Type Harvesting* (Haupt et al., 2011) utilizan las ventajas en metaprogramación de Smalltalk para recolectar información de tipos y mejorar en simultáneo las capacidades en tiempo de desarrollo. El costo de dichos análisis es elevado, sin embargo, los autores proponen la evaluación de una implementación a nivel máquina virtual (VM) como alternativa para mejorar los tiempos de ejecución. LiveTyping (Wilkinson, 2019) es justamente, una técnica que aprovecha el ambiente vivo y unificado de desarrollo y ejecución que ofrece Smalltalk para capturar información de tipos a nivel VM y hacer uso de la misma en tiempo de desarrollo.

LiveTyping

En esta técnica, la *virtual machine* recolecta los tipos observados durante la ejecución de asignaciones, *method calls* y en el *bytecode* de retorno, obteniendo así información de tipos para variables de instancia, variables temporales, parámetros de funciones y tipos de retorno.

Actualmente, existe una implementación de LiveTyping en OpenSmalltalk-VM[1] y la imagen de Cuis[2].

A diferencia de las herramientas antes mencionadas, Hermion (Röthlisberger et al., 2008) y *Type Harvesting* (Haupt et al., 2011), LiveTyping captura los tipos a nivel VM sin requerir intervención del desarrollador ni una suite de test particularmente buena. Su costo de ejecución es mínimo en comparación a esas herramientas, permitiendo su ejecución constante de ser necesario (Wilkinson, 2019). De esta manera, permite extender las herramientas de desarrollo para incorporar la información de tipos con todos los beneficios observados para *Hermion* y *Type Harvesting* con una performance significativamente superior.

Dos cambios fundamentales posibilitan esto:

- La adición de *typedArrays* en toda clase y método, utilizados para recolectar los tipos.
- La intervención a nivel VM de primitivas, donde se populan dichos arrays.

Existen limitaciones a la solución actual de *LiveTyping*, sin embargo. Utilicemos el siguiente caso para ejemplificar el caso más relevante para nuestro trabajo:

```
setVariableWith: aValue andAnotherVariableWith: anotherValue
    instVariable := aValue.
    anotherInstVariable := anotherValue.
```

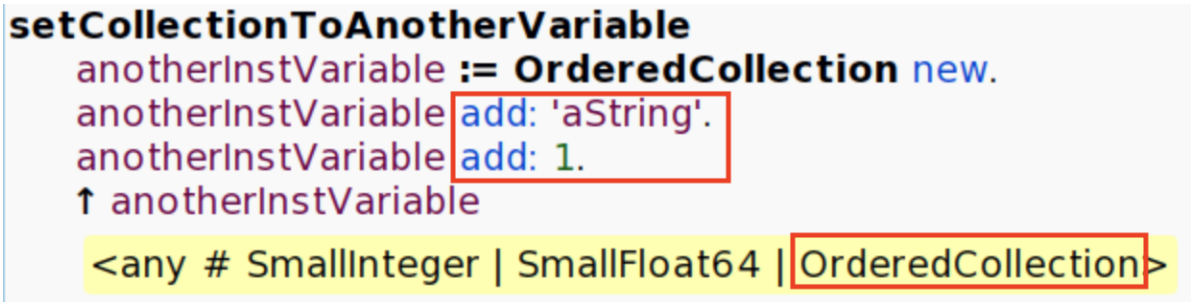
Con *LiveTyping*, la VM intercepta la asignación y *method activations* para guardar los tipos de las variables y parámetros. Además, esta información de tipos queda resguardada en la clase o método que contiene la variable. Supongamos entonces dos ejecuciones del método, donde el primer parámetro será el *string "value"* mientras que el segundo será inicialmente el entero 1 y luego el número de punto flotante 1.0.

Utilizando el tooling actual de *LiveTyping* podemos ver mediante un *mouseover* sobre la variable *anotherInstanceVariable* que esta ha sido asignada con dos tipos que representan números:

```
setVariableWith: aValue andAnotherVariableWith: anotherValue
    instVariable := aValue.
    anotherInstVariable := anotherValue.

    <Number # SmallInteger | SmallFloat64>
```

[1] OpenSmalltalk-VM: https://github.com/OpenSmalltalk/opensmalltalk-vm
[2] Juan Vuletich. 2008. Cuis: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk

Estas evaluaciones involucran tipos simples, veamos que sucede en el caso de tipos paramétricos, los que dejan sin especificar alguno de sus tipos constituyentes y luego se proporcionan como parámetros al momento de su uso (Gamma et al. 1994), como es el caso de las colecciones. Esto también se conoce como tipos genéricos *(generics)* o parametrización polimórfica (Pierce, 2002). Al asignar una colección a la variable, *LiveTyping* guarda el tipo de ella, pero no los tipos de los elementos de la misma. Más aún, *LiveTyping* no tiene forma de saber ni poder recolectar los futuros tipos de los elementos que se agreguen a posteriori.



Vemos en el tooling que la información de tipos para *anotherInstVariable* sólo contiene a *OrderedCollection* en vez de *OrderedCollection<any # String | SmallInteger>* como el agregado de los elementos a la colección se hace después de la asignación, esta información no se tiene disponible. Este problema es propio de los tipos paramétricos ya que sólo al momento de su instanciación pueden conocerse los tipos efectivos y en el caso de un lenguaje dinámicamente tipado como Smalltalk, esta instanciación ocurre de manera diferida y específica para cada tipo, no al momento de la asignación de una variable, el envío de un mensaje o el retorno de los mismos, los puntos donde *LiveTyping* opera hasta el momento. Llamamos a este problema asignación de tipos diferida o *deferred type binding.*

Esta limitación sobre el uso de clases con tipos paramétricos es importante dado que hacen a la completitud del sistema de tipos. Por otro lado, hacen una diferencia muy grande en la comprensión de un sistema, incluso considerando un uso excesivo de tales tipos y herramientas de desarrollo faltantes (Hoppe & Hanenberg, 2013). Finalmente, su importancia es crítica cuando consideramos el extenso uso de colecciones en el modelado de soluciones computacionales.

El objetivo de este trabajo de tesis fue extender *LiveTyping* para obtener información de tipos sobre la jerarquía de colecciones y mejorar las herramientas existentes con esta nueva fuente, concentrándose en un principio en aquellas basadas en arrays dado que cubren la gran mayoría de los casos. Como objetivo secundario, buscamos idear una abstracción para cubrir de manera genérica los tipos paramétricos y poder a futuro extender aún más *LiveTyping*.

Problema

Evaluemos con un diagrama de secuencia las principales interacciones que tendrán lugar con la ejecución del ejemplo anterior (crear la colección y agregarle un *string* y un entero) entre la VM y los objetos de la imagen.

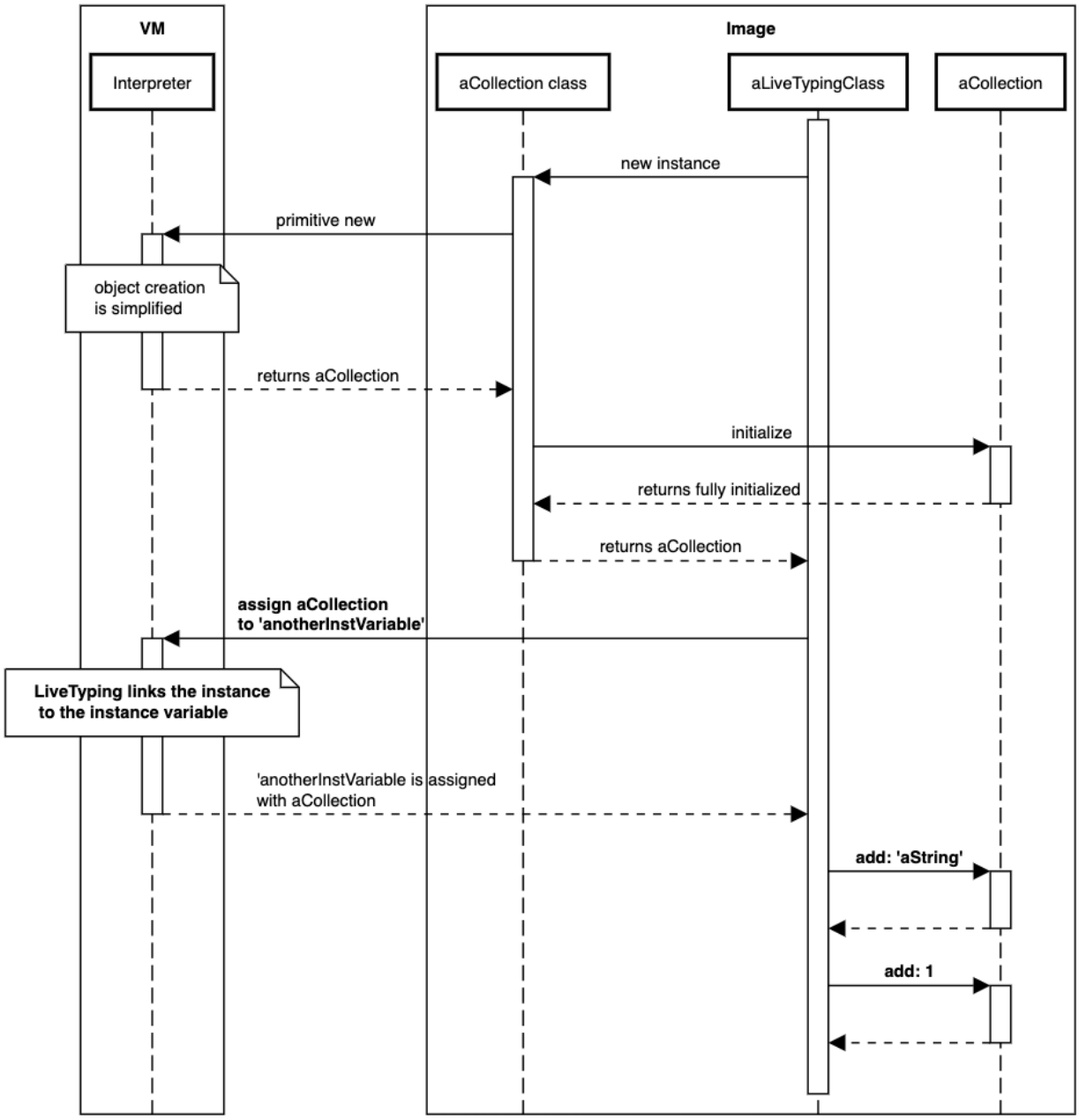


Diagrama de secuencia de la asignación de una colección a una variable, y el posterior agregado de un string y un entero.

Podemos distinguir tres principales interacciones:

- La instanciación de la colección
- La asignación de la colección a una variable
- La inserción de elementos en la colección

El principal desafío de capturar los tipos paramétricos de las colecciones es que, si bien en nuestro ejemplos las últimas interacciones son secuenciales, en verdad podrían suceder en cualquier orden, reiteradas veces y de forma independiente. Por otro lado, mientras que hasta ahora *LiveTyping* analiza los tipos en base a las clases estáticas de los distintos objetos, analizar el tipo de colecciones requiere evaluar sus instancias específicas a lo largo del tiempo.

En este caso*, LiveTyping* recolecta los tipos al momento de la asignación del objeto a una variable. Para colecciones, sin embargo, no alcanza sólo con la clase del objeto asignado, sino que se debe mantener relación activa de forma tal que en el posterior agregado de elementos en esas instancias de colección, esta información pueda accederse.

Llamamos a este problema donde el type binding ocurre posteriormente a la instanciación de un objeto *type binding diferido.*

Este problema nos lleva a enfrentar adicionalmente un cambio de magnitud respecto a los objetos analizados. Hasta el momento, *LiveTyping* inspecciona los objetos capturando sus clases. Ahora, sin embargo, debemos considerar las instancias en sí mismas y no solo sus clases, ya que su tipado podrá modificarse en el tiempo. En una imagen de Cuis standard tenemos alrededor de 1.900 clases y alrededor de 480.000 instancias.

Para resolver estos problemas en el caso de colecciones basadas en arrays, creamos un array tipado, es decir, con una variable interna donde se capturan los tipos de los elementos insertados al mismo. Esto último es posible mediante la intervención del mensaje *at:put:*.

Luego, extendimos la lógica existente de captura de tipos de *LiveTyping* para reconocer las instancias de colecciones y capturar esta nueva variable interna del array en vez de su clase. De esta manera, mantenemos una relación entre el tipado de una variable de instancia y colecciones asignadas a la misma ya que la variable interna de los arrays de las colecciones irán reflejando su contenido.

Finalmente, para mitigar la explosión de instancias, asociamos esta nueva variable interna del array a su uso, de manera de fusionar nuevas instancias a las capturadas previamente. Es decir, cuando a una variable de instancia se le asigne una colección, la captura de la nueva variable tendrá en cuenta la existencia de previas capturas para mantener una única referencia.

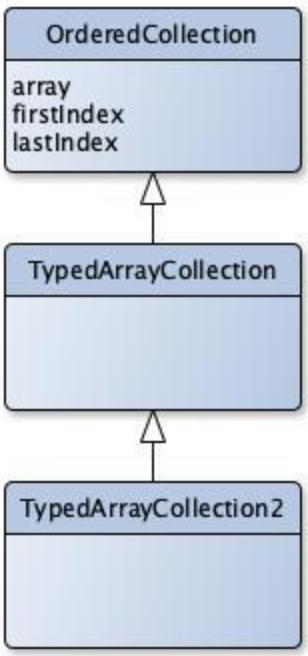
Detallaremos a continuación la implementación de esta solución. Al iterar la tesis, descubrimos una generalización al problema y la implementación inicial se transformó en una prueba de concepto. Se concluirá más adelante que los mecanismos actuales de *LiveTyping* pueden generalizarse para soportar *type binding diferido.*

La sección de implementación de este informe, se divide en explicar:

- *type binding diferido* para colecciones.
- cómo se implementó para un subconjunto de colecciones.
- cómo se puede extender el *tooling* para mejorar la experiencia de desarrollo
- cómo puede generalizarse el *type binding diferido* para cualquier tipo de objeto.

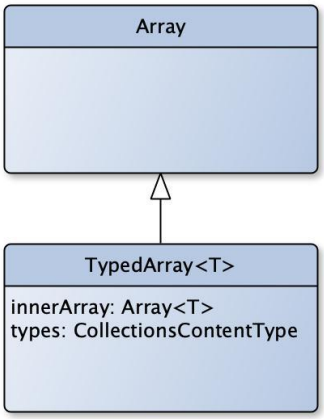
Se agrega además como apéndice, los desafíos encontrados en el desarrollo del trabajo.

Se crearon dos colecciones para la prueba de concepto: *TypedArrayCollection* y *TypedArrayCollection2*. Dado que estas colecciones no se usan por el sistema, simplifican el proceso de *debugging* y desarrollo. Sin embargo, los conceptos aprendidos, son extensibles para cualquier implementación.



Jerarquía de herencia para TypedArrayCollection y TypedArrayCollection2.

Ambas colecciones sobreescriben la utilización del *Array* provisto por la imagen en favor de un *TypedArray* que decora el *Array* subyacente agregando la variable *types* donde se capturan los tipos.



Jerarquía de herencia para TypedArray.

NOTA: A partir de este momento, utilizaremos en los gráficos, los caracteres '<' y '>' para delimitar el generics o la instanciación del generic.

Type binding diferido para colecciones

Para poder soportar *LiveTyping* en colecciones, es necesario resolver el *deferred type binding* de las mismas. Una implementación que cumple con los siguientes 5 ítems resuelve el problema:

1. ***Reconocimiento de colecciones a nivel VM:*** Permitir el reconocimiento de todos los tipos de colecciones que se quieran usar en el momento de la captura de tipos para distinguirlas de las clases regulares.
2. ***Almacenamiento de tipos paramétricos:*** Crear y controlar un objeto que permita guardar los tipos de los elementos de la colección.
3. ***Captura de tipos paramétricos*** Conocer qué métodos agregan elementos a cada colección para realizar la captura de tipos de los objetos insertados.
4. ***Relación entre tipos paramétricos y sus usos:*** Mantener una relación entre ese objeto y la variable a la que se está asignando.
5. ***Aliasing:*** Unificar la información de tipos para instancias de colecciones con un contexto común.

1. Reconocimiento de colecciones a nivel VM

El *specialObjectsArray* es un arreglo definido por la imagen que es bien conocido por la VM. Sus elementos, pueden ser utilizados por la *virtual machine* durante su ejecución. Las VMs que utilicen este arreglo quedan altamente acopladas con las imágenes que soportan, ya que deben poseer el arreglo cargado. Su definición se encuentra en el método *recreateSpecialObjects* de *SystemDictionary.* La imagen tiene una única instancia del *specialObjectsArray*.

Se agregaron dos entradas al *specialObjectsArray*: la primera contiene las clases de colecciones que soportan *LiveTyping;* la segunda entrada, corresponde a índices que parametrizan una búsqueda que debe efectuar la VM para que funcione el algoritmo.

Con el *specialObjectsArray*, la VM de *LiveTyping* es modificada para validar en la asignación, si el objeto a asignar es una colección o no y luego actuar en consecuencia. Vemos a continuación, el método '*collectionIndexFor:'* agregado al intérprete de la VM, el cual detecta si la clase del objeto a asignar a una variable es una colección o no.

```
collectionIndexFor: aClass

    <inline: true>
    | isNotCollection isCollectionSearchIndex supportedCollectionAtIndex supportedCollections
supportedCollectionsSize collectionIndex|

    isNotCollection := true.
    supportedCollections := objectMemory splObj: SupportedCollections.
    supportedCollectionsSize := (objectMemory lengthOf: supportedCollections)-0.
    isCollectionSearchIndex := 0.
    [isNotCollection and: [isCollectionSearchIndex < supportedCollectionsSize]]
        whileTrue: [
            supportedCollectionAtIndex := objectMemory followObjField: isCollectionSearchIndex
ofObject: supportedCollections.
            supportedCollectionAtIndex = aClass
            ifTrue: [
                isNotCollection := false]
            ifFalse: [
                isCollectionSearchIndex := isCollectionSearchIndex +1].
             ].

    collectionIndex := isCollectionSearchIndex.
    isNotCollection ifTrue: [collectionIndex := -1].

    ^ collectionIndex.
```

Código a nivel VM utilizado para reconocer instancias de colecciones.

NOTA: El intérprete de la VM tiene como colaborador al *ObjectMemory*, el cual puede acceder al *specialObjectsArray* vía el mensaje *`splObj: SupportedCollections`*.

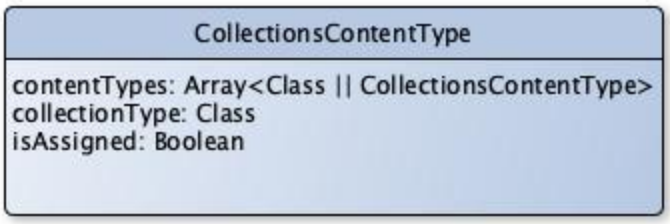
Para reconocer si es una colección soportada por *LiveTyping* entonces, se analiza el nuevo objeto agregado del *specialObjectsArray*, que llamamos *SupportedCollections,* para determinar si la clase siendo evaluada pertenece al array o no. Para optimizar la performance del algoritmo general, el mensaje no retorna simplemente un boolean: retorna el índice de la clase en el array, de efectivamente encontrarse allí (*-1* en caso contrario). De esta manera, podemos reconocer colecciones y obtener el índice correspondiente del segundo array en una sola operación. El segundo arreglo agregado, es una tabla de índices para encontrar el objeto que describiremos a continuación: el *CollectionsContentType.*

2. Almacenamiento de Tipos Paramétricos

Definimos a *CollectionsContentType (CCT),* como la clase responsable de mantener los tipos asociados a los elementos que se agregan a las colecciones. Este objeto referencia al menos a:

- La clase de la colección,

- Una lista de clases y/o *CollectionsContentType*. Cada entrada representa el tipo de algún elemento en la colección, y en el caso de colecciones anidadas, se representa con *CCT*s anidados.
- Si la colección que representa fue asignada previamente a una variable *(colaborador isAssigned)*, se necesita además considerar el aliasing.



Definición de CollectionsContentType (CCT).

En resumen, evaluamos para colecciones basadas en arrays la introducción de una implementación de array *(TypedArray)* que recolecta los tipos introducidos en un objeto que llamamos *CollectionsContentType (CCT)* mediante la intervención del mensaje *at:put:.* Si bien los *CCTs* se encuentran en cada instancia de array, se unifican y reutilizan a medida que son asignados de manera de quedar asociados principalmente a su uso en variables de instancia o mensajes.

Luego, para soportar el tipado de tales colecciones en *LiveTyping*, la recolección usual de tipos a nivel VM fue extendida para, en caso de tener una colección, recolectar su *CCT* correspondiente en vez de su clase. *LiveTyping* tiene un solo punto efectivo de recolección del que dependen todas las primitivas analizadas: el mensaje *keepTypeInformationIn:for:.* Este mensaje fue refactorizado y modificado para evaluar si el objeto analizado es una instancia de colección y actuar en consecuencia.

En primer lugar, se extrajo la lógica de *keepTypeInformationIn:for:* para la alocación de elementos en el *rawTypesArray,* teniendo en cuenta repeticiones y el primer slot disponible para inserción. Esto nos permitió reutilizar esta lógica en el momento de unificar CCTs, algoritmo que veremos más adelante durante la discusión de *aliasing*.

```
storeType: aType in: typesArray
    <inline: true>
    |typesSize index typeNotStored typeAtIndex |

    typesSize := (objectMemory lengthOf: typesArray)-0.
    index := 0.
    typeNotStored := true.
    [typeNotStored and: [index < typesSize]]
        whileTrue: [
            typeAtIndex := objectMemory followObjField: index ofObject: typesArray.
            typeAtIndex = objectMemory nilObject
                ifTrue: [
                    objectMemory storePointer: index ofObject: typesArray withValue: aType.
                    typeNotStored := false]
                ifFalse: [
                    typeAtIndex = aType
                        ifTrue: [
                            typeNotStored := false].
                    index := index + 1.
                ]
        ]
```

Implementación a nivel VM de la alocación de un elemento dentro de un array de tipos.

Entonces, el método *keepTypeInformationIn:for:* fue extendido para, como se observa en el detalle a continuación:

- Evaluar la clase del objeto asignado
- Validar si se trata de una colección y de serlo
    - Obtener el CCT de la misma
    - Validar si no ha sido asignado y de serlo
        - Validar si ya existe un CCT equivalente y de existir
            - Unificarlos y reemplazarlo en la colección
        - De lo contrario, marcar el CCT como asignado e insertarlo
    - De lo contrario, insertarlo
- De lo contrario, insertar la clase

```
keepTypeInformationIn: types for: anAssignedObject
    | assignedObjectClass assignedObjectClassTag contentTypes instanceInTypes
      collectionIndex typedArrayIndex hasBeenAssigned collectionType storedCCT |
    <inline: true>
    types = objectMemory nilObject
        ifFalse: [
            (self isInstanceOfClassArray: types)
                ifTrue: [
                    assignedObjectClassTag := objectMemory fetchClassTagOf: anAssignedObject.
                    self deny: (objectMemory isForwardedClassTag: assignedObjectClassTag).
                    assignedObjectClass := objectMemory classForClassTag: assignedObjectClassTag.
                    self deny: assignedObjectClass isNil.
                    collectionIndex := self collectionIndexFor: assignedObjectClass.
                    "Then it is a live-typeable collection"
                    (collectionIndex  > -1)
                        ifTrue: [
                            typedArrayIndex := self typedArrayIndexFor: collectionIndex.
                            contentTypes := self contentTypesOf: anAssignedObject with: typedArrayIndex.
                            contentTypes = objectMemory nilObject
                                ifFalse: [
                                    "Index of 'isAssigned' variable within CCT is 2."
                                    hasBeenAssigned := objectMemory followObjField: 2 ofObject: contentTypes.
                                    hasBeenAssigned = objectMemory trueObject
                                        ifTrue: [
                                            instanceInTypes := self is: contentTypes in: types.
                                            instanceInTypes ifFalse: [
                                                "Belongs to another variable, we just add reference"
                                                self storeType: contentTypes in: types.
                                            ]
                                        ]
                                        ifFalse: [
                                            "Index of 'collectionType' variable within CCT is 1."
                                            collectionType := objectMemory followObjField: 1 ofObject: contentTypes.
                                            collectionType = objectMemory nilObject
                                                ifFalse: [
                                                    storedCCT := self findCollectionContentType: collectionType in: types.
                                                    (storedCCT = objectMemory nilObject)
                                                        ifFalse: [
                                                            self merge: storedCCT with: contentTypes.
                                                            self replaceContentType: storedCCT
                                                                in: anAssignedObject
                                                                givenTypedArrayIndex: typedArrayIndex.
                                                        ]
                                                        ifTrue: [
                                                            self setAssigned: contentTypes.
                                                            self storeType: contentTypes in: types.
                                                        ]
                                                ]
                                        ]
                                ]
                        ]
                        ifFalse: [
                            self storeType: assignedObjectClass in: types.
                        ]]].
```

Implementación a nivel VM de keepTypeInformationIn:for: de LiveTyping.

Las instancias de *CCT* se construyen a partir del mensaje de clase *new.* El *TypedArray* es el que crea la instancia del *CCT* y el tipo de la colección nunca se le pasa como colaborador (se envía el mensaje *'new'* sin parámetros), sino que se utilizan los contextos de ejecución para

detectar la colección de origen. Esta decisión es para mantener al *CCT* desacoplado del *TypedArray*, que es una solución de compromiso al no haber sido implementado en *Array*.

```
initialize
|currentContext iteration continue|
    currentContext := thisContext sender.
    iteration := 0.
    continue := true.
    [iteration < 10 and: currentContext notNil and: continue] whileTrue: [
        (currentContext receiver isKindOf: Collection) ifTrue: [
            collectionType := currentContext receiver class.
            continue := false.
            ].
        currentContext := currentContext sender.
        iteration := iteration + 1.
        ].
    contentTypes := Array new:10.
    isAssigned := false.
    ↑ self.
```

Método 'initialize' de instancia de CollectionsContentType.

Vemos que esta implementación busca en hasta 10 contextos padres por la primera colección que haya desencadenado en su creación y asigna en su variable *collectionType* al tipo de la colección. Esta información se utilizará luego para proponer mejoras en el *tooling* de Cuis.

NOTA: para leer más sobre por qué la existencia de TypedArrayCollection y TypedArray son una solución de compromiso dirigirse al Apéndice I: Desafíos.

3. Captura de Tipos Paramétricos

Investigamos las distintas implementaciones de *Collection* de Cuis y resultó que varias de ellas están basadas en Arrays, como por ejemplo: *OrderedCollection* y *Set*. Estas implementaciones guardan los elementos en un *Array* interno, utilizando el método *at:put:* como mecanismo. Para este subconjunto, queda bien claro qué métodos debemos interceptar: únicamente el *at:put:* de *Array.*

Decidimos en este trabajo, ahondar primero en las colecciones basadas en *Array,* para luego buscar la generalización del problema. Utilizamos la estructura de *OrderedCollection* como ejemplo. En el apéndice I detallamos todas las implementaciones analizadas, para la prueba de concepto, se les aplicó un patrón de diseño *proxy* (Gamma et al., 1994) tanto a *Array* como *OrderedCollection*. Se creó el objeto *TypedArray* en la imagen encargado de intervenir el

at:put:. El *TypedArray* es un *proxy* del *array* común, que además recolecta información de tipos en el *CCT*.

TypedArrayCollection, hereda de *OrderedCollection,* creada para contener inicialmente el impacto de modificar a *Array* por *TypedArray:* existen múltiples instancias de *OrderedCollection* vivas en la imagen, y la solución debería contemplar que el sistema dinámicamente se actualice a usar *TypedArray*. Esta nueva colección posee los mismos colaboradores que su padre.

La intervención del *at:put:* en la imagen luce de la siguiente manera:

```
at: anIndex put: aValue

    |innerReturn valueClass specialObjectsArray liveCollections classIndex isLiveCollection |
    innerReturn := innerArray at: anIndex put: aValue.
    valueClass := aValue class.
    specialObjectsArray := Smalltalk specialObjectsArray.
    liveCollections := specialObjectsArray at: 66.
    classIndex := 0.
    isLiveCollection := liveCollections anySatisfy: [ :class |
        classIndex := classIndex + 1.
        class = valueClass
    ].
    isLiveCollection
        ifTrue: [
            | typedArray |
            typedArray := aValue instVarAt: ((specialObjectsArray at: 67) at: classIndex) + 1.
            types add: typedArray contentTypes.
        ]
        ifFalse: [
            types add: valueClass.
        ].
    ↑innerReturn.
```

Implementación de mensaje at:put: en TypedArray.

Podemos ver que la primera colaboración es enviar el mensaje *at:put:* al colaborador interno *(innerArray)* del *TypedArray*, el arreglo que se está enmascarando. Los siguientes pasos corresponden a recolectar el tipo del elemento que se desea agregar, debiendo prestar atención si lo que se agrega es una colección con *LiveTyping + generics (la llamaremos LiveTypingCollection)* o no.

- Para detectar que el elemento es de tipo *LiveTypingCollection* utilizamos el *specialObjectsArray*. En nuestra implementación, la posición 66 contiene un arreglo con todas las colecciones con *LiveTyping* soportadas.
- Si el elemento no es una *LiveTypingCollection*, se agrega la clase de la instancia al *CCT* del *TypedArray*.
- Si el elemento a agregar es una *LiveTypingCollection*, se busca entonces su *CCT* asociado. Vimos recién que la implementación de *TypedArray* es la que referencia al *CCT*, y no la colección: esta decisión es por diseño, dándole la responsabilidad al arreglo tipado de mantener actualizado los tipos.
  Dado que en la solución final, el código antes mostrado debería ejecutarse en la VM, la cual maneja punteros a objetos, la opción que decidimos implementar, es agregar un nuevo arreglo al *specialObjectsArray* (en la posición 67) que posee el índice de la variable que representa el *TypedArray* dentro de la *LiveTypingCollection*. Esto nos permite independizarnos de la jerarquía y las variables que cada colección posea. Además, esta implementación se puede utilizar perfectamente desde la VM.

En la VM, debemos realizar la misma búsqueda del índice dentro de la colección que contiene el *TypedArray*. El algoritmo para encontrar ese índice se puede ver a continuación, notar que en el código a continuación, el segundo array del *specialObjectsArray* se conoce como TypedArrayIndexes.

```
typedArrayIndexFor: collectionIndex

    | index typedArrayIndexes |
    typedArrayIndexes := objectMemory splObj: TypedArrayIndexes.
    index := objectMemory followField: collectionIndex ofObject: typedArrayIndexes.
    ^ objectMemory integerValueOf: index.
```

Implementación a nivel VM de la búsqueda del TypedArray para una colección.

NOTA: la lógica se encuentra repetida en la VM e imagen ya que TypedArray y TypedArrayCollection tienen sentido en una prueba de concepto. En la generalización, la intervención del at:put: u otra técnica para lograr el mismo objetivo se realizaría completamente en la VM.

Sabiendo que el objeto siendo analizado por *LiveTyping* es una colección y el índice entre sus variables de instancia del array ahora tipado, el proceso de obtener la referencia al *CCT* es simplemente navegar la estructura del *TypedArray* y obtener su colaborador interno.

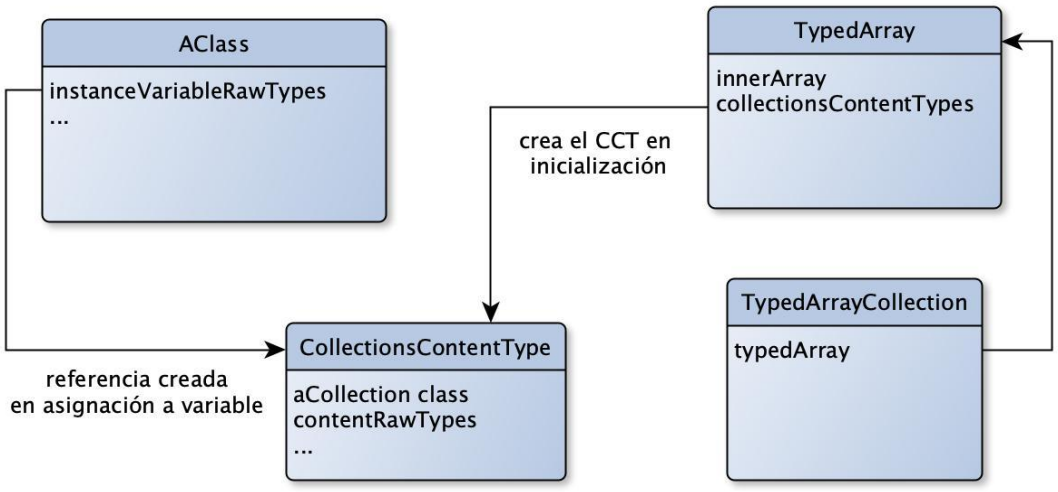
```
contentTypesOf: aCollection with: typedArrayIndex

    | typedArray contentTypes |
    typedArray := objectMemory followObjField: typedArrayIndex ofObject: aCollection.
    contentTypes := objectMemory nilObject.
    typedArray = objectMemory nilObject
        ifFalse: [
            contentTypes := objectMemory followObjField: 1 ofObject: typedArray "Obtain
'types' variable within the TypedArray instance"
        ].
    ^ contentTypes.
```

Implementación a nivel VM de la búsqueda del CCT correspondiente a una colección.

4. Relación entre Tipos Paramétricos y sus Usos

Una variable puede ser asignada con distintas colecciones, y una colección puede asignarse a distintas variables que potencialmente pueden ser de distintas clases. El diseño de la solución, junto con el *CCT*, deben contemplar esta cardinalidad.

La opción elegida, es que la instancia de colección conozca al *CCT* y en el caso de asignarla, guardar en el *rawTypes* el *CCT* asociado a esa instancia. El *CCT* continúa siendo actualizado a medida que los elementos se agreguen a la colección, y la relación con la variable permanece viva mediante el *rawTypes*. En caso de que el *garbage collector* borre la referencia a la colección, *LiveTyping* no se ve impactado ya que no la requiere para el cómputo, sólo el *CCT*.



LiveTyping para colecciones decorando cada instancia con un CCT, para el caso de *TypedArrayCollection.*

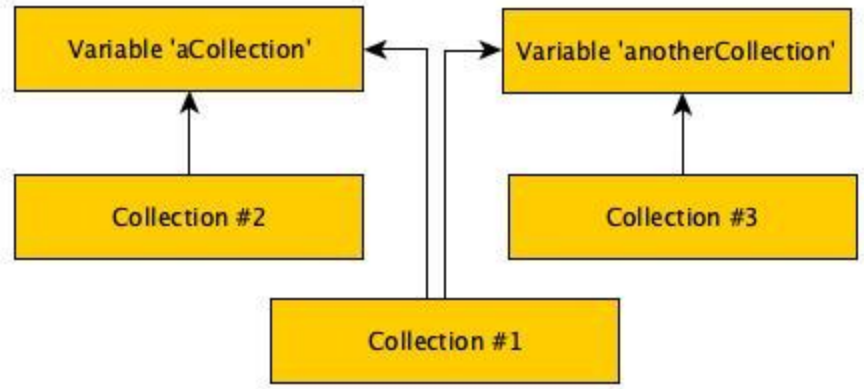
Podemos ver cómo se relacionan los dos mundos *(la asignación de cada variable y la colección viva que muta sus elementos internos)*: el *rawTypes* de *AClass* referencia al *CCT* de la colección, y esta lo mantiene actualizado conforme sus elementos cambian.

5. Aliasing

Vimos en la sección anterior que el *CCT* se mantiene actualizado en la intervención del método *at:put:*, sin embargo, no hemos tratado el problema de las múltiples colecciones y la diferencia de magnitudes entre capturar clases versus capturar información de múltiples instancias.

La relación entre *CTTs* y variables es n a m, es decir, una colección puede haber sido asignada a múltiples variables, y cada variable puede ser asignada por múltiples colecciones. Además, los *raw types* que *LiveTyping* utiliza tienen 10 posiciones de tamaño por defecto, siendo este tamaño de arreglos, suficiente para obtener *tooling* relevante al desarrollo.

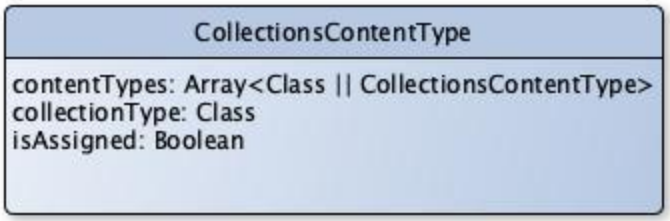
El siguiente gráfico, ejemplifica un posible escenario que *LiveTyping* debe poder manejar, una instancia de colección referenciando más de una variable.



Ejemplos de relaciones entre colecciones y variables.

A priori, en el *instanceVariableRawTypes* veríamos dos *CCTs* para cada variable. Ahora bien, supongamos que las colecciones #1 y #2 son del mismo tipo y que en este ejemplo en particular, tiene sentido considerar que representan el mismo conjunto de *generics*. Llamamos a esto *aliasing de generics:* los *raw types* ahora deben administrar eficientemente potencialmente gran cantidad de objetos del mismo tipo.

Recordemos primero de qué está compuesto el *CCT* en nuestro modelo.



Colaboradores internos del CollectionsContentType.

Para una variable, debemos tener al menos un *CCT* por tipo de *Collection*, ya que el *CCT* posee como colaborador interno a la clase de la colección en la variable *collectionType*. Esto implica por ejemplo, que un *CCT* de *Set* no es unificable con uno de *OrderedCollection*, y deben mantenerse separados en el *rawTypes* de la variable a la que la colección se asigna. Esta decisión es consistente con la implementación de *LiveTyping* donde clases polimórficas (ej: *Float* e *Integer*) se mantienen como elementos distintos en el *rawTypes* y se unifican al utilizarse en la imagen. Veremos más adelante en la sección de *tooling* cómo se realiza ese proceso de unificación.

El primer algoritmo de *aliasing* implementado dio como resultado que rápidamente se forman grupos de variables todas relacionadas entre sí, la información de tipos se generaliza muy rápidamente y resulta inefectivo.

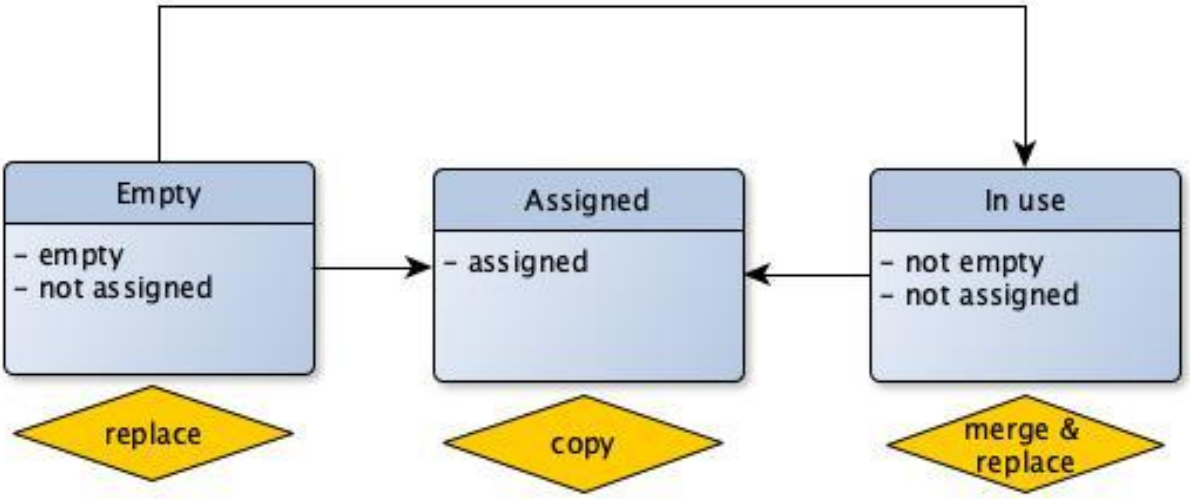
Para poder romper con la propagación de *aliasing*, manteniendo la capacidad de sumarización, se decidió acotar el *aliasing* sólo dentro de una variable. Es aquí donde surge la necesidad del colaborador *'isAssigned'*: El *CCT* sólo puede unificarse con *CTTs* de la primera variable a la que se asigna, sin poder propagar *aliasing* a las otras potenciales variables en las que puede estar referenciado.

```
setAssigned: contentTypes
    <inline: true>
    |trueObject|
    trueObject := objectMemory trueObject.
    objectMemory storePointer: 2 ofObject: contentTypes withValue: trueObject.
```

Implementación a nivel VM del cambio de estado de un CCT a 'isAssigned := true'.

Podemos pensar entonces en que la colección pasa por tres estados:

- Vacía: aún no se encuentra asignada, y no posee ningún elemento.
- En uso: continúa sin asignarse, pero ya posee elementos.
- Asignada: la colección se encuentra asignada a una variable.



Diferentes estados por los que una colección puede pasar en el proceso de asignación a variable

Supongamos que tenemos en el *rawTypes* un *CCT* para *OrderedCollection* y se quiere asignar a la misma variable una nueva *OrderedCollection,* el algoritmo implementado para reducir la cantidad de CCTs ejecuta los siguientes pasos, si la colección:

- **Está vacía y no fue asignada:** se reemplaza su *CCT* por el de la colección ya asignada.
- **No está vacía y no fue asignada:** se unifican ambos *CCTs* en el *rawTypes* y la nueva colección referencia al *CCT* unificado.
- **Ya fue asignada a otra variable:** se agrega una nueva entrada en el *rawTypes* y no se unifica con el *CCT* ya existente, rompiendo el *aliasing* entre variables.

En caso de que el *rawTypes* no posea aún un *CCT* para ese tipo de colección, la nueva colección a asignar se guardará en el arreglo, y marcará como *assigned*.

A continuación, se detallan las extensiones de la VM para aplicar el algoritmo de *aliasing.* El primer gráfico, describe cómo se reemplaza el *CCT* de la colección a asignar por el *CCT* que ya se encuentra en el *rawTypes:*

```
replaceContentType: cct in: aCollection givenTypedArrayIndex: typedArrayIndex

    | typedArray |
    typedArray := objectMemory followObjField: typedArrayIndex ofObject: aCollection.
    objectMemory storePointer: 1 ofObject: typedArray withValue: cct.
```

Implementación a nivel VM del reemplazo de CCTs.

El siguiente gráfico, refleja el proceso de unificación, que es necesario cuando el *CCT* a reemplazar tiene tipos internamente:

```
merge: contentTypes with: anotherContentTypes

    |typesArray anotherTypesArray ithType size index|

    "Index of 'contentTypes' variable within CCT is 0."
    typesArray := objectMemory followObjField: 0 ofObject: contentTypes.
    "Index of 'anotherContentTypes' variable within CCT is 0."
    anotherTypesArray := objectMemory followObjField: 0 ofObject: anotherContentTypes.
    size := (objectMemory lengthOf: anotherTypesArray)-0.
    index := 0.

    [index < size]
        whileTrue: [
            ithType := objectMemory followObjField: index ofObject: anotherTypesArray.

            (ithType = objectMemory nilObject)
                ifFalse: [
                    self storeType: ithType in: typesArray.
            ].

            index := index + 1.
    ]
```

Implementación a nivel VM de la unificación de CCTs.

Vale recordar que tanto la unificación como el reemplazo se realizan contra una instancia de *CCT* ya capturada y equivalente, es decir, que ya es parte de un *rawTypesArray* de *LiveTyping* y posee el mismo tipo base (*OrderedCollection*, por ejemplo). El siguiente método muestra cómo la VM realiza ese chequeo:

```
findCollectionContentType: collectionType in: anArrayOfTypes

    |size cct cctClass index objectAtIndex notFound objectAtIndexClassTag objectAtIndexClass collectionClass |

    size := (objectMemory lengthOf: anArrayOfTypes)-0.
    index := 0.
    notFound := true.
    cct:= objectMemory nilObject.
    cctClass := objectMemory splObj: ClassCollectionsContentType.

    [notFound and: [index < size]]
        whileTrue: [
            objectAtIndex := objectMemory followObjField: index ofObject: anArrayOfTypes.
            objectAtIndex = objectMemory nilObject
                ifFalse: [
                    objectAtIndexClassTag := objectMemory fetchClassTagOf: objectAtIndex.
                    self deny: (objectMemory isForwardedClassTag: objectAtIndexClassTag).
                    objectAtIndexClass := objectMemory classForClassTag: objectAtIndexClassTag.
                    self deny: objectAtIndexClass isNil.

                    objectAtIndexClass = cctClass
                        ifTrue: [
                            collectionClass := objectMemory followObjField: 1 ofObject: objectAtIndex.
                            collectionClass = collectionType
                                ifTrue: [
                                    notFound := false.
                                    cct := objectAtIndex.
                                ]
                        ].
                ].
                index := index + 1.
        ].

    ^ cct.
```

Implementación a nivel VM de la búsqueda de CCT equivalente dentro de un rawTypesArray.

Consideraciones

Unificar un *CCT* sólo si su colección no fue asignada previamente puede generar escenarios donde el *rawTypes* es rápidamente completado, por ejemplo:

- Sean $v1,...,v10$ variables que fueron asignadas cada una con `*OrderedCollection with: 10*`.
- Sea v otra variable que es asignada con $v1,...,v10$. Luego, el *rawTypes* de v va a estar lleno y compuesto por 10 CCTs distintos.
    - Si $v1,...,v10$ eventualmente evolucionan de manera distinta, está bien que la información permanezca separada, ya que al inspeccionar su *LiveTyping*, encontraremos información más detallada.
    - Si al contrario, $v1,...,v10$ no terminan teniendo *contentTypes* distintos, entonces v pierde la posibilidad de unificar esa información y dejar lugar a recopilar

información de tipos de otras colecciones que no han podido guardarse ya que no hay más espacio en el *rawTypes.*

Existen dos soluciones a este problema:

- Agrandar el *rawTypes* para que deje de ser un problema: el *re-size* puede ser dinámico y en base a una condición sobre el contenido del *rawTypes.*
- Efectuar una unificación más inteligente al entender que $v1,\ldots,v10$ son unificables en ese caso.

La implementación y análisis sobre estos dos enfoques de *aliasing* dinámico se encuentran por fuera de este trabajo de investigación. Desde un punto de vista de diseño, cada objeto debería mantener el encapsulamiento y evitar hacer accesible colaboradores internos como podría ser una colección; si fuese necesario devolverla, se debería retornar una copia. En ese caso, la copia podría ser devuelta sin ser asignada previamente y el proceso de unificación podría realizarse por *LiveTyping*.

Tooling

Una vez resuelto el problema de capturar los tipos de colecciones, el desafío es hacer uso de esa información para mejorar las herramientas ofrecidas por *LiveTyping*. Nos concentramos sobre 2 de las herramientas donde se evidencia con mayor claridad las consecuencias de no soportar esos objetos: la visualización de tipos (o *tooltips*) y el *autocompletion*.

Consideremos el siguiente ejemplo, la clase *SomeClass* con una única variable de instancia llamada *variable,* la cual se asigna en la inicialización de la instancia de *SomeClass*:

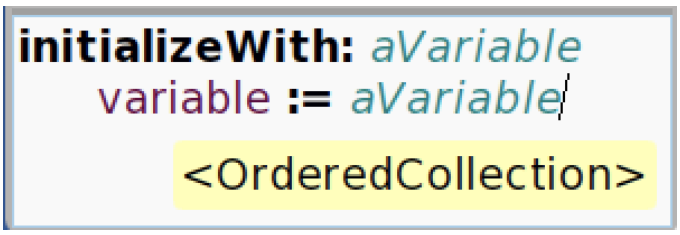
Clase 'SomeClass', 1 variable inicializada en 'initializeWith:'.

Supongamos que además, en la imagen se ejecuta el siguiente conjunto de colaboraciones:

```
create
| aCollection |
aCollection := OrderedCollection new.
aCollection add: 'hello'.
aCollection add: 'there'.
SomeClass new: aCollection
```

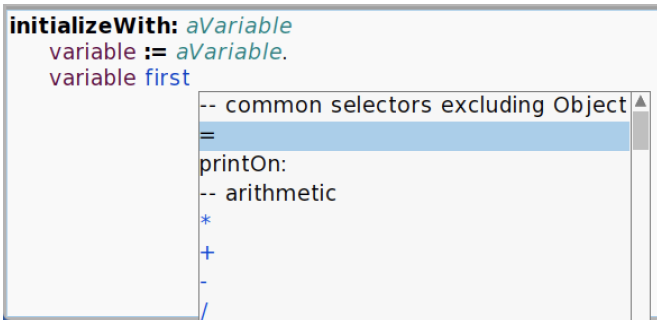
Colaboraciones que influyen en el LiveTyping en el colaborador `variable` de SomeClass.

Como habíamos mencionado, al visualizar los tipos, *LiveTyping* sólo muestra la clase de la colección involucrada, y no los elementos que contiene, que ya sabemos que son instancias de *String:*



Información de tooltip de LiveTyping sin soporte de colecciones.

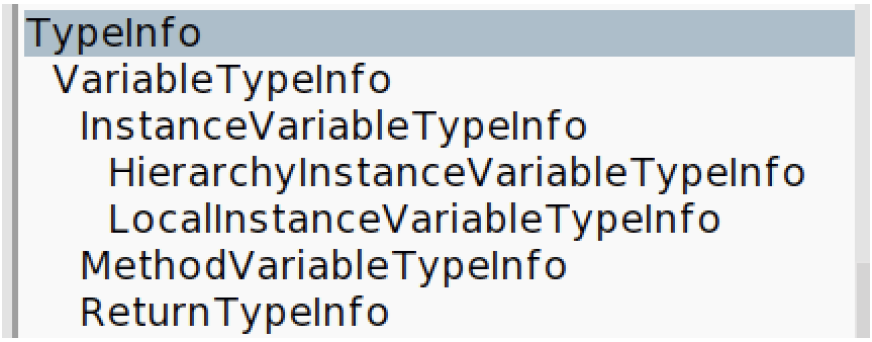
Por otro lado, las sugerencias de *autocomplete* no analizan los tipos internos de la colección y por lo tanto no se listan mensajes válidos para un String.



Sugerencias de autocomplete sin soporte de LiveTyping para colecciones.

Es importante mencionar que los *rawTypes* son procesados por LiveTyping a nivel imagen en distintos tipos de objetos o *type info* para poder simplificar el uso en el tooling. En este nivel, se distinguen las fuentes de estos rawTypes, es decir, si pertenecen a variables de instancia locales o jerárquicas, a parámetros o a tipos de retorno. También se agrupan para tratar en un solo objeto a todos los parámetros de un método o variables de instancia de una misma clase. Estos objetos son los responsables, entre otros, de encontrar supertipos y selectores comunes, como también de decidir cómo se mostrarán esos valores al inspeccionarse.

La actual jerarquía de *TypeInfo* luce así:



Jerarquía de TypeInfo

El procesamiento en la imagen de los *rawTypes* permite optimizar el tooling definiendo dónde realmente se hace uso de los tipos recolectados o se debe intervenir con lógica especial para el caso. Por ejemplo, si consideramos el mensaje *new*, la recolección de tipos de retorno será irrelevante al utilizarse en todas las clases. Luego, en este caso en vez de retornar lo recolectado, *LiveTyping* indica que el tipo de retorno es dinámico, es decir, dependiente del contexto. Para el caso de *new*, se debe analizar la instancia de clase que se pidió instanciar. Los tipos dinámicos *(DynamicType)*, en conjunto con las clases regulares, forman parte de la base de *LiveTyping* para proveer *autocompletion* y *tooltips*. Existen además otros métodos

donde es necesario devolver el tipo de la clase, como *Object >> #class*, o la instancia como es el caso de *Object >> #yourself.*

Jerarquía de DynamicType

Hasta el momento, el tooling de *LiveTyping* trabaja pura y exclusivamente con clases de una dimensión, para soportar las colecciones y sus *generics*, debemos considerar el anidamiento de clases y una representación de múltiples dimensiones. Explicaremos a continuación los cambios realizados para poder obtener mensajes de tipos que incluyan la información de *generics* y sugerencias de compleción de código acordes a estos.

Tooltips

Los primeros pasos sobre tooling fueron en la funcionalidad más básica, donde inmediatamente se evidencia el impacto del nuevo soporte para colecciones: los *tooltips* que se muestran al examinar métodos y variables de todo tipo. Estos, utilizan el *type info* para imprimir los tipos, haciendo un análisis de supertipos comunes y utilizando el formato:

```
<super tipo # tipo 1 | tipo 2 | ... | tipo n>
```

Dicho formato resultó adecuado para un modelo con *generics* considerando que cualquiera de los tipos ahora podría anidarse. Es decir:

```
<super tipo<T> # tipo 1<A> | tipo 2<B> | ... | tipo n<C>>
```

En este caso, mientras el supertipo original representa al supertipo común entre los tipos 1, 2 a n; T representará el supertipo común entre A, B y C, cuya representación se hará en el mismo formato de manera recursiva. Esto significa que para representar los tipos necesitamos:

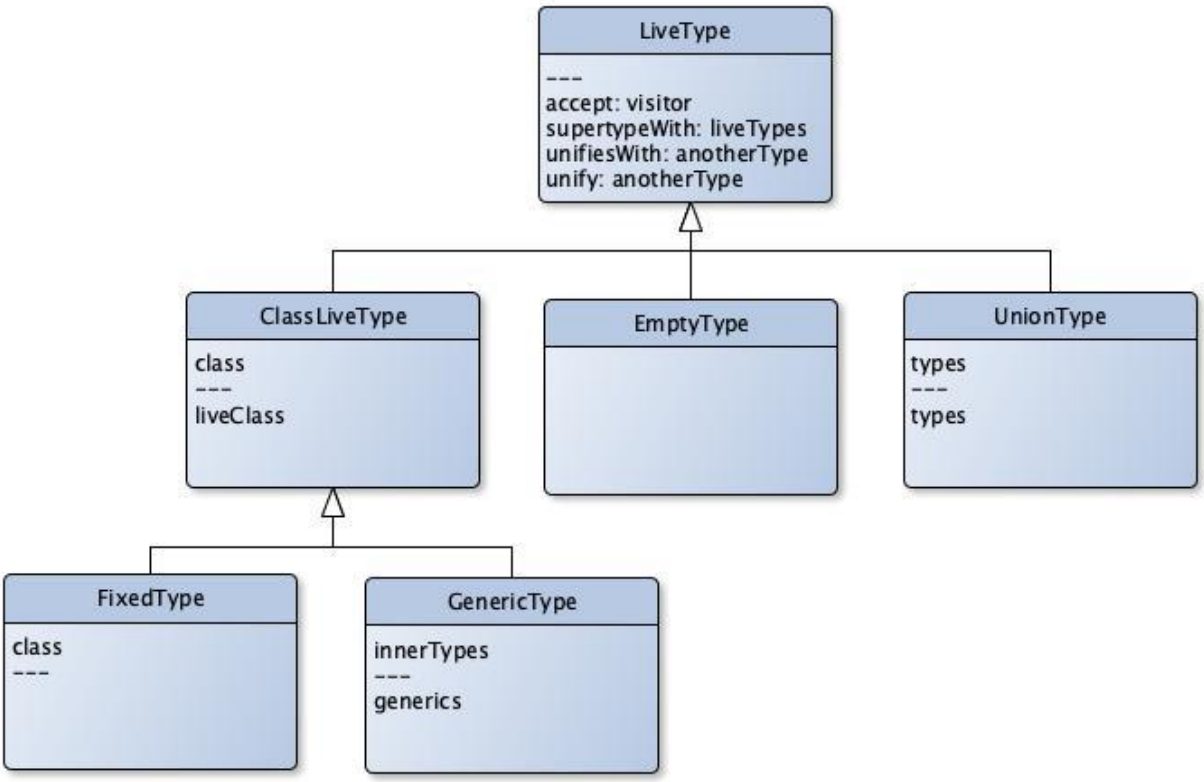
- Calcular los supertipos comunes de manera recursiva, para el tipo base y los posibles *generics.*
- Mantener referencia a los tipos originales, incluyendo posibles *generics.*

NOTA: debe resolverse también el hecho de que todo el tooling hasta el momento maneja clases y ahora existen CCTs en los rawTypes. Adicionalmente, debe considerarse los problemas de aliasing cuando una misma colección se referencia desde distintos puntos, que

se acepta almacenar múltiples CCTs y resolver la multiplicidad de CCTs cuando distintos tipos de colecciones han sido asignadas.

La solución para ambos problemas fue la adaptación de los *rawTypes* en *liveTypes*, una nueva jerarquía de tipado con el concepto de *generics*, unificación y supertipos. Es decir, en vez de trabajar a nivel *tooling* con los objetos en el *rawTypes (clases y CCTs)*, generamos a partir de ellos objetos que representan los tipos correspondientes. Estos objetos conocen como calcular su supertipo y como unificarse contra otros tales objetos, además de poder ser visitados (Gamma et al., 1994). Los *liveTypes* están conformados por:

- **FixedType**: Un tipo regular o clase.
- **GenericType**: Los nuevos tipos paramétricos, que además de tener una clase principal tendrá un set de tipos internos.
- **EmptyType**: Representa la ausencia de tipado, es decir, un *rawTypes* vacío.
- **UnionType**: Representa la unificación de tipos.



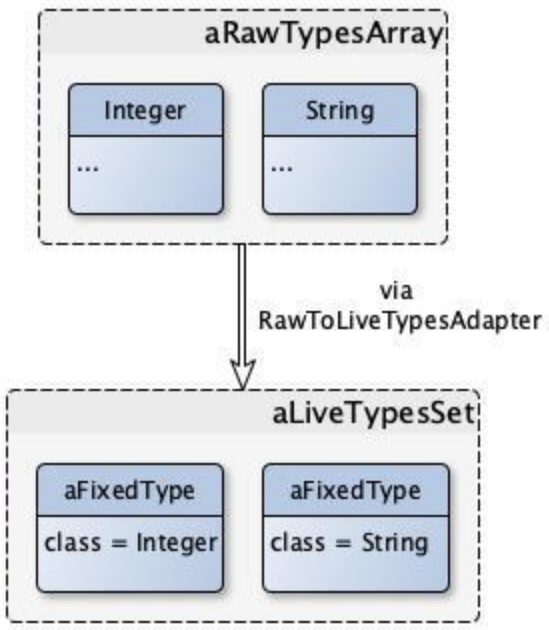
Jerarquía de LiveTypes

De esta manera, mientras una clase guardada en un *rawTypes* se transforma en un *FixedType*, un *CCT* se transforma en un *GenericType* y de haber múltiples, se unifican en uno solo. A su vez, al analizar los supertipos podremos representar la unión de tipos y la ausencia

de información. El proceso comienza en los *TypeInfo*, transformando los *rawTypes* a *liveTypes* mediante un adapter:

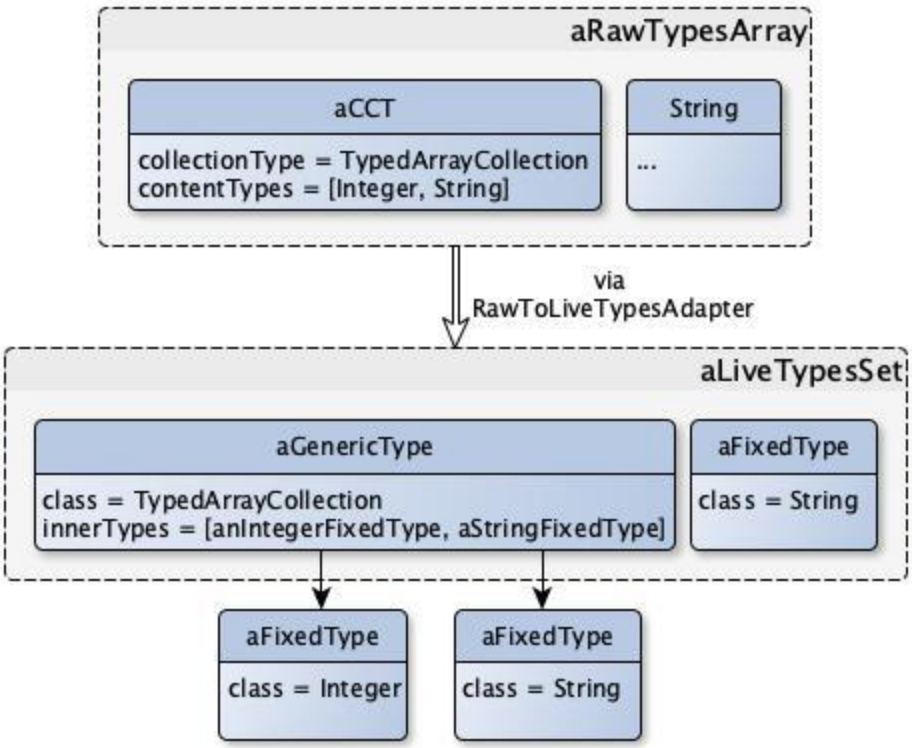
```
printTypesOn: aStream upTo: aNumberOfTypes

    | liveTypes |

    liveTypes := RawToLiveTypesAdapter new adapt: self types.
    (LiveTypesPrinter on: aStream) print: liveTypes upTo: aNumberOfTypes.
```

A continuación, se ejemplifican cómo son las transformaciones de *rawTypes* a *liveTypes* que realizan instancias de *RawToLiveTypesAdapter*. El primer escenario es la transformación de clases que no poseen *generics* internamente:



Transformación de clases sin generics a instancias de FixedType

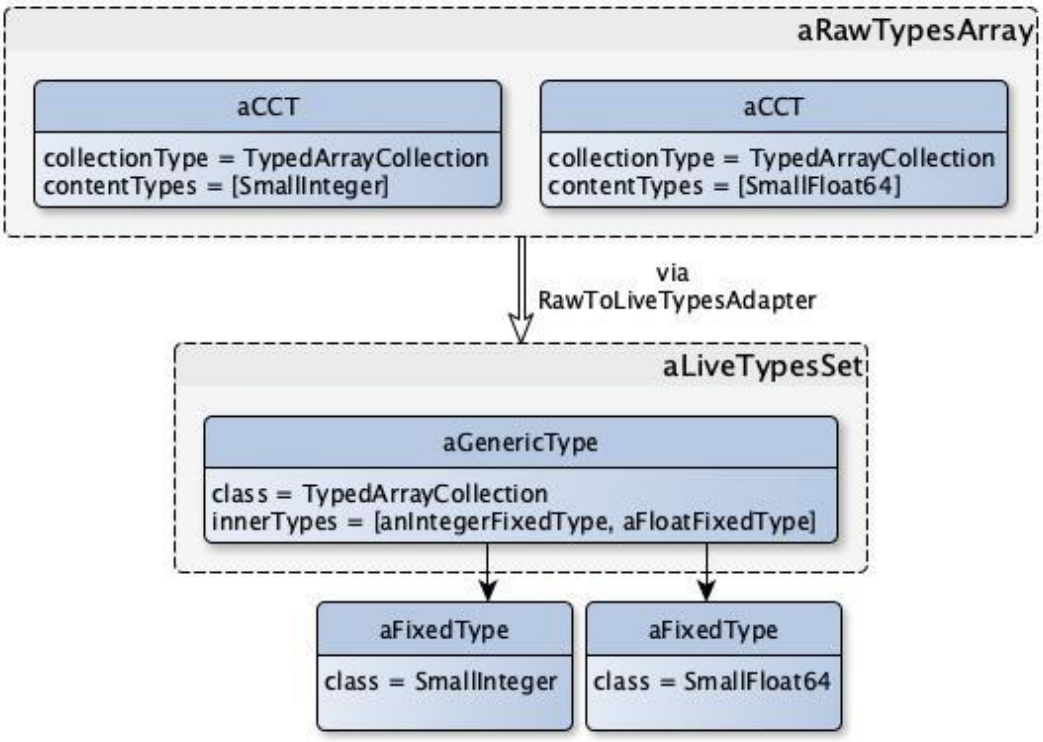
El siguiente escenario detalla cómo se realiza la transformación de un *CCT* a una instancia de *GenericType:*



Transformación de un CCT y una clase sin generics a instancias de GenericType y FixedType respectivamente

Como ya mencionamos, la transformación de *rawTypes* a *liveTypes* es recursiva, los elementos del contentTypes de la instancia de *CCT* fueron transformados por instancias de *FixedTypes.* Si el *CCT* referencia a *CCTs*, cada uno será transformado a una instancia de *GenericType* y el *liveTypesSet* va a poseerlos anidados de la misma manera que sucede con el caso de *Integer* y *String* previamente diagramado.

Debido al *aliasing*, un *rawTypes* puede contener varios *CCTs* sobre la misma colección. En esta situación, los *LiveTypes* se encontrarán unificados:

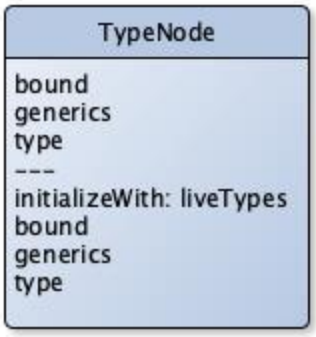


Transformación de dos CCTs a una instancia de GenericType

Vale notar que la unificación se produce al ser la misma colección para ambos *CCTs* (mismo *collectionType)*, caso contrario, se mantienen como *GenericsTypes* separados, aún perteneciendo a la misma jerarquía de colecciones. En aliasing vimos que es un caso válido para nuestra implementación, ya que si la colección fue asignada previamente a otra variable, no se unificará a los *CCTs* de otras variables a las que se asigne.

Con esta nueva información disponible diseñamos un algoritmo de impresión de tipos para los *tooltips*. El objeto encargado de imprimir la información de tipos se llama *LiveTypesPrinter.*

La recursividad del proceso de impresión y la necesidad continua de calcular supertipos, nos llevó a la creación de un árbol de tipos como estructura para optimizar dicho proceso. En este árbol cada nodo contiene el supertipo de su nivel, los tipos utilizados para su cálculo y el siguiente nodo interno correspondiente a sus *generics*, de existir. Llamamos a la clase *TypeNode* y su estructura se detalla a continuación:



Clase *TypeNode*

El *TypeNode* se construye recursivamente a partir de todo los *LiveTypes* recolectados en el *initializeWith:*. Utilizando el ejemplo de la sección de tooltips podemos mapear su estructura con la del *TypeNode*:

```
<super tipo<T> # tipo 1<A> | tipo 2<B> | ... | tipo n<C>>
```

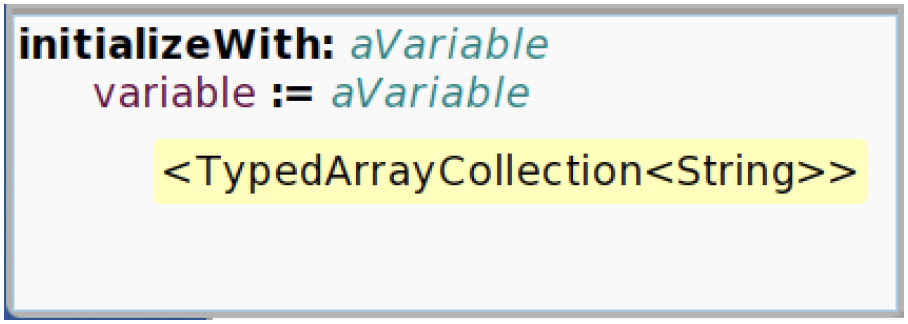
- **`super tipo`** es *aTypeNode type*
- **`<T>`** es el *TypeNode bound*. Notar que esta estructura también es un *TypeNode*.
- **`tipo 1<A> | tipo 2<B> | ... | tipo n<C>`** son los *TypeNode generics.* Al igual que **`<T>`**, es una estructura de *TypeNodes.*

Como resumen, para imprimir los tipos y generar el *tooltip,* la siguiente secuencia de pasos ocurre:

1. Los *rawTypes* se transforman en *liveTypes*
2. Los *liveTypes* se analizan y organizan en un árbol
3. Se imprime el supertipo principal, visitando su *liveType*
4. Se imprime recursivamente el siguiente nodo del árbol
5. Se imprime cada tipo utilizado en el cálculo del supertipo, con un separador, visitando su *liveType*

Cabe destacar que la visita en el caso de UnionTypes y GenericType, también resulta en la impresión recursiva de sus partes.

Con *LiveTyping* para generics, el tooltip para el ejemplo previamente descrito agrega ahora la información de los tipos internos:



Tooltip de LiveTyping con soporte de colecciones.

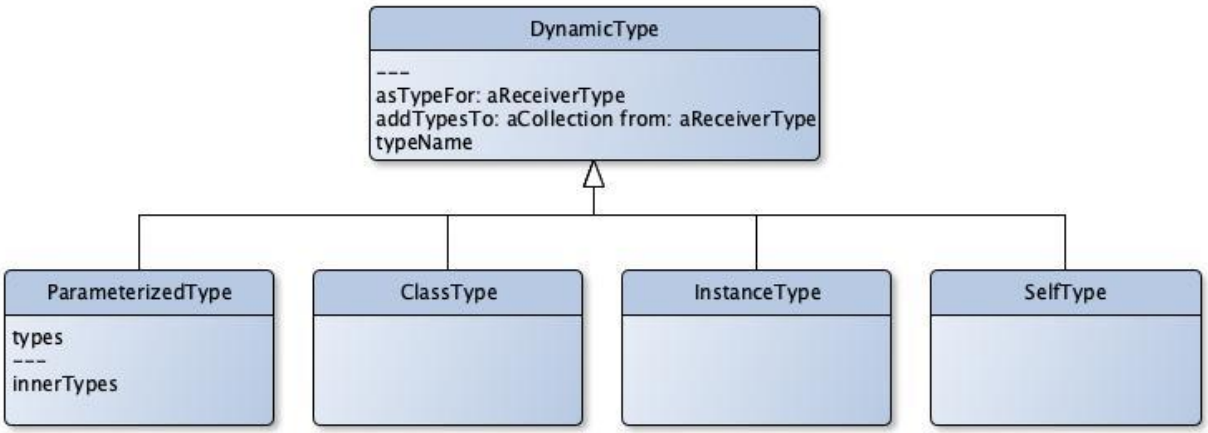
Autocompletion

Para este caso nos enfocamos en analizar las sugerencias al trabajar con selectores sobre variables tipadas como colecciones. El objetivo fue lograr que las sugerencias reflejaran los tipos de contenido de las colecciones, es decir, que las sugerencias luego de obtener un elemento de una colección sean selectores del tipo del elemento.

La dificultad en este caso, es que la información requerida se encuentra dentro de los *CCT* y no basta con evaluar los *rawTypes*, de hecho necesitaremos los *liveTypes* generados para los *tooltips* para obtener los tipos con *generics* resueltos y unificados.

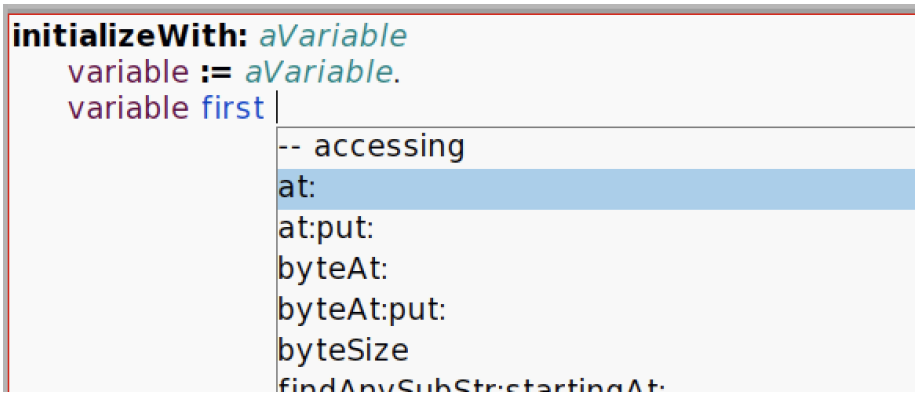
Utilizamos la lógica existente para la jerarquía de *DynamicType* definida en la introducción de tooling: así como el método *new* indicará que se debe evaluar la instancia de retorno, los métodos de las colecciones que sean paramétricos, es decir, tengan *generics*, deberán indicarlo. Esta distinción es importante: tener una clase con generics no significa que sus métodos también los tengan. De hecho, sólo un subconjunto de los métodos de una colección, como *first* o *last*, exponen el *generic* de la misma.

Introducimos a la jerarquía de tipos dinámicos un objeto que representa la presencia de un *generic* y la necesidad de interpretar el contenido del *GenericType* que se observará al adaptar los *rawTypes* de la información de tipos para el método. Llamamos a este objeto *ParameterizedType*, es decir, el “T” de nuestro *generic*.



Jerarquía extendida de DynamicType con ParameterizedType

Luego, al interpretarse la información de tipos de un método como *first* y recibir nuestro *ParameterizedType*, se obtienen los tipos del *GenericType* correspondiente para generar las sugerencias de selectores.



Autocomplete de String detectado por LiveTyping para colecciones.

Es decir, la información de tipos de retorno para los métodos de colecciones que expongan su parametrización explícitamente indicarán esto conteniendo una instancia de *ParameterizedType.* De esta manera, cuando al analizar los tipos de retorno de un mensaje nos encontremos con tal instancia, pasaremos a analizar el *CCT* del receptor del mensaje en primer lugar. En nuestro ejemplo, el análisis del mensaje "*first*" resultará en un *ParameterizedType* y entonces, el análisis del tipado capturado para "*variable*", que como vimos anteriormente es una colección de *Strings*.

Si bien fueron necesarios algunas modificaciones a cómo *LiveTyping* implementa el *autocomplete*, en especial en la clase *MessageNode*, lo más relevante a tener en cuenta en este trabajo es que sobreescribiendo el método *createMethodReturnTypeInfoOf:* en la clase que contempla *return types* paramétricos es suficiente:

- *createMethodReturnTypeInfoOf:* se envía cada vez que el *autocomplete* se encuentra en funcionamiento.
- En caso de que el método de parámetro sea un selector con return paramétrico, este mensaje deberá devolver *ParameterizedType,* dentro de un *ReturnTypeInfo* correspondiente a la jerarquía de *TypeInfo.*

```
createMethodReturnTypeInfoOf: aMethod
    | returnTypeInfo parameterizedSelectors |
    parameterizedSelectors := Array with: #first with: #atLast:.
    returnTypeInfo := super createMethodReturnTypeInfoOf: aMethod.

    (parameterizedSelectors includes: aMethod selector) ifTrue: [
        | parametricType |
        parametricType := ParameterizedType with: returnTypeInfo types.
         ↑ ReturnTypeInfo of: aMethod are: (Array with: parametricType).
        ].

    ↑returnTypeInfo
```

createMethodReturnTypeInfoOf sobrescrito en SequenceableCollection class para contemplar los returns generics de #first y #atLast:.

Generalización

Si bien este trabajo comenzó como *"Anotación automática de tipos para colecciones en ambientes dinámicos"*, en su desarrollo descubrimos que el problema de tipos genéricos, no abarca sólo al problema de colecciones. Nos enfocamos entonces en construir de manera *bottom-up* una solución que pueda extenderse a tipos que no sean colecciones. Nuestros descubrimientos se detallan a continuación.

Es importante distinguir algunos ejemplos de uso de *generics* para entender las variantes que deberá cubrir una implementación general.

El caso más abundante es equiparable a nuestro *TypedArray*, es decir, una clase que internamente tiene uno o más elementos de cualquier tipo. Por ejemplo, si pensamos en un *optional* o *maybe*, su contenido tendrá un tipo genérico.

```
public final class Optional<T> {
    /**...*/
    private static final Optional<?> EMPTY = new Optional<>();

    /**
     * If non-null, the value; if null, indicates no value is present
     */
    private final T value;

    /**...*/
    private Optional() {
        this.value = null;
    }

    /**...*/
    public static<T> Optional<T> empty() {...}

    /**...*/
    private Optional(T value) { this.value = Objects.requireNonNull(value); }
```

Ejemplo básico de uso de generics en lenguaje estáticamente tipado (Java).

Por otro lado, podemos pensar en un nodo en el contexto de un árbol, donde tendríamos dos opciones:

- **Opción 1:** Referenciar al padre

```java
public abstract class Node<T> {

  private Node<T> parent;
  private T value;

  public abstract Node<T> addValue(T value);
}
```

Ejemplo de uso de generics en lenguaje estáticamente tipado (Java) con un generics anidado.

- **Opción 2:** Referenciar a los hijos

```java
public abstract class Node<T> {

  private Collection<Node<T>> children;
  private T value;

  public abstract Node<T> addValue(T value);
}
```

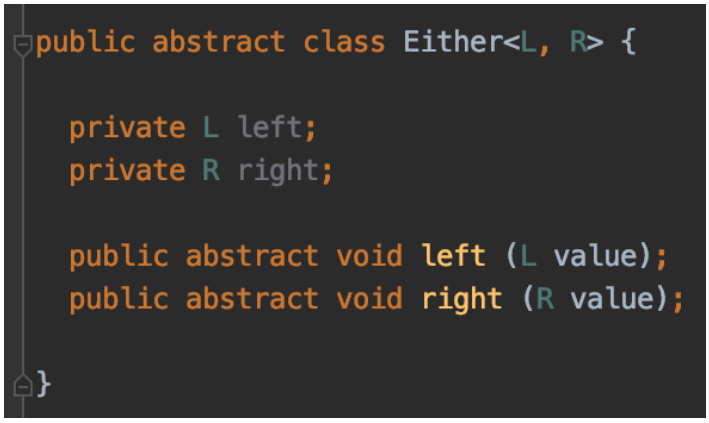
Ejemplo de uso de generics en lenguaje estáticamente tipado (Java) con varios generics anidados.

En el primer caso, los nodos se encuentran enlazados indicando otro nodo donde nuestro *generic* vuelve a repetirse. En el segundo, los nodos se encuentran enlazados indicando grupos de nodos, así la variable *children* puede desarrollarse en:

- *children* es `Collection<N>`
- `N` es `Node<T>`
- `T` queda definido por la instanciación de la clase

Es decir, la captura de tipos para un *generic* no siempre será lineal como en el caso de *TypedArray* si no que puede ser delegada en otro tipo paramétrico. Este caso no nos es ajeno, ya que es el caso con *TypedArrayCollection* donde es el *TypedArray* interno quien define la parametrización del tipado que termina utilizando la colección. La gran mayoría de las colecciones analizadas que utilizan *arrays* internamente se encuentran bajo estas condiciones.

Debemos destacar también un caso particular del analizado para nodos, el caso de múltiples *generics*. Por ejemplo, si consideramos una clase que represente la mónada *Either* tendremos dos tipos genéricos en vez de uno.



Ejemplo de uso de múltiples generics en lenguaje estáticamente tipado (Java).

A continuación se detallan dos posibles generalizaciones para el problema de *LiveTyping* en tipos genéricos.

Soporte de *generics* vía *method instrumentation*

El trabajo realizado sobre *TypedArray* podría ser extendido a cualquier clase que indique la presencia de un tipo paramétrico y un punto de captura sobre el mismo, sea un valor directo u otro tipo paramétrico. En este último caso, como serían las colecciones o clases similares, se debería poder indicar la propagación del generic en vez de la captura inmediata.

Recordemos que para soportar la solución actual es necesario:

- El listado de clases paramétricas que indica la instrumentación a la VM *(specialObjectsArray),*
- El *CollectionContentTypes* donde se colectan los tipos para su posterior uso,
- La intervención del método *at:put:* como punto de captura diferida de tipos.

Para extender y generalizar la solución, entonces debemos:

- Incluir dinámicamente nuevas clases al listado de clases paramétricas ó hacer bien conocido ante la VM alguna variable particular de la clase que indique si es genérica, y cuáles son sus variables con dichas propiedades,
- Generalizar el *CCT* en un colector de múltiples tipos genéricos, llamemoslo *GenericsTypeCollector (GTC)*,
- Intervenir dinámicamente métodos específicos donde se observen los tipos genéricos, o como discutimos antes, propagar los tipos.

Los algoritmos de aliasing, seguirán siendo relevantes para consolidar la información obtenida de distintas instanciaciones, siendo lo desarrollado en este trabajo relevante: se guardarán múltiples *GenericsTypeCollectors* cuando sus clases de origen sean distintas o

cuando se encuentren previamente asignados a una variable. De no estar asignados y ser la misma clase de origen, se unificarán en una instancia.

Una vez implementada la generalización, queda como responsabilidad del usuario definir que una clase es genérica para que *LiveTyping* comience a recolectar con *generics.* Deberá indicar además cuál es la fuente del o los tipos diferidos: un método y parámetro o una variable de instancia de donde se propaga. *LiveTyping* agrega entonces una variable de instancia con un *GenericsTypeCollector* a la clase, incorpora la clase al listado de clases paramétricas y, dependiendo de la fuente del *generic,* recompila el método indicado para colectar el tipo del parámetro indicado dentro del *GenericsTypeCollector* o generar un enlace entre este último y su contraparte que vive dentro de la variable que propaga el *generic.*

Como ejemplo, si el usuario define que se debe recolectar el tipo diferido del método *add:, LiveTyping* agrega el *GTC* al objeto dueño del mensaje y recompila el método agregándole la llamada al *GTC*. Un nota de color es que la UI podría no mostrar el conjunto de colaboraciones inyectadas si así lo desea.

`add: aValue`
`  value := aValue.`
	`add: aValue`
	`  genericsTypeCollector collect: aValue from: 0.`
	`  value := aValue.`

Ejemplo de recompilación de método de recolección incluyendo la variable de instancia de GenericsTypeCollector.	

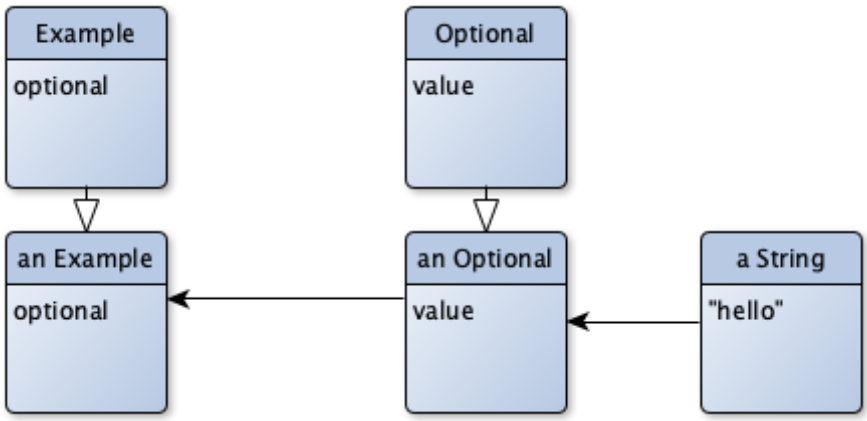
Vale mencionar que *GenericsTypeCollector* tendrá una lógica similar a la que incluímos en el *at:put:,* debe identificar si es necesario guardar la clase del valor o si se trata de justamente otra clase paramétrica y por lo tanto, debe guardar la instancia de *GenericsTypeCollector*, para manejar *generics* anidados. Deberá indicarse también la enumeración del *generic* a capturar dado que a diferencia del *CollectionsContentType*, trabaja con múltiples *generics*.

Consideremos los casos de *Array* y *OrderedCollection* para ilustrar las posibles circunstancias. En el caso de *Array* que, requiere una implementación customizada en la VM, va a ser necesario indicar la parametrización y la intervención del método *at:put:* sobre el segundo parámetro; esto será cierto para toda implementación realizada de manera nativa. En el caso de *OrderedCollection*, en vez de requerir la intervención del método *add:*, sólo requiere indicar la propagación por la variable de instancia *array*. La solución entonces debe enlazar el *type collector* de *OrderedCollection* con el del *Array* en cuestión.

El siguiente caso a considerar es el de múltiples *generics*. En nuestro trabajo nos concentramos sobre colecciones simples, pero por ejemplo, si consideramos la clase *Dictionary*, su tipo sería *Dictionary<T>* donde T es un tipo de *Association* que a su vez tendría tipo *Association<K,V>*, donde *K* es el tipo de sus claves y *V* el tipo de sus valores. Para

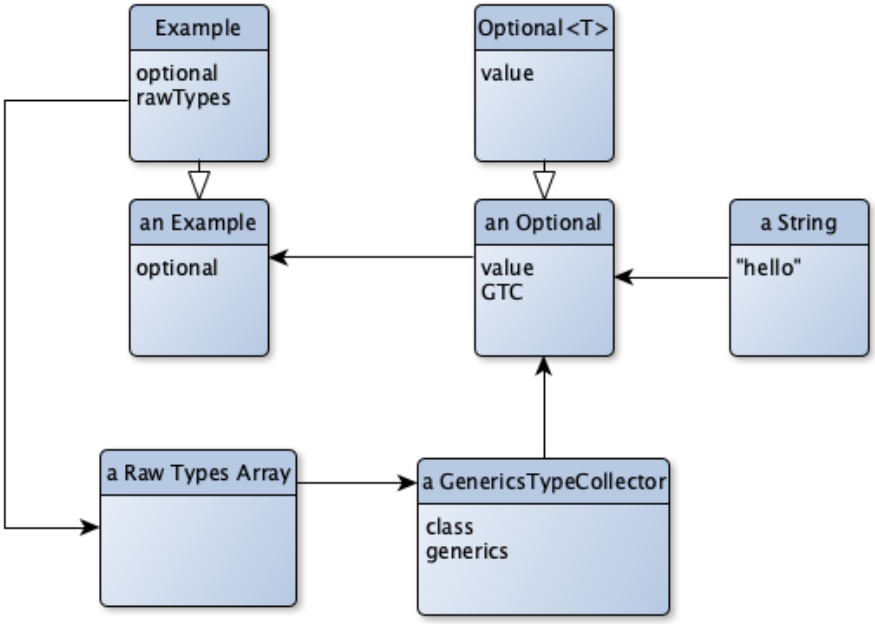
soportar este escenario, el objeto *GenericsTypeCollector* debe soportar múltiples *generics*, es decir, arrays internos de recolección. Cuando se indique los puntos de introspección, se deberá indicar simplemente sobre qué *generic* se realizará la recolección.

Volviendo al caso de *Optional*, supongamos el siguiente uso en una clase *Example* y la relación entre sus instancias:



Ejemplo del uso de una instancia de Optional con el String “hello”.

Al considerar a *Optional* como una clase paramétrica, sus instancias deben referenciar a un *GenericsTypeCollector*, que se utilizará en el *rawTypes* de la clase (Example) de la instancia (an Example) que lo referencie.

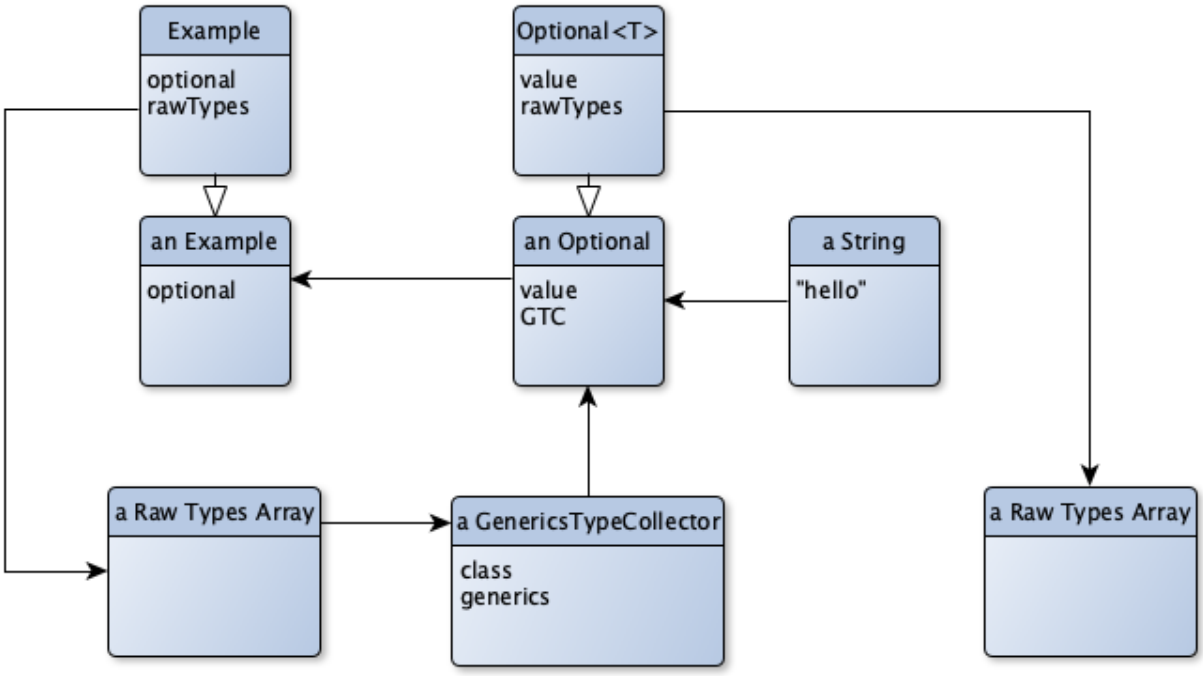


Extensión de LiveTyping para ejemplo del uso de una instancia de Optional con el String “hello”.

Soporte de *generics* vía *variable instrumentation*

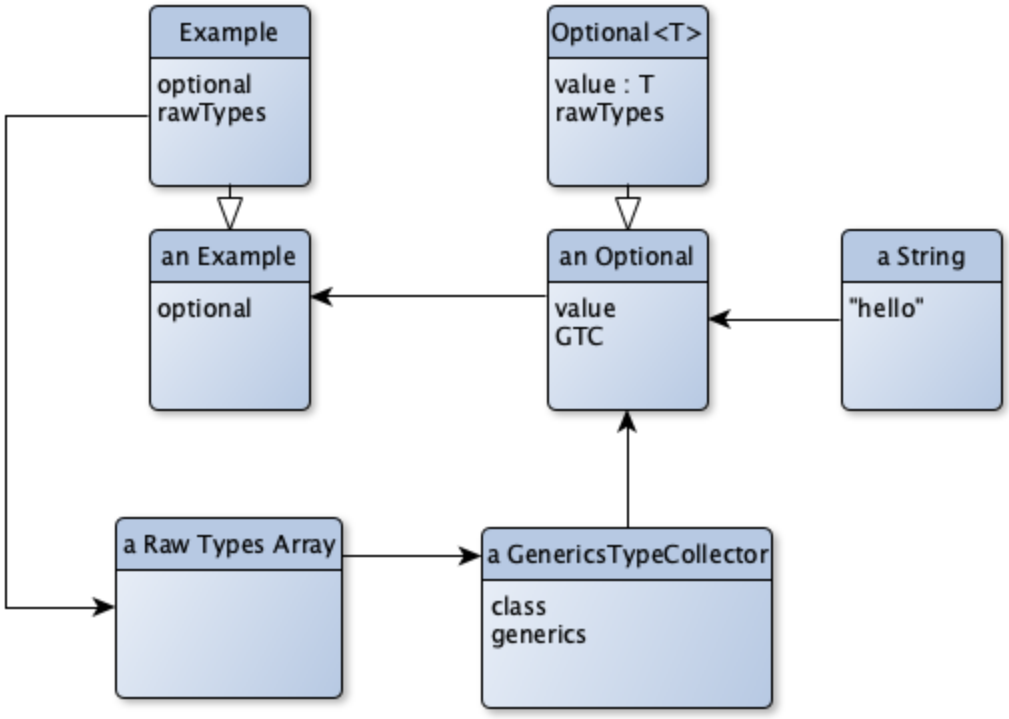
Alternativamente, y considerando que nuestra propuesta requiere la anotación de los tipos paramétricos en las clases y los mensajes donde se usan, analizamos si es posible la anotación de la parametrización sólo en las variables de instancia afectadas. Dado que *LiveTyping* actualmente introspecta la asignación de dichas variables, se podría distinguir que se trata de variables con tipado paramétrico y guardar los tipos en el *GenericsTypeCollector* de su instancia en vez del *instanceVariableRawTypes* de su clase. De esta manera, no sería necesario modificar el código fuente de ningún mensaje ya que utilizaríamos la introspección existente de *LiveTyping,* que es en la asignación.

Consideremos nuevamente el ejemplo anterior respecto a *Optional*: esta clase en sí misma mantiene un *rawTypes* para su variable *'value'*.



LiveTyping para ejemplo del uso de una instancia de Optional con el String "hello".

Sin embargo, es justamente la variable *value* quien es paramétrica y por la cual mantenemos un *GenericsTypeCollector*. Si esta información fuese explícita, podríamos evitar el *rawTypes* y al asignarse el *String* a la variable *value*, capturar ese valor dentro del *GenericsTypeCollector en la instancia*.



LiveTyping con anotación de variables genéricas para caso con el String "hello".

Como mencionamos anteriormente, la instrumentación de un mensaje donde capturar el tipo paramétrico ya no sería necesaria. Utilizando los mecanismos actualmente disponibles en *LiveTyping* e información explícita respecto a la presencia de tipos paramétricos, la captura de los tipos en el *GenericsTypeCollector* sucedería al momento de asignar sus valores.

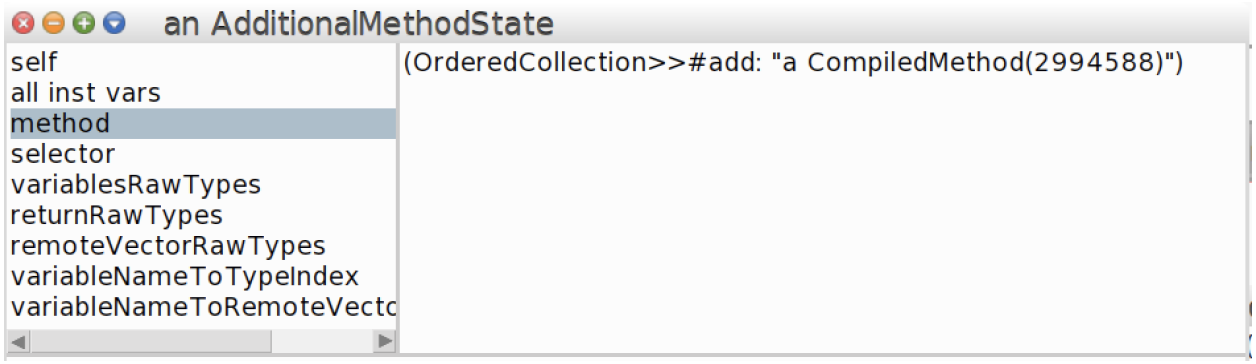
Descubrimiento automático

Hasta ahora sólo hemos discutido mecanismos manuales con los que un usuario identifica una clase paramétrica e instrumenta *LiveTyping*. Sin embargo, podrían usarse algunas heurísticas para sugerir la instrumentación:

- La presencia de un *rawTypes* sin slots vacíos y que la superclase común a ellos resuelva a *Object.*
- El uso interno de variables de instancia con clases paramétricas donde los *generics* cumplan con el punto anterior

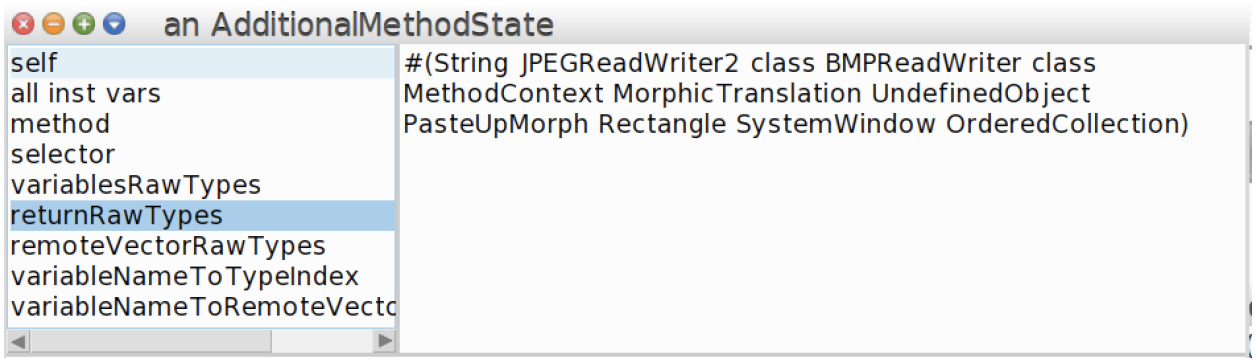
El punto principal es el hecho de completar un *rawTypes* sin una superclase específica. Si bien esto puede ocurrir cuando efectivamente estamos modelando con *Object*, es lo que sucede internamente con *LiveTyping* ante la presencia de *generics* sin soporte alguno: la variación de tipos es tal que rápidamente se completan los 10 espacios disponibles para recolectar tipos y el análisis de los mismos solo puede ofrecernos a *Object* como superclase.

Por ejemplo, si miramos el *methodDictionary* de OrderedCollection, podemos ver como se debe llegar a *Object* para encontrar el supertipo en común:



AdditionalMethodState para el método OrderedCollection>>#add

Si inspeccionamos su *returnRawTypes,* podremos ver que la colección se utiliza para tareas disímiles, siendo esta información la que se puede utilizar para sugerir si la clase debe o no ser *generic.*



ReturnRawTypes para el método OrderedCollection>>#add

Conclusion

Este trabajo presenta las bases para un modelo general de soporte para *generics* en *LiveTyping*, comprobando el valor del mismo en el manejo de las ampliamente utilizadas *Collections*. Para mantener las optimizaciones sobre *Arrays*, sin embargo, es necesario reconsiderar su implementación a nivel VM.

Aunque puede decirse que este enfoque se basa en la anotación de los tipos paramétricos de forma manual al requerir conocer las clases, métodos y variables donde se utilizan, una vez instrumentado por el desarrollador de las clases con *generics*, su funcionamiento es automático para los usuarios de las mismas. En un lenguaje estáticamente tipado, el usuario de las clases con *generics*, debe continuamente explicitar el tipo del generic que utiliza mientras que con *LiveTyping* esto no sería necesario.

La prueba de concepto realizada para la colección *TypedArrayCollection* nos permitió elaborar el modelo de manera *bottom-up*, permitiéndonos validar problemáticas como la explosión de instancias de *CCTs* de manera temprana e iterar sobre algoritmos para su mitigación. De esta manera, pudimos evaluar instancias concretas respecto al uso de tipos paramétricos, las colecciones, y desarrollar un modelo general a partir de esa experiencia.

Por otro lado, nuestra solución mantiene la esencia de *LiveTyping*: depende exclusivamente de la ejecución de código y, como antes mencionamos, es transparente para los usuarios de clases paramétricas.

Trabajo Futuro

A continuación presentaremos las áreas de posible expansión sobre nuestro trabajo, desde problemáticas abiertas y oportunidades de mejora, hasta modelos alternativos.

Implementación y validación del modelo general

El principal punto de exploración abierto respecto a nuestro trabajo es la implementación del modelo de anotaciones para tipos paramétricos. Si bien es una evolución de lo validado en nuestro trabajo, resta ponerlo a prueba fehacientemente contra casos concretos, poner a prueba nuestro algoritmo de aliasing y evaluar el impacto en performance que acarrearía el modelo.

Instrumentación de la jerarquía de colecciones

En segundo lugar, y una vez implementado el modelo anterior, se podría instrumentar para toda la jerarquía de colecciones actualmente disponible en Cuis Smalltalk. Esto permitiría su distribución y extensiva validación, tanto en performance como experiencia de usuario.

La instrumentación requeriría la anotación de todas las clases de la jerarquía como clases paramétricas, de sus colaboradores internos y todos los mensajes que exponen la parametrización. Adicionalmente, requeriría la instrumentación de *Array* a nivel VM.

Unificación de modelo de tipado para tooling

Una gran oportunidad de mejora pendiente involucra el modelo actualmente utilizado en *tooling* luego del análisis de los *rawTypes*. Los tipos paramétricos generaron la necesidad de extender el modelo más allá de las simples clases presentando una oportunidad para unificar tanto el modelo de *LiveType* como el de *DynamicType*. Esto simplificaría el modelo del tooling y gran parte del código sobre el mismo ya que permitiría tratar con cualquier clase de tipo de manera polimórfica. A su vez, resta explorar si los usos de *DynamicType* son casos específicos de tipos paramétricos.

Extensión de tooling con información de tipos paramétrico

Por otro lado, si bien en nuestro trabajo exploramos la visualización de los tipos en tooltips y el *autocompletion* al enviar mensajes parametrizados a clases paramétricas, resta explorar más oportunidades de mejora en el tooling. Por ejemplo, evaluar el impacto de la nueva información disponible en *refactorings* y en la navegación de tipos.

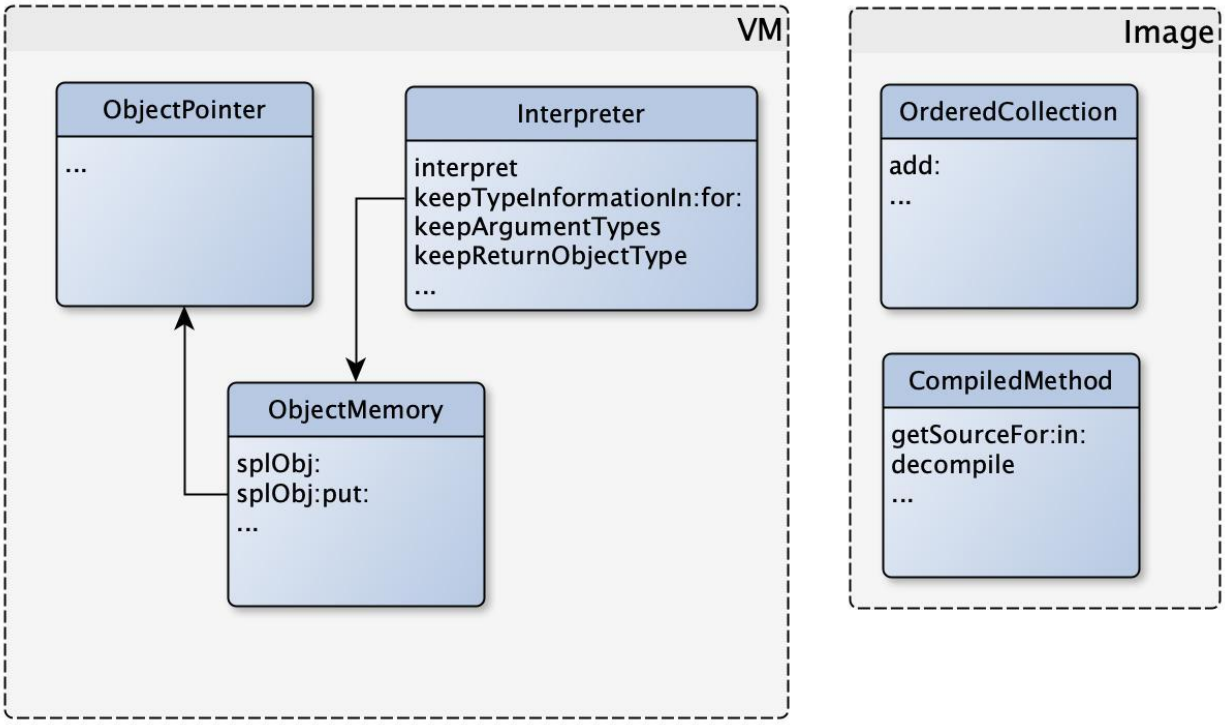
Exploración de modelo basado en inferencia de tipos

Por último, queda pendiente la exploración de resoluciones alternativas al problema de tipos paramétricos y su uso en colecciones. En particular, es interesante considerar modelos de inferencia de tipos que, en conjunto con la información actualmente disponible gracias a *LiveTyping,* podrían restringirse a un conjunto limitado de estructuras. Dado que este es uno de los principales problemas de los algoritmos de inferencia y el costo asociado a los mismos, tal modelo podría resultar performante y requerir menos instrumentación para su funcionamiento.

Apéndice I: Desafíos

Collection desconocida para la VM

Como veremos en la sección de Smalltalk 80, el Intérprete de stack realiza la ejecución del *bytecode* de cada método. El object memory a su vez, entiende sobre la memoria y punteros a objetos. Una forma de compartir información entre la VM y la imagen es a través del *special objects array* que veremos en detalle en la próxima sección. A priori, las clases y jerarquías de subclasificación pertenecen al dominio de la imagen y son desconocidas por la VM. Por ejemplo: la VM desconoce la jerarquía de Collection y al procesar un método, tampoco sabe si el sender es una instancia de Collection.



Separación de responsabilidades entre la VM y la imagen

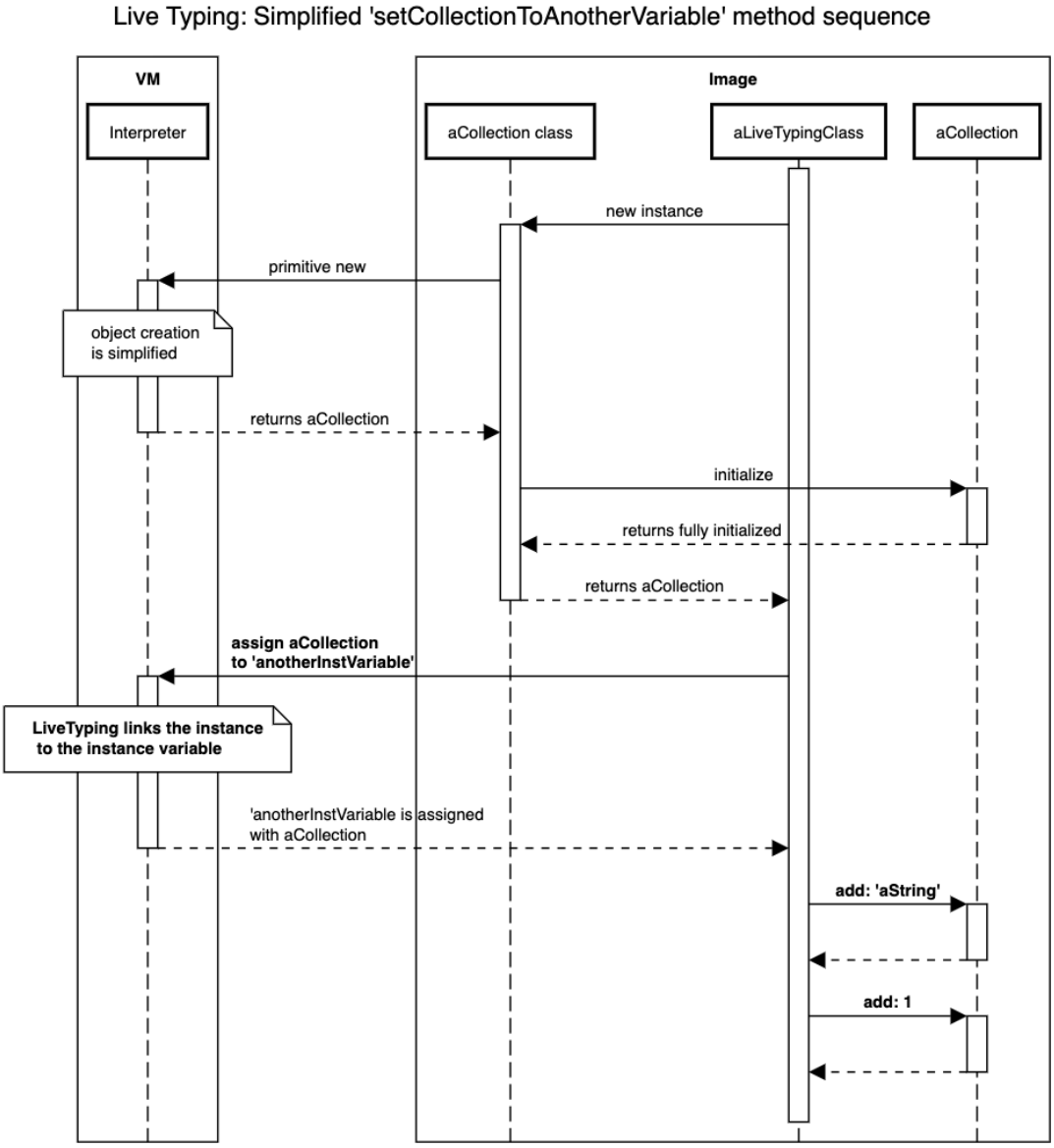
Momento de asignación

LiveTyping recolecta la información de tipos al procesar la asignación de objetos en la VM.

```
setCollectionToAnotherVariable
    anotherInstVariable := OrderedCollection new.
    anotherInstVariable add: 'aString'.
    anotherInstVariable add: 1.
    ↑ anotherInstVariable

<any # SmallInteger | SmallFloat64 | OrderedCollection>
```

Ejemplo de colección asignada a una variable, que contiene un string y un entero.

Como ejemplo, utilizamos el caso de estudio del método *setCollectionToAnotherVariable* en el siguiente diagrama de secuencia.



NOTA: *Cabe destacar que se simplificaron las interacciones entre la VM y la imagen a los efectos de mostrar el trabajo que queremos resolver, un diagrama de secuencia completo debería tener como el intérprete ejecute cada bytecode provisto por la imagen.*

Con *LiveTyping*, desde la VM sólo se puede conocer el tipo de la instancia de *Collection,* pero no sus elementos internos ya que estos se agregan luego de la asignación. La VM al interpretar el *bytecode* del método *add:* desconoce que es el mensaje de una *Collection* y que está relacionada con la variable que asignó algunas operaciones atrás. El escenario se complejiza aún más, cuando la colección se retorna en un método y los elementos se le agregan desde algún otro contexto. Es decir, si bien la creación de una colección, su asignación y la inserción de elementos en la misma son eventos relacionados, son temporalmente independientes a pesar de ser requeridos para determinar y capturar el tipo de la colección.

Type binding diferido

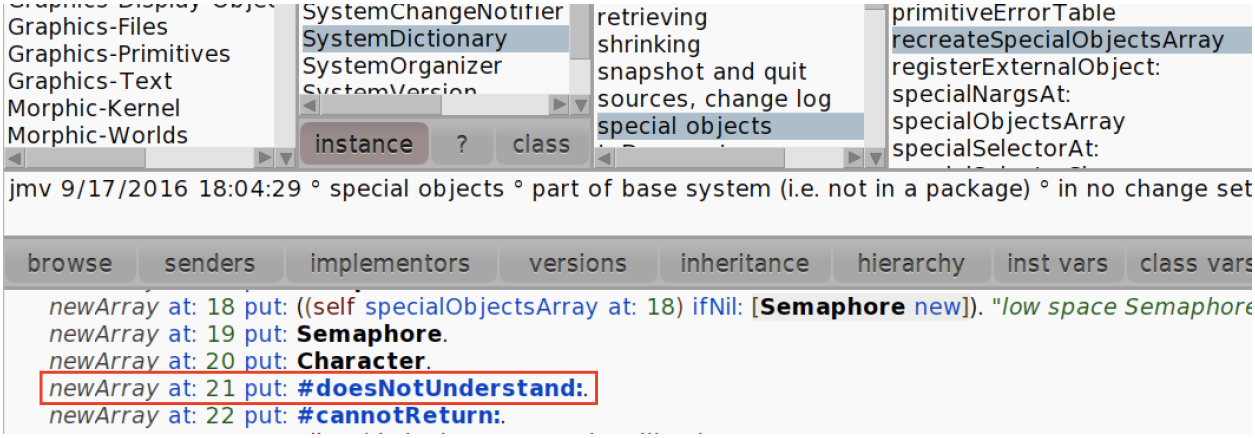
Como corolario, para poder soportar colecciones y el *deferred type binding* de las mismas*,* la solución debe:

1. Conocer todos los tipos de colecciones que se quieran usar en el momento de la asignación.
2. Crear y controlar algún objeto que permita guardar los tipos de los elementos de la colección.
3. Mantener una relación entre ese objeto y la variable a la que se está asignando.
4. Conocer qué métodos agregan elementos a cada colección.
5. Mantener actualizado el objeto que agrega la información de tipos de la colección al procesar cada mensaje que agrega elementos.

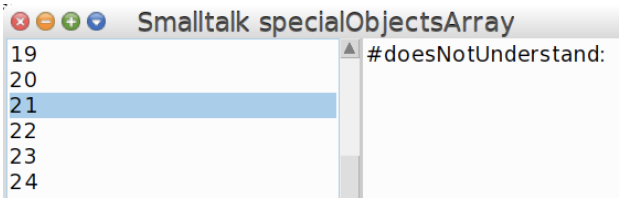
Ahondaremos a continuación, en cada uno de estos ítems.

1. Conocer desde la VM los tipos de las colecciones: *Special Objects Array*

El *specialObjectsArray* es un arreglo definido por la imagen que es bien conocido por la *VM*. Los elementos allí definidos, pueden ser utilizados por la *virtual machine* en su ejecución. Su definición se encuentra en el método *recreateSpecialObjects* de *SystemDictionary.* Como ejemplo, podemos ver que la imagen define en la posición 21 el selector *#doesNotUnderstand:,* utilizado para cuando el receptor no sabe responder un mensaje recibido *(los arreglos en la imagen son 1-based).* La VM entonces, puede acceder al selector yendo al *specialObjectsArray* y accediendo a la posición 20 *(para la VM, los arreglos son 0-based).*

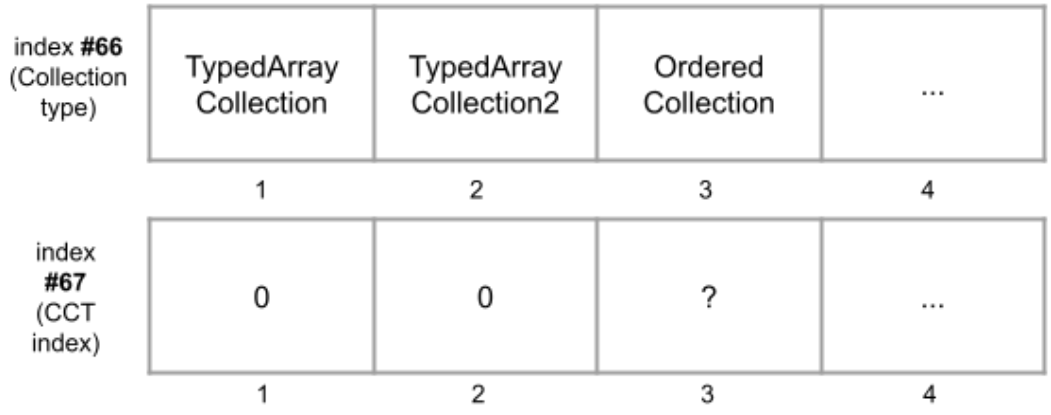


La imagen tiene una única instancia del *specialObjectsArray*.



Podemos entonces agregar un nuevo ítem al arreglo en donde se almacene un arreglo de colecciones que soporten *LiveTyping.* La *VM* puede validar en la asignación, si el objeto a asignar es una colección o no y luego actuar en consecuencia.

De hecho, en la implementación de este trabajo, fueron necesarios dos arreglos, el primero para definir el tipo de cada *Collection* con *LiveTyping*, y el segundo para definir el índice donde encontrar el objeto que contiene los tipos de los elementos dada una instancia de colección. El i-ésimo elemento del primer arreglo se relaciona uno a uno con el i-ésimo elemento del segundo.



Arrays agregados al specialObjectsArray para soporte LiveTyping en colecciones.

Notar que el índice para *OrderedCollection* se encuentra con un signo de pregunta. Como hemos mencionado, el desarrollo de este trabajo se realizó utilizando como ejemplos sólo las primeras dos colecciones: *TypedArrayCollection* y *TypedArrayCollection2.* Explicaremos más adelante los motivos por los que no trabajamos con *OrderedCollection*.

```
collectionIndexFor: aClass

    <inline: true>
    | isNotCollection isCollectionSearchIndex supportedCollectionAtIndex supportedCollections
supportedCollectionsSize collectionIndex|

    isNotCollection := true.
    supportedCollections := objectMemory splObj: SupportedCollections.
    supportedCollectionsSize := (objectMemory lengthOf: supportedCollections)-0.
    isCollectionSearchIndex := 0.
    [isNotCollection and: [isCollectionSearchIndex < supportedCollectionsSize]]
        whileTrue: [
            supportedCollectionAtIndex := objectMemory followObjField: isCollectionSearchIndex
ofObject: supportedCollections.
            supportedCollectionAtIndex = aClass
            ifTrue: [
                isNotCollection := false]
            ifFalse: [
                isCollectionSearchIndex := isCollectionSearchIndex +1].
             ].

    collectionIndex := isCollectionSearchIndex.
    isNotCollection ifTrue: [collectionIndex := -1].

    ^ collectionIndex.
```

Código a nivel VM utilizado para reconocer instancias de colecciones.

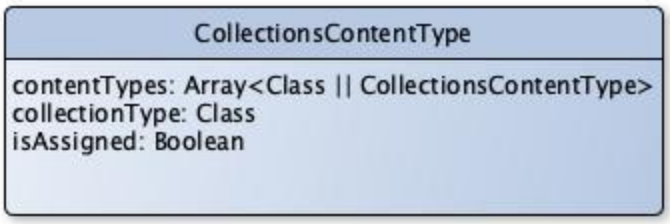
Como se observa en el mensaje anterior, para reconocer si hay que recolectar tipos de una colección, se la busca en el array llamado *SupportedCollections* del *specialObjectsArray*. Para optimizar la performance del algoritmo general, el mensaje no retorna simplemente un boolean: retorna el índice de la clase en el array, de efectivamente encontrarse allí. De esta manera, podemos reconocer colecciones y obtener el índice correspondiente del segundo array en una sola operación.

2. Guardar los tipos de los elementos: *Collections Content Type (CCT)*

Definimos a *CollectionsContentType (CCT),* como la clase responsable de crear las instancias que monitoreen los elementos que se agregan a las colecciones. Este objeto referencia al menos a:

- La clase de la colección,

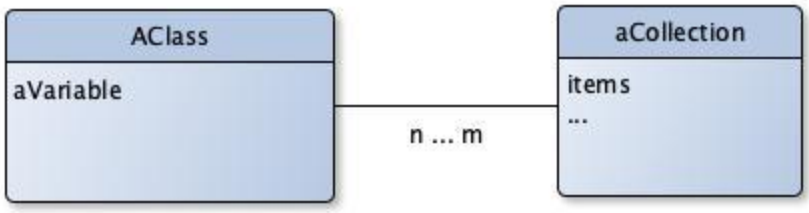
- Una lista de clases y/o *CollectionsContentType*. Cada entrada representa el tipo de algún elemento en la colección, y en el caso de colecciones anidadas, se representa con *CCT*s anidados.



Definición de CollectionsContentType (CCT).

3. Link entre los CCTs y la variable

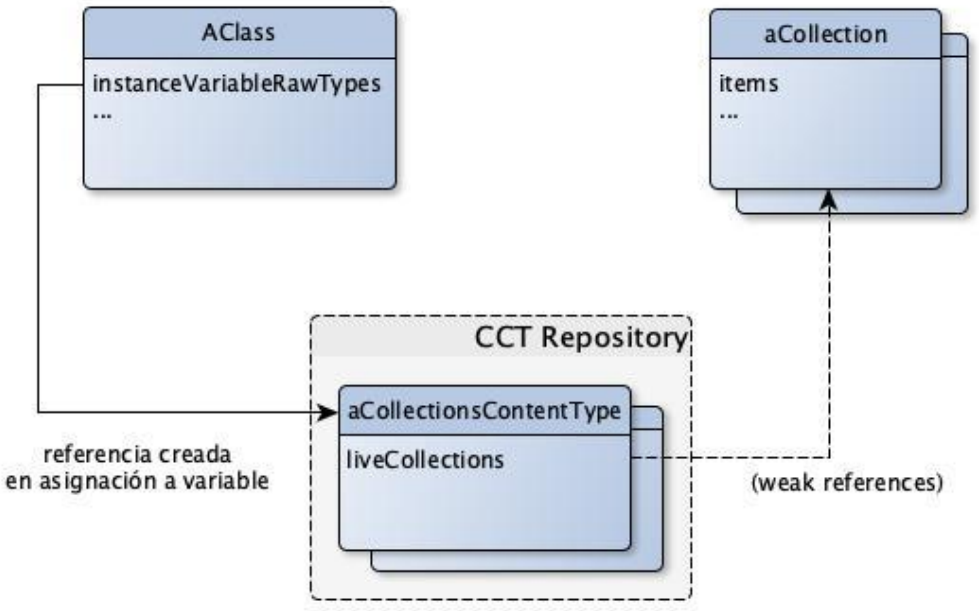
Debemos primero ahondar en la cardinalidad de la relación variable-colección:



Cardinalidad de relación variable-colección

Una variable puede ser asignada con distintas colecciones, y una colección puede asignarse a distintas variables que potencialmente pueden ser de colaboradores de distintas clases. El diseño de la solución, junto con el *CCT*, deben contemplar esta cardinalidad.

Evaluamos distintas formas de generar la asociación entre el *CCT*, la colección y la variable. Como primera opción, consideramos no realizar ninguna modificación a las colecciones existentes y mantener una implementación transparente como tiene *LiveTyping.* Decidimos pensar en un repositorio centralizado de *CCTs*, donde es necesario pensar en las referencias a las colecciones como colaboradores del *CCT*. El modelo que surge de ese enfoque es el siguiente:



LiveTyping para colecciones mediante un repositorio centralizado.

La idea de repositorio nos abstrae del problema de múltiples variables referenciando la misma colección, también conocido como *aliasing*, pero requiere que cada CCT conozca todas las colecciones a las que referencia en esas variables.

Para calcular los tipos en un *CCT*, sería necesario recorrer todos los elementos de las colecciones relacionadas. Este modelo presenta un problema que se manifiesta en el manejo de memoria, para saber los tipos dentro de la colección es necesario tener los elementos y estos pueden ser pesados de mantener vivos. Consideramos dos propuestas:

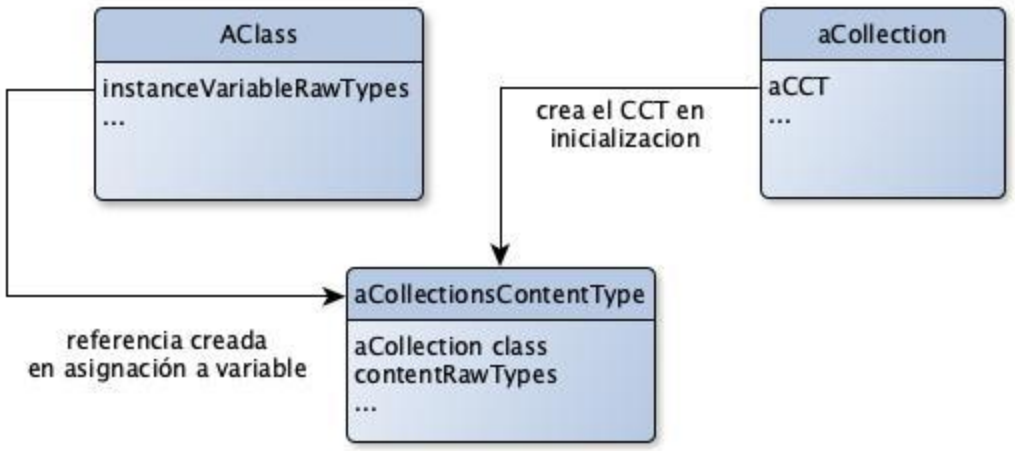
- Mantener sólo referencias débiles desde los *CCTs* a las colecciones para que el *garbage collector* pueda removerlas cuando no se usen más. En *tooling*, los algoritmos deberían contemplar el hecho de que algunas colecciones pueden haberse eliminado. Esta solución no parece ser la más útil ya que si el sistema es diseñado para que los objetos se mantengan poco tiempo referenciado, esa información de tipos no va a poder ser explotada.
- Mantener la referencia débil entre *CCT* y *Collection*, e inspeccionar periódicamente las colecciones, recolectar los tipos y copiarlos al *CCT*. Este proceso es asincrónico al agregado de elementos a las colecciones, por lo que puede existir un *delay* hasta que la información se vea reflejada y de todas maneras, existe el caso borde donde el *garbage collector* remueva la colección antes de que el *CCT* sea actualizado.

El modelo de repositorio para los *CCTs* se puede implementar desde la *VM* siempre y cuando:

- Se puedan crear instancias de *CCT* desde la *VM*.
- Ante la asignación de colección a una variable, se debe verificar si ya existe la asociación con la colección o debe crearse una nueva.

Existe otra forma de implementar el repositorio y es que cada entrada en él referencie tanto a la colección como al *rawTypes* (invirtiendo la fecha entre AClass y aCollectionsContentType). Esta estructura para la lectura es $O(n)$ donde $n = \#variables + \#colecciones$. Notar que la cantidad de objetos a revisar no se encuentra acotada por lo que puede ser un enfoque muy costoso.

Como segunda opción, analizamos que la instancia de colección conozca al *CCT* y en el caso de asignar una colección, guardar en el *rawTypes* un *CCT* asociado a esa instancia. El *CCT* continúa siendo actualizado a medida que los elementos se agreguen a la colección, y la relación con la variable permanecerá viva mediante el *rawTypes*. En caso de que el *garbage collector* borre la referencia a la colección, *LiveTyping* no se ve impactado ya que no la requiere para el cómputo, sólo el *CCT*.



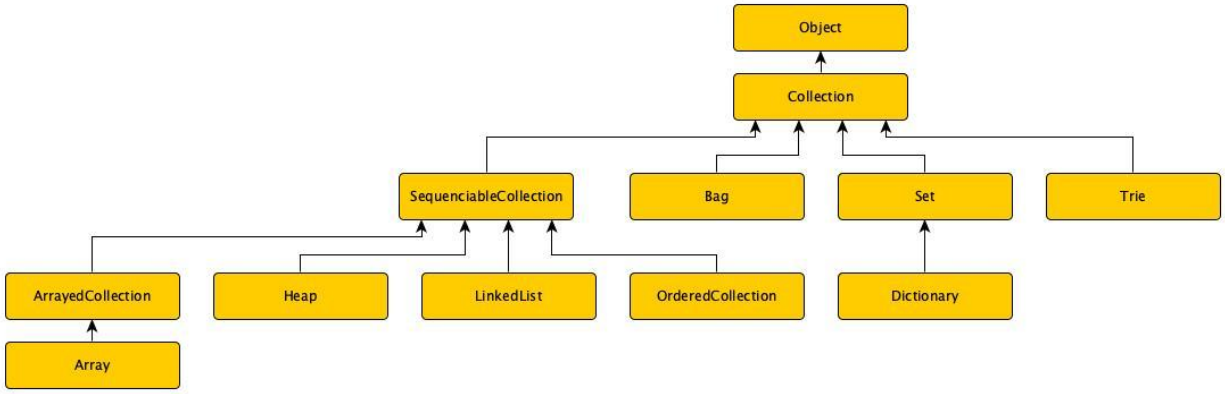
LiveTyping para colecciones decorando cada instancia con un CCT

Podemos ver cómo se relacionan los dos mundos *(la asignación de cada variable y la colección viva que muta sus elementos internos)*: la instancia de la clase *AClass* referencia al *CCT* de la colección, y esta lo mantiene actualizado conforme sus elementos cambian. Definitivamente, para esta solución es necesario poder agregar al *CCT* como colaborador de las colecciones.

Para el análisis de *tooling* que hicimos, utilizamos este modelo, restringido al subconjunto de tipos de colecciones ya mencionados.

4. Métodos que agregan elementos

Los mensajes que agregan elementos quedan definidos por la jerarquía a la que pertenece una colección. El siguiente gráfico, representa un subconjunto de clases de la jerarquía de *Collection* en Cuis:



Subset representativo de la jerarquía de Collection para Cuis 5.0.

LiveTyping plantea solucionar el problema de tipos para todos los objetos, por lo que para colecciones, la solución debería contemplar todas sus implementaciones.

El primer enfoque que abordamos fue el de decorar los métodos de agregación definidos en *Collection*:

```
add:
add:withOccurrences:
addAll:
```

Métodos para agregar elementos en la clase Collection.

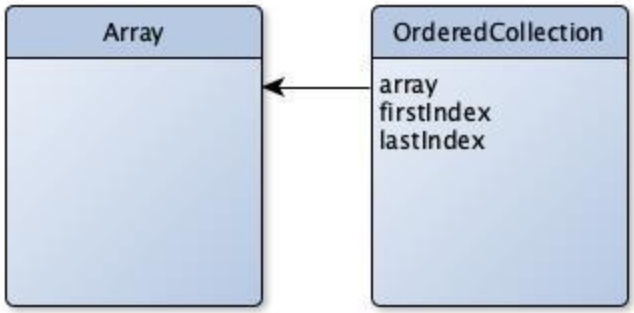
Si la VM puede introspectar estos métodos, se podría recolectar la información necesaria. Sin embargo, el problema surge cuando las implementaciones generan sus propios métodos que no terminan llamando a estos tres definidos por *Collection.* Además, para el caso particular de *Array*, usualmente se utiliza el *at:put:* para agregar elementos, el cual se encuentra definido en *Object* y no en *Collection*.

Investigamos entonces las distintas implementaciones de *Collection* de Cuis y resultó que varias de ellas están basadas en Arrays, como por ejemplo: *OrderedCollection* y *Set*. Estas implementaciones guardan los elementos en un *Array* interno, utilizando el método *at:put:* como

mecanismo. Para este subconjunto, queda bien claro que debemos interceptar únicamente el *at:put:* de *Array.*

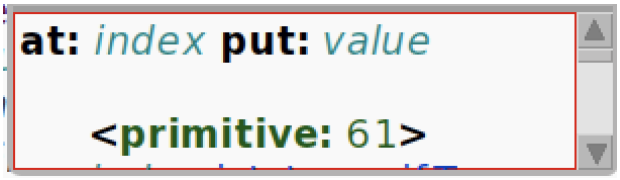
Decidimos en este trabajo, ahondar primero en las colecciones basadas en *Array,* para luego buscar la generalización del problema. Utilizamos a *OrderedCollection* como ejemplo.

Una *OrderedCollection* posee tres colaboradores internos, *array, firstIndex* y *lastIndex,* siendo el primero de ellos una instancia de la clase *Array.* Esta última sin embargo, no posee ningún colaborador interno ya que es una *variableSubclass* y parte de su implementación está escrita como primitivas.



Dependencia entre OrderedCollection y Array, junto con los colaboradores accesibles para cada instancia desde la imagen.

De hecho, podemos ver que el *at:put: de Array* se encuentra definido en *Object* y es la primitiva número 61.



Primitiva at:put: definida en Object.

Agregar un elemento a una colección basada en arreglos, entonces, desencadena la ejecución de un *at:put:*. El siguiente diagrama de secuencia detalla dicho proceso.

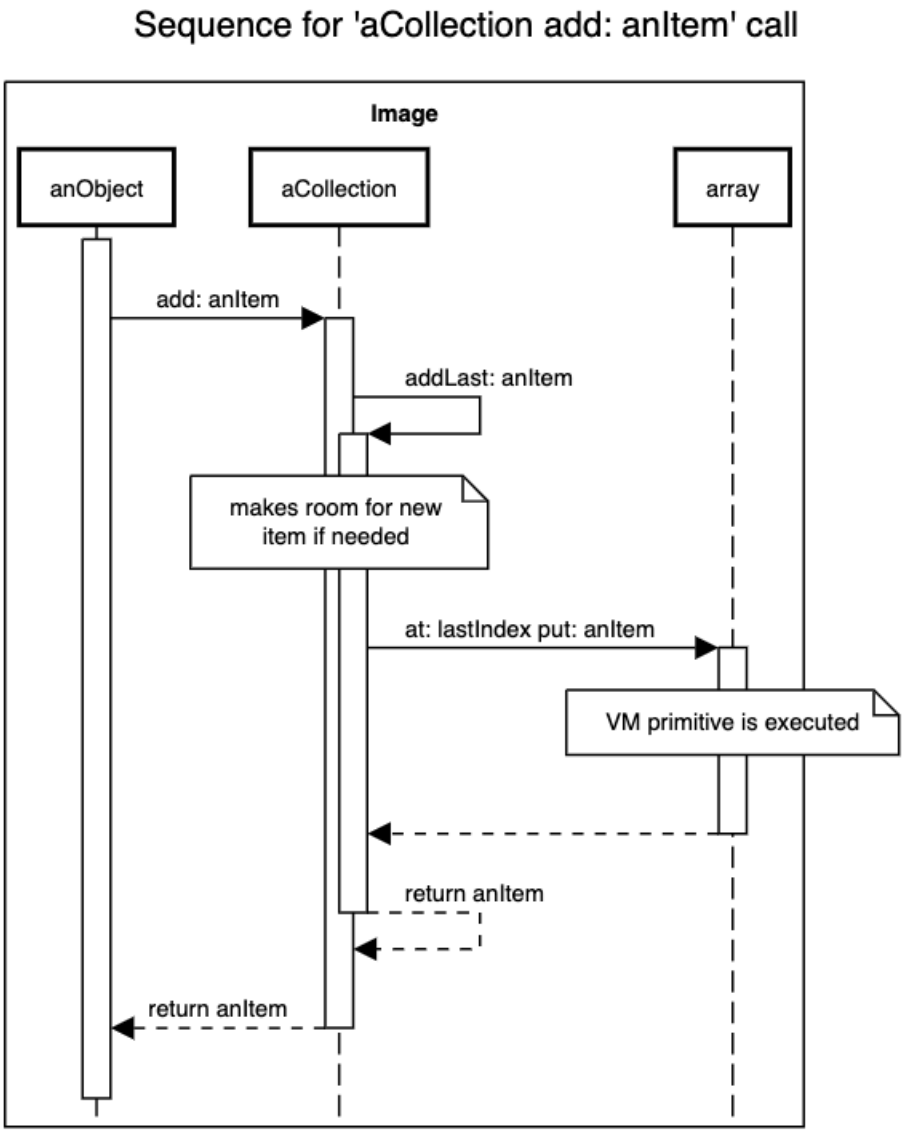


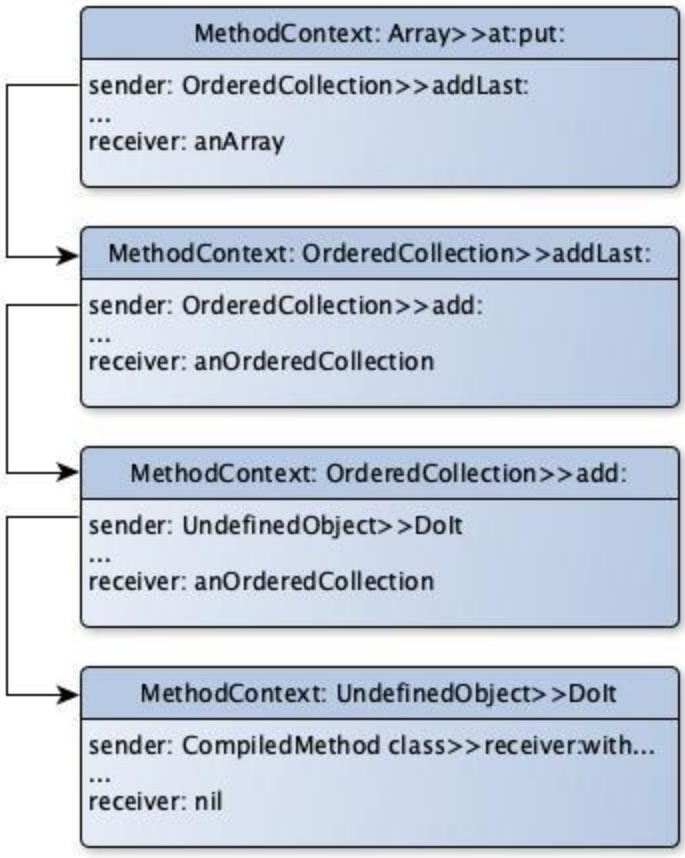
Diagrama de secuencia para el conjunto de colaboraciones desencadenado por 'aCollection add: anItem', siendo aCollection instancia de OrderedCollection.

Como *at:put:* es una primitiva, podemos modificarla en la VM para que además de guardar el objeto que se quiere agregar, guarde además la información de tipos en el CCT. Estando en la primitiva en la VM, surge la pregunta de cómo reconocer que ese *at:put:* proviene del agregado de un elemento en una colección y no cualquier otro tipo de colaboración en otros objetos que no son colecciones.

Como veremos en el apéndice sobre Smalltalk, la VM utiliza los contextos de ejecución para ejecutar el *bytecode*. Podemos utilizarlos entonces para buscar si el *sender* es efectivamente una *Collection*. Definimos que la implementación busca hasta 5 contextos más arriba si el emisor del mensaje es instancia de una de las colecciones marcada en el *specialObjectsArray* como que maneja *generics.* La decisión de revisar cinco contextos como

máximo es arbitraria y tiene como objetivo reconocer a la colección aún cuando el *at:put:* se ejecutó luego de varios *forwarding* de mensajes dentro del *Array*, o en objetos intermedios entre la colección y el arreglo: implementaciones de colecciones podrían utilizar el arreglo encapsulado y el algoritmo aún seguiría funcionando.

A continuación, se diagrama un subconjunto de los contextos que se anidan ante la ejecución del método *aCollection add: anItem.*



Contextos de ejecución anidados para 'aCollection add: anItem' instanciado desde un workspace.

Si instrumentamos el *at:put:* en la VM, podemos utilizar los contextos de ejecución para subir en el *stack de contextos* y encontrar si el *receiver* es una colección, y a su vez, encontrar la instancia que provee el *CCT*.

En este trabajo de tesis, si bien implementamos el algoritmo de búsqueda de contextos y la intervención del *at:put:* en la VM, decidimos cambiar el enfoque y mover parte de aquella implementación a la imagen ya que la naturaleza de la extensión de la VM es compleja: el simulador que existe para emular la VM en una imagen de Smalltalk y así probar los cambios es lento; en caso de no usar el simulador y recompilar la imagen, al haber un error de codificación generalmente obtenemos un error de *segmentation fault* y no es posible hacer TDD

con la infraestructura actual. Optamos por evaluar otras opciones de implementación que aceleren el desarrollo y permitan enfocarnos en otras tareas, como la de generación de herramientas para el desarrollador.

TypedArrayCollection

Una de las opciones evaluadas, fue la de crear un objeto *TypedArray* en la imagen encargado de intervenir el *at:put:*. El *TypedArray* es un *proxy* del arreglo común, que además recolecta información de tipos en el CCT.

Creamos entonces la *TypedArrayCollection*, que hereda de *OrderedCollection* y posee los mismos colaboradores que su padre. Fue creada para poder contener inicialmente el impacto de modificar a *Array* por *TypedArray:* existen múltiples instancias de *OrderedCollection* vivas en la imagen, y la solución debería contemplar que el sistema dinámicamente se actualice a usar *TypedArray*.

La intervención del *at:put:* en la imagen luce de la siguiente manera:

```
at: anIndex put: aValue

    |innerReturn valueClass specialObjectsArray liveCollections classIndex isLiveCollection |
    innerReturn := innerArray at: anIndex put: aValue.
    valueClass := aValue class.
    specialObjectsArray := Smalltalk specialObjectsArray.
    liveCollections := specialObjectsArray at: 66.
    classIndex := 0.
    isLiveCollection := liveCollections anySatisfy: [ :class |
        classIndex := classIndex + 1.
        class = valueClass
    ].
    isLiveCollection
        ifTrue: [
            | typedArray |
            typedArray := aValue instVarAt: ((specialObjectsArray at: 67) at: classIndex) + 1.
            types add: typedArray contentTypes.
        ]
        ifFalse: [
            types add: valueClass.
        ].
    ↑innerReturn.
```

Implementación de mensaje at:put: en TypedArray.

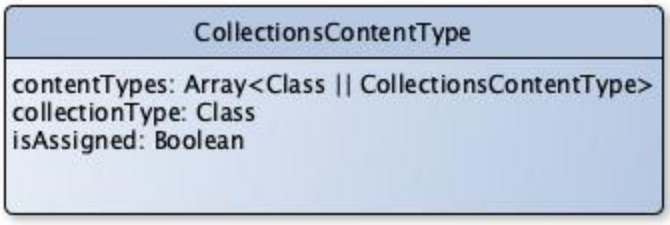
Podemos ver que la primera colaboración es enviar el mensaje *at:put:* al colaborador interno *(innerArray)* del *TypedArray*, el arreglo que se está enmascarando. Los siguientes pasos corresponden a recolectar el tipo del elemento que se desea agregar, debiendo prestar atención si lo que se agrega es una colección con *LiveTyping + generics (la llamaremos LiveTypingCollection)* o no.

- Para detectar que el elemento es de tipo *LiveTypingCollection* utilizamos el *specialObjectsArray*. En nuestra implementación, la posición 66 contiene un arreglo con todas las colecciones con *LiveTyping* soportadas.
- Si el elemento no es una *LiveTypingCollection*, se agrega la clase de la instancia al *CCT* del *TypedArray*.
- Si el elemento a agregar es una *LiveTypingCollection*, se busca entonces su *CCT* asociado. Vimos recién que en la implementación de *TypedArray*, este referencia al *CCT*, y no la colección: esta decisión es por diseño, dándole la responsabilidad al arreglo tipado de mantener actualizado los tipos.

Dado que en la solución final, el código antes mostrado debería ejecutarse en la VM, la cual maneja punteros a objetos, la opción que decidimos implementar, es agregar un nuevo arreglo al *specialObjectsArray* (en la posición 67) que posee el índice de la variable que representa el *TypedArray* dentro de la *LiveTypingCollection*. Esto nos permite independizarnos de la jerarquía y las variables que cada colección posea. Además, esta implementación se puede utilizar perfectamente desde la VM.

NOTA: *Una futura extensión a este algoritmo es modificar el arreglo de índices de la posición 67 para que sea un arreglo de arreglo de índices y poder detectar un TypedArray en un objeto anidado, en este caso, esa LiveTypingCollection tendría la posición de todos los objetos intermedios hasta llegar al arreglo. Cabe destacar que esto genera alto acoplamiento del specialObjectsArray con las LiveTypingCollections. Queda como trabajo futuro agregar tooling que mantenga automáticamente sincronizados estos arreglos con las colecciones que soporten LiveTyping.*

En este momento, podemos detallar la estructura del CCT, así como el uso de contextos de ejecución para reconocer la colección donde está aplicada.



El colaborador *'isAssigned'* lo veremos en la siguiente sección, pasando a detallar sobre la variable *'collectionType'*.

Las instancias de *CCT* se construyen a partir del mensaje de clase *new.* Recordemos que el *TypedArray* es el que crea la instancia del *CCT* y el tipo de la colección nunca se le pasa como colaborador (se envía el mensaje *'new'* sin parámetros), sino que se utilizan los contextos para detectar la colección de origen. Esta decisión es para mantener al CCT desacoplado de TypedArray, que es una solución de compromiso al no implementarlo en *Array*.

```
initialize
|currentContext iteration continue|
    currentContext := thisContext sender.
    iteration := 0.
    continue := true.
    [iteration < 10 and: currentContext notNil and: continue] whileTrue: [
        (currentContext receiver isKindOf: Collection) ifTrue: [
            collectionType := currentContext receiver class.
            continue := false.
            ].
        currentContext := currentContext sender.
        iteration := iteration + 1.
        ].
    contentTypes := Array new:10.
    isAssigned := false.
    ↑ self.
```

Método 'initialize' de instancia de CollectionsContentType.

Vemos que esta implementación busca en hasta 10 contextos padres por la primera colección que haya desencadenado en su creación y asigna en su variable *collectionType* al tipo de la colección. Esta información se utilizará luego para proponer mejoras en el *tooling* de Cuis.

Un detalle de color de esta implementación, es que en Smalltalk es trivial implementar un proxy (Gamma et al., 1994) de un objeto. En *TypedArray* hicimos justamente eso, el único mensaje que implementamos fue *at:put:*, para el resto utilizamos *doesNotUnderstand:* para reenviar el mensaje a la instancia intervenida.

```
doesNotUnderstand: aMessage
    "This message allows proxying every Array message we have yet to reimplement."

    |selectorInArray|

    selectorInArray := innerArray class lookupSelector: aMessage selector.

    selectorInArray ifNotNil: [
        ↑ aMessage sendTo: innerArray.
        ].

    ↑ super doesNotUnderstand: aMessage.
```

Implementación de Proxy pattern en TypedArray.

Respecto a la implementación a nivel VM, si consideramos entonces que los *CCT* se encuentran en cada instancia de *TypedArray*, el análisis de una colección por parte de *LiveTyping* debe poder relacionar la clase de la colección con la locación interna de su *TypedArray*. Para ello se utiliza el esquema de 2 arrays en el specialObjectsArray antes mencionado. Como podemos observar en el código a continuación, el segundo array se conoce como *TypedArrayIndexes*.

```
typedArrayIndexFor: collectionIndex

    | index typedArrayIndexes |
    typedArrayIndexes := objectMemory splObj: TypedArrayIndexes.
    index := objectMemory followField: collectionIndex ofObject: typedArrayIndexes.
    ^ objectMemory integerValueOf: index.
```

Implementación a nivel VM de la búsqueda del TypedArray para una colección.

De esta manera, una vez comprobado que el objeto siendo analizado por *LiveTyping* es una colección y el índice entre sus variables de instancia del array ahora tipado, obtener la referencia al *CCT* correspondiente es simplemente navegar al *TypedArray* y obtener su colaborador interno.

```
contentTypesOf: aCollection with: typedArrayIndex

    | typedArray contentTypes |
    typedArray := objectMemory followObjField: typedArrayIndex ofObject: aCollection.
    contentTypes := objectMemory nilObject.
    typedArray = objectMemory nilObject
        ifFalse: [
            contentTypes := objectMemory followObjField: 1 ofObject: typedArray "Obtain
'types' variable within the TypedArray instance"
        ].
    ^ contentTypes.
```

Implementación a nivel VM de la búsqueda del CCT correspondiente a una colección.

5. Mantener actualizado el CCT

Vimos en la sección anterior que el CCT se mantiene actualizado en la intervención del método *at:put:*, sin embargo, no hemos tratado el problema de las múltiples colecciones.

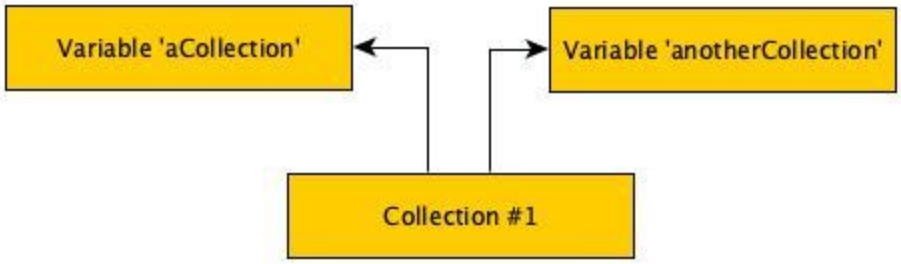
Por otro lado, en el punto 3 destacamos que la relación entre CTTs y variables es 'n' a 'm', es decir, una colección puede haber sido asignada a múltiples variables, y cada variable puede ser asignada por múltiples colecciones. Los *raw types* además tienen un tamaño finito.

Veamos como quedarían las relaciones entre variables y colecciones en el siguiente método de instancia creado en la clase *'Example'*, la cual posee dos variables de instancia, *'aCollection'* y *'anotherCollection'*:

```
m1: aCondition
aCondition
    ifTrue: [
        aCollection := OrderedCollection new. "Collection #1"
        aCollection add: 2.
        anotherCollection := aCollection
    ] ifFalse: [
        aCollection :=  OrderedCollection with: 1.0. "Collection #2"
        anotherCollection :=  Set with: 'aString'. "Collection #3"
    ].
↑ aCollection
```

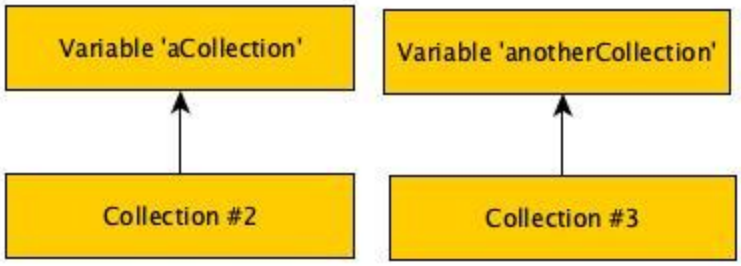
Método de ejemplo donde se asigna dos variables de instancia de tipo colección.

Si creamos la clase con este método y llamamos a *'m1'* por única vez como *'m1: true'*, *LiveTyping* deberá generar la siguiente relación entre las colecciones y las variables:



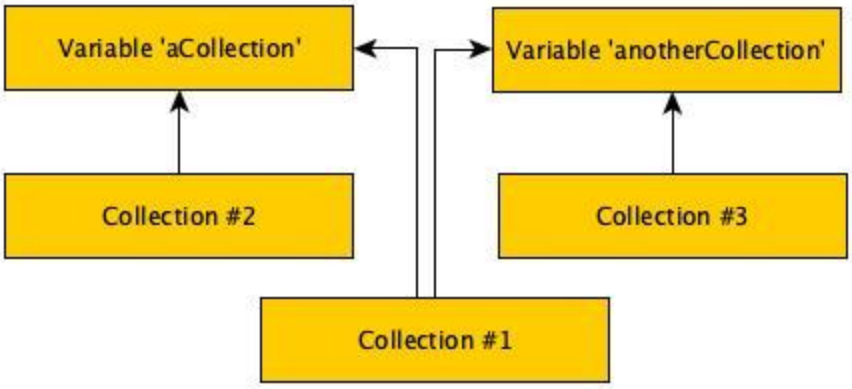
Relación entre colecciones y variables luego de ejecutar 'Example new m1: true' únicamente.

Si en cambio luego de crear la clase llamaramos únicamente a *'m1: false'* el diagrama sería distinto:



Relación entre colecciones y variables luego de ejecutar 'Example new m1: false' únicamente.

Si por último, ejecutamos tanto *'m1: true'* como *'m1: false'*, el diagrama queda distinto:



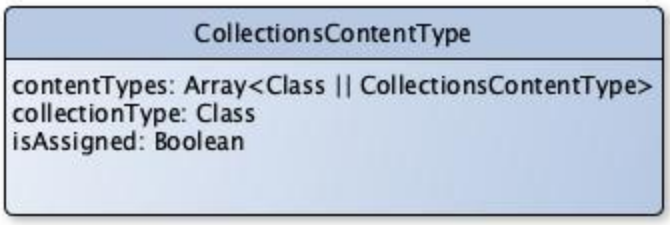
Relación entre colecciones y variables luego de ejecutar 'Example new m1: true. Example new m1:false'.

A priori, en el *instanceVariableRawTypes* veríamos dos CCTs para cada variable. Ahora bien, las colecciones #1 y #2 son ambas de tipo *OrderedCollection* y en este ejemplo en

particular, tendría sentido considerar que representan el mismo conjunto de *generics*. Llamamos a este problema: *aliasing de generics.*

Aliasing

Recordemos primero de qué está compuesto el CCT en nuestro modelo resultante.



Colaboradores internos del CollectionsContentType

- **contentTypes:** Referencia a los tipos de los elementos que se agregan u otro *CCT* si son generics anidados
- **collectionType:** Define el tipo de la colección que se está relevando sus generics.
- **isAssigned:** Retorna si la colección se encuentra asignada a una variable.

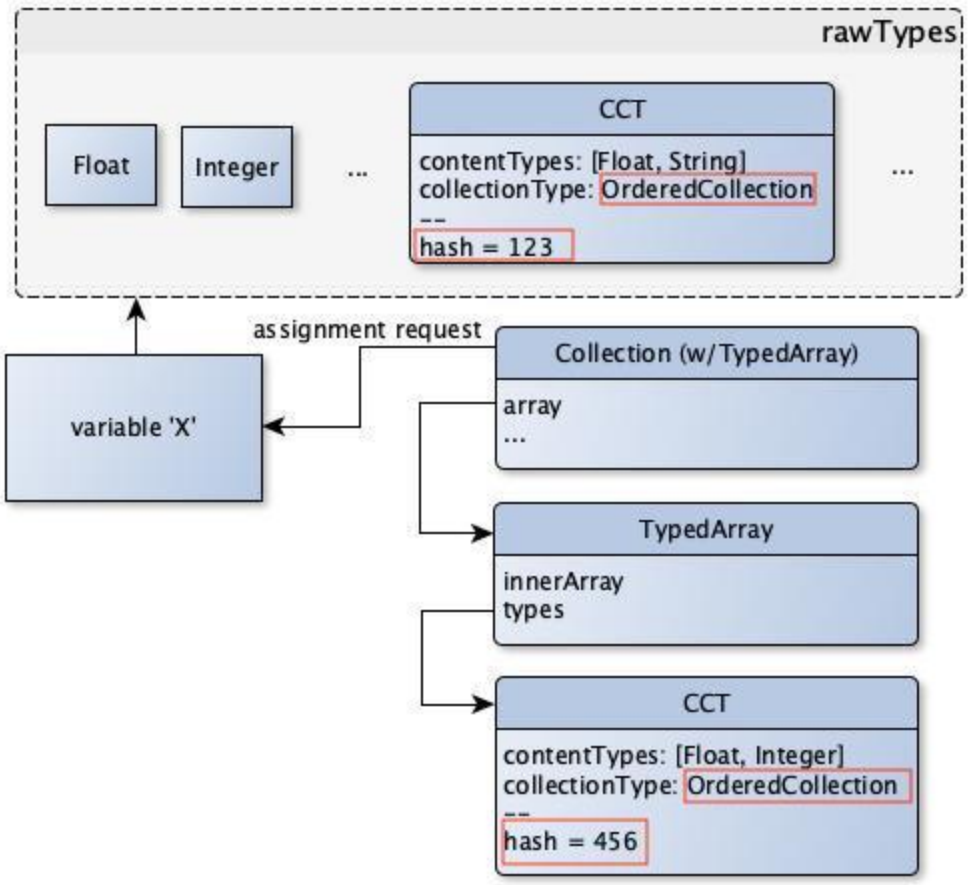
Del modelo podemos ver que para una variable, se definió un *CCT* por tipo de *Collection*. Esto implica que en el ejemplo anterior, un *CCT* de *Set* no unifica con uno de *OrderedCollection*, manteniendo separados en el *rawTypes* de la variable. Esta decisión es consistente con la implementación de *LiveTyping* dónde clases polimórficas (ej: Float e Integer) se mantienen como elementos distintos en el *rawTypes* y se unifican al utilizarse en la imagen. En la sección de *tooling* vimos cómo se realiza el proceso de unificación.

El primer algoritmo de *aliasing* fue implementado en la VM, no necesitando de la variable *isAssigned* para funcionar, siendo su lógica la siguiente:

- La colección debe construir un *CCT* desde la imagen
- Cuando el intérprete debe asignar una colección a una variable,
    - Si el *rawTypes* ya posee un *CCT* con el mismo tipo de colección, se sobreescribe el *CCT* de la colección y se actualiza *contentTypes* con la unión de tipos que se encuentran en ambos CCTs.
    - Si el *rawTypes* no lo contiene, se agrega la referencia al *CCT* de la colección.

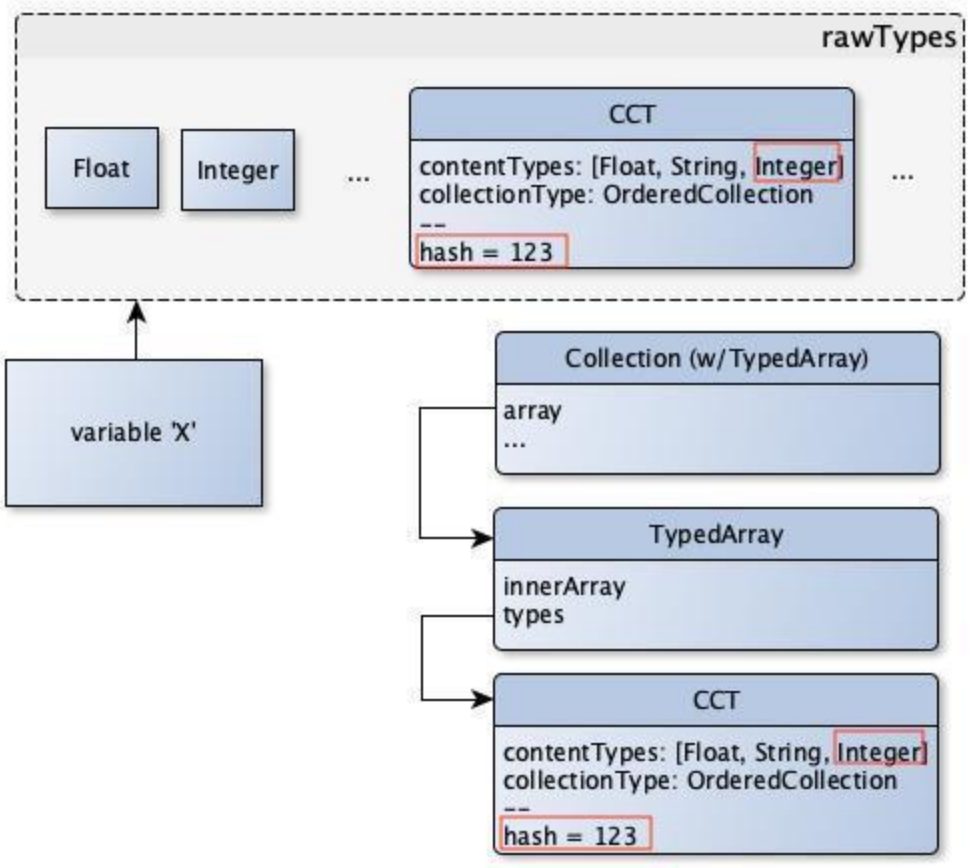
Sean los siguientes gráficos, ejemplos del algoritmo de aliasing previamente descripto:

Ejemplo 1: Unificación de *contentTypes* con mismo *collectionType.*



Posible estado del instanceVariableRawTypes previo a la asignación de una nueva colección.

En este ejemplo tenemos la variable *'X'* que fue asignada previamente con una *OrderedCollection* la cual poseía un *CCT* con hash '123'. Como se le está asignando una *OrderedCollection* pero que posee un *CCT* distinto (hash '456'), la VM procede a efectuar la unificación. A continuación, vemos el estado final:

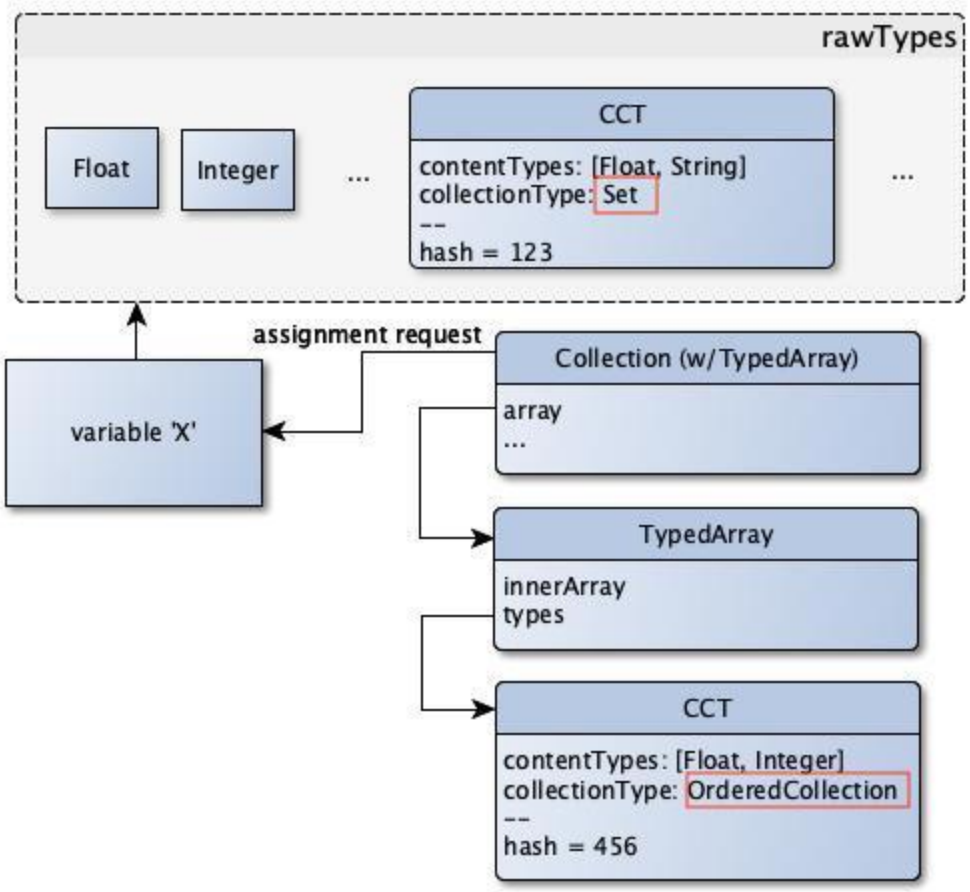


Estado del instanceVariableRawTypes luego de la asignación de una nueva colección que genera aliasing.

La VM actualiza el *CCT* de la variable con todos los *contentTypes* desconocidos hasta el momento que la nueva colección aporta (en este caso, el *Integer*), y luego el *TypedArray* con la referencia al *CCT* de la variable (notar que ambas poseen el mismo hash ahora pues son la misma instancia).

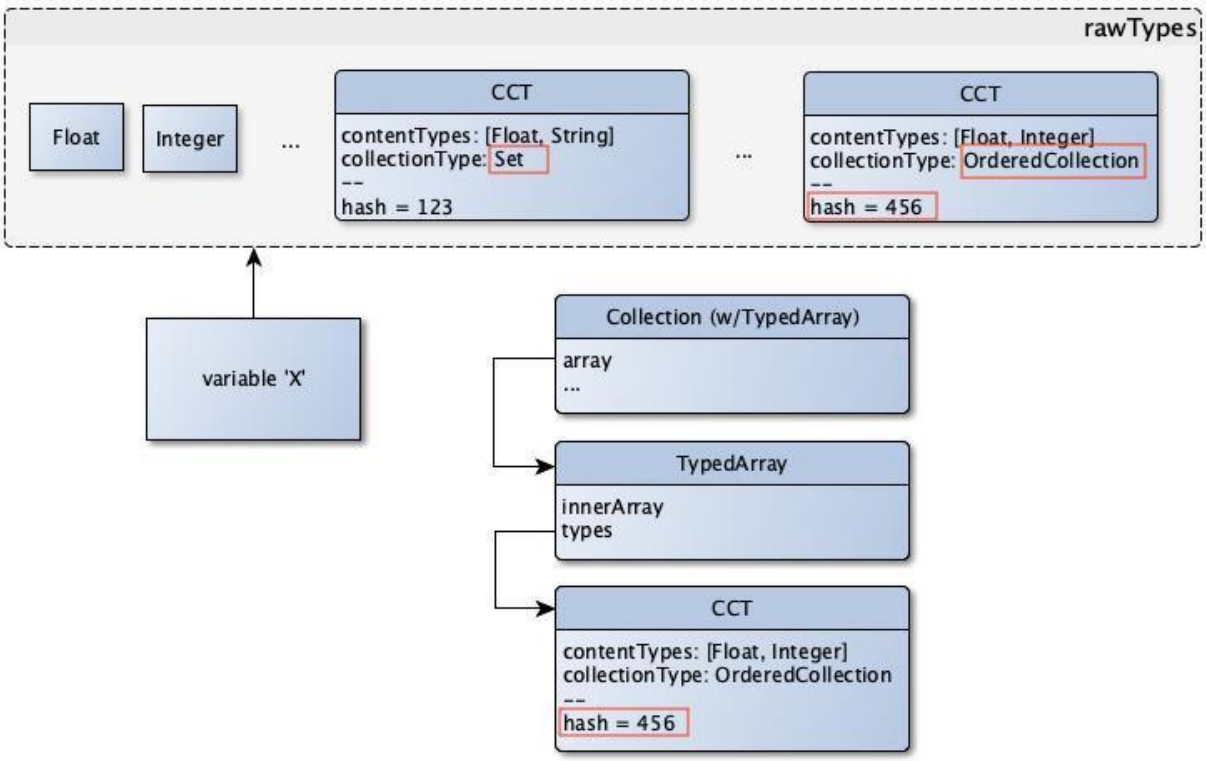
Ejemplo 2: No unificación al tener distintos *collectionTypes*

En el caso de que la variable ‘X’ no haya sido asignada previamente con una colección o posee un *CCT* pero de otro tipo de colección, nos encontraremos en el siguiente estado:



Posible estado del instanceVariableRawTypes con un CCT de Set previo a la asignación de una nueva colección.

Si el *rawTypes* no posee referencia a otro *CCT* o poseen *collectionTypes* distintos, entonces la VM no unifica y agrega el nuevo *CCT* al *rawTypes.* El estado resultante es el siguiente:



*Estado del instanceVariableRawTypes luego de la asignación de una nueva colección que **no** genera aliasing.*

Vemos que el *rawTypes* ahora posee dos CCT, uno para cada *collectionType*, siendo el nuevo de la colección que acaba de asignarse (mismo hash).

Al validar empíricamente este algoritmo, el resultado es que se forman grupos de variables todas relacionadas entre sí. Más aún, *LiveTyping* funciona no sólo para variables de instancias, sino para los colaboradores y los *return types* de la ejecución de métodos. Cuando el algoritmo de *aliasing* se enciende, fusiona en los tres puntos donde se obtienen tipos manteniendo el mismo CCT para decenas de variables y puntos de inspección de *LiveTyping.* Con este enfoque, la información de tipos se generaliza muy rápidamente y resulta ineficaz. Este resultado es fácil de ver en los tests cuando se asigna una colección a dos variables distintas y a partir de ese punto en adelante, ambas variables reportan los mismos tipos genéricos.

Para poder romper con la propagación de *aliasing*, manteniendo la capacidad de sumarización, se debe acotar el *aliasing* sólo dentro de una variable. Es aquí donde surge la necesidad del colaborador *'isAssigned'*: el CCT sólo puede unificarse con la primera variable a la que se asigna, sin poder propagar *aliasing* a las otras potenciales variables en las que puede estar referenciado.

```
setAssigned: contentTypes
    <inline: true>
    |trueObject|
    trueObject := objectMemory trueObject.
    objectMemory storePointer: 2 ofObject: contentTypes withValue: trueObject.
```

Implementación a nivel VM del assignment de un CCT.

A continuación, se detallan las extensiones de la VM para aplicar el algoritmo de *aliasing.* El primer gráfico, describe cómo se reemplaza el *CCT* de la colección a asignar por el *CCT* que ya se encuentra en el *rawTypes:*

```
replaceContentType: cct in: aCollection givenTypedArrayIndex: typedArrayIndex

    | typedArray |
    typedArray := objectMemory followObjField: typedArrayIndex ofObject: aCollection.
    objectMemory storePointer: 1 ofObject: typedArray withValue: cct.
```

Implementación a nivel VM del reemplazo de CCTs.

El siguiente gráfico, refleja el proceso de unificación, que es necesario cuando el *CCT* a reemplazar tiene tipos internamente:

```
merge: contentTypes with: anotherContentTypes

    |typesArray anotherTypesArray ithType size index|

    "Index of 'contentTypes' variable within CCT is 0."
    typesArray := objectMemory followObjField: 0 ofObject: contentTypes.
    "Index of 'anotherContentTypes' variable within CCT is 0."
    anotherTypesArray := objectMemory followObjField: 0 ofObject: anotherContentTypes.
    size := (objectMemory lengthOf: anotherTypesArray)-0.
    index := 0.

    [index < size]
        whileTrue: [
            ithType := objectMemory followObjField: index ofObject: anotherTypesArray.

            (ithType = objectMemory nilObject)
                ifFalse: [
                    self storeType: ithType in: typesArray.
            ].

            index := index + 1.
    ]
```

Implementación a nivel VM de la unificación de CCTs.

Vale recordar que tanto la unificación como el reemplazo se realizan contra una instancia de *CCT* ya capturada y equivalente, es decir, que ya es parte de un *rawTypesArray* de *LiveTyping* y posee el mismo tipo base (*OrderedCollection*, por ejemplo). El siguiente método muestra cómo la VM realiza ese chequeo:

```
findCollectionContentType: collectionType in: anArrayOfTypes

    |size cct cctClass index objectAtIndex notFound objectAtIndexClassTag objectAtIndexClass collectionClass |

    size := (objectMemory lengthOf: anArrayOfTypes)-0.
    index := 0.
    notFound := true.
    cct:= objectMemory nilObject.
    cctClass := objectMemory splObj: ClassCollectionsContentType.

    [notFound and: [index < size]]
        whileTrue: [
            objectAtIndex := objectMemory followObjField: index ofObject: anArrayOfTypes.
            objectAtIndex = objectMemory nilObject
                ifFalse: [
                    objectAtIndexClassTag := objectMemory fetchClassTagOf: objectAtIndex.
                    self deny: (objectMemory isForwardedClassTag: objectAtIndexClassTag).
                    objectAtIndexClass := objectMemory classForClassTag: objectAtIndexClassTag.
                    self deny: objectAtIndexClass isNil.

                    objectAtIndexClass = cctClass
                        ifTrue: [
                            collectionClass := objectMemory followObjField: 1 ofObject: objectAtIndex.
                            collectionClass = collectionType
                                ifTrue: [
                                    notFound := false.
                                    cct := objectAtIndex.
                                ]
                        ].
                ].
                index := index + 1.
        ].

    ^ cct.
```

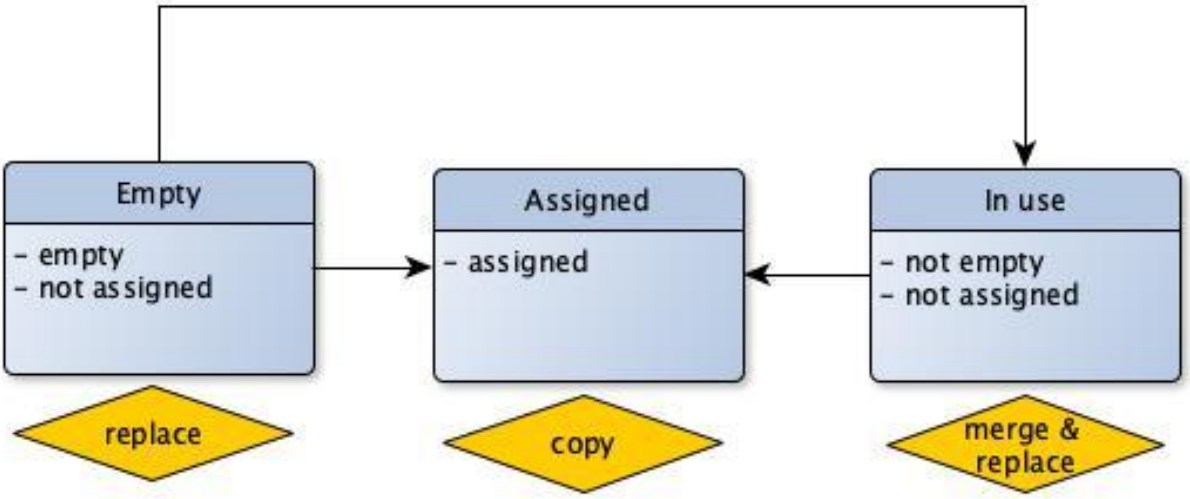
Implementación a nivel VM de la búsqueda de CCT equivalente dentro de un rawTypesArray.

Estados

Dado el código definido anteriormente, podemos pensar entonces en que la colección pasa por tres estados:

- Vacía: aún no se encuentra asignada, y no posee ningún elemento.
- En uso: continúa sin asignarse, pero ya posee elementos.
- Asignada: la colección se encuentra asignada a una variable.

El siguiente diagrama muestra estos estados y sus relaciones.

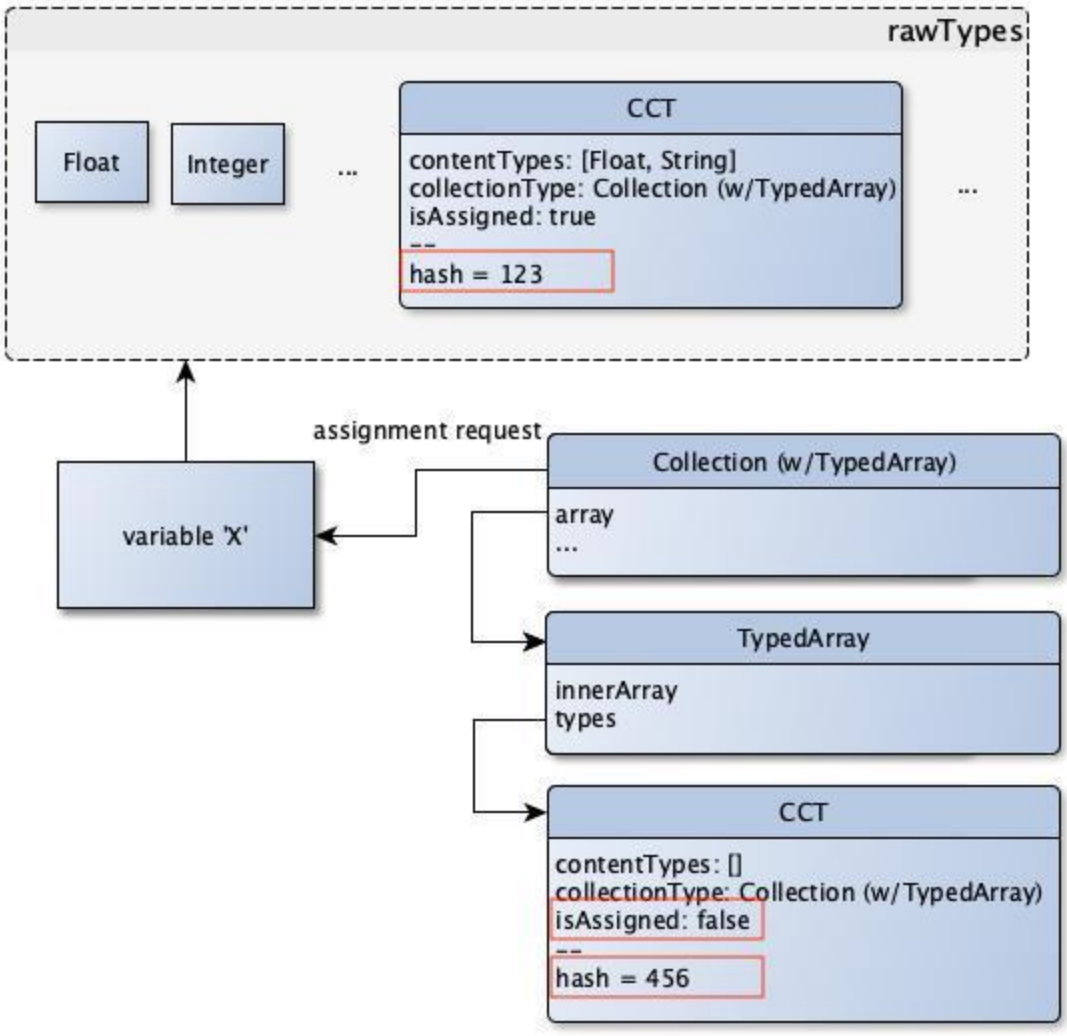


Diferentes estados por los que una colección puede pasar en el proceso de asignación a variable

A continuación analizaremos en detalle los estados posibles y las transiciones de un *CCT*, poniendo foco en las consecuencias del mismo según el caso.

Empty → Assigned

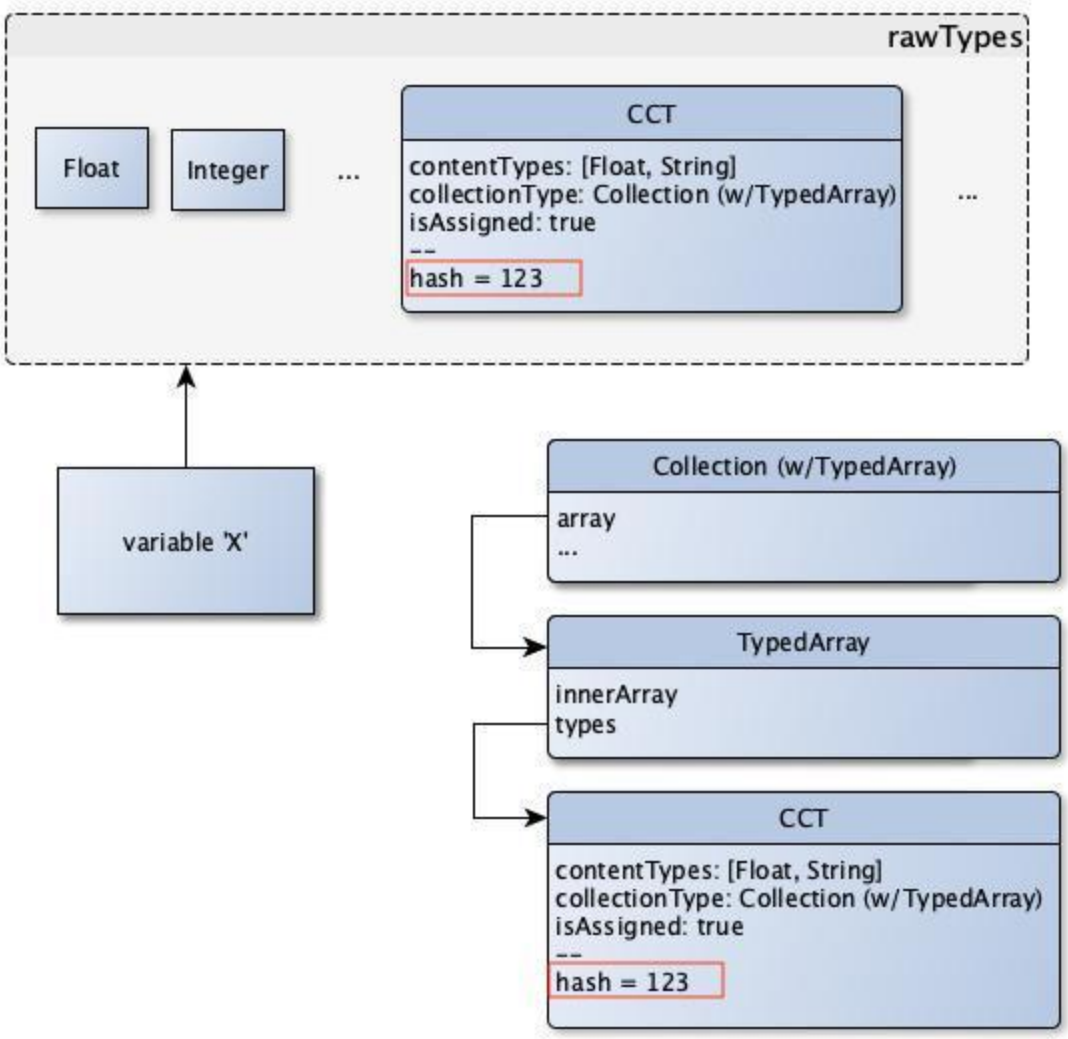
Vale notar que pasar de In Use → Assigned es un escenario análogo.



Asignación de colección no asignada a variable.

En este ejemplo, se desea asignar una *collection* que nunca ha sido asignada anteriormente. El *rawtypes* de la variable ya posee un CCT para el mismo tipo *(OrderedCollection)* por lo que el algoritmo procede a reemplazar el CCT de la colección por el del *rawTypes*. En este caso particular, como el nuevo CCT no posee ningún elemento, el *contentTypes* del *rawTypes* se mantiene con *Float* y *String.* Si la colección estuviera *In Use*, se realizaría el *merge* entre ambos CCTs antes de reemplazar el de la colección.

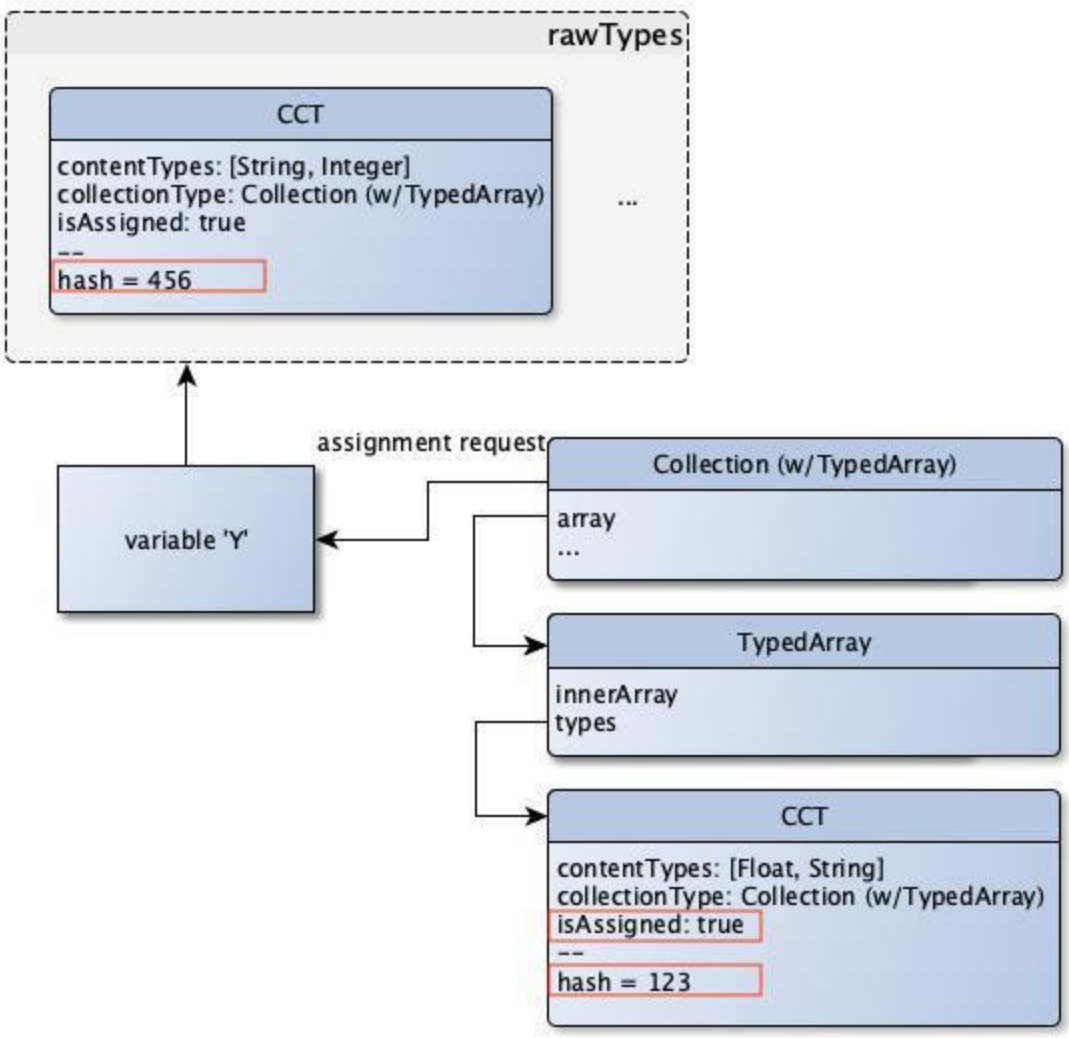
El resultado final es que la colección asignada referencia al CCT del *rawTypes:*



Actualización del CCT luego de asignación de colección a variable.

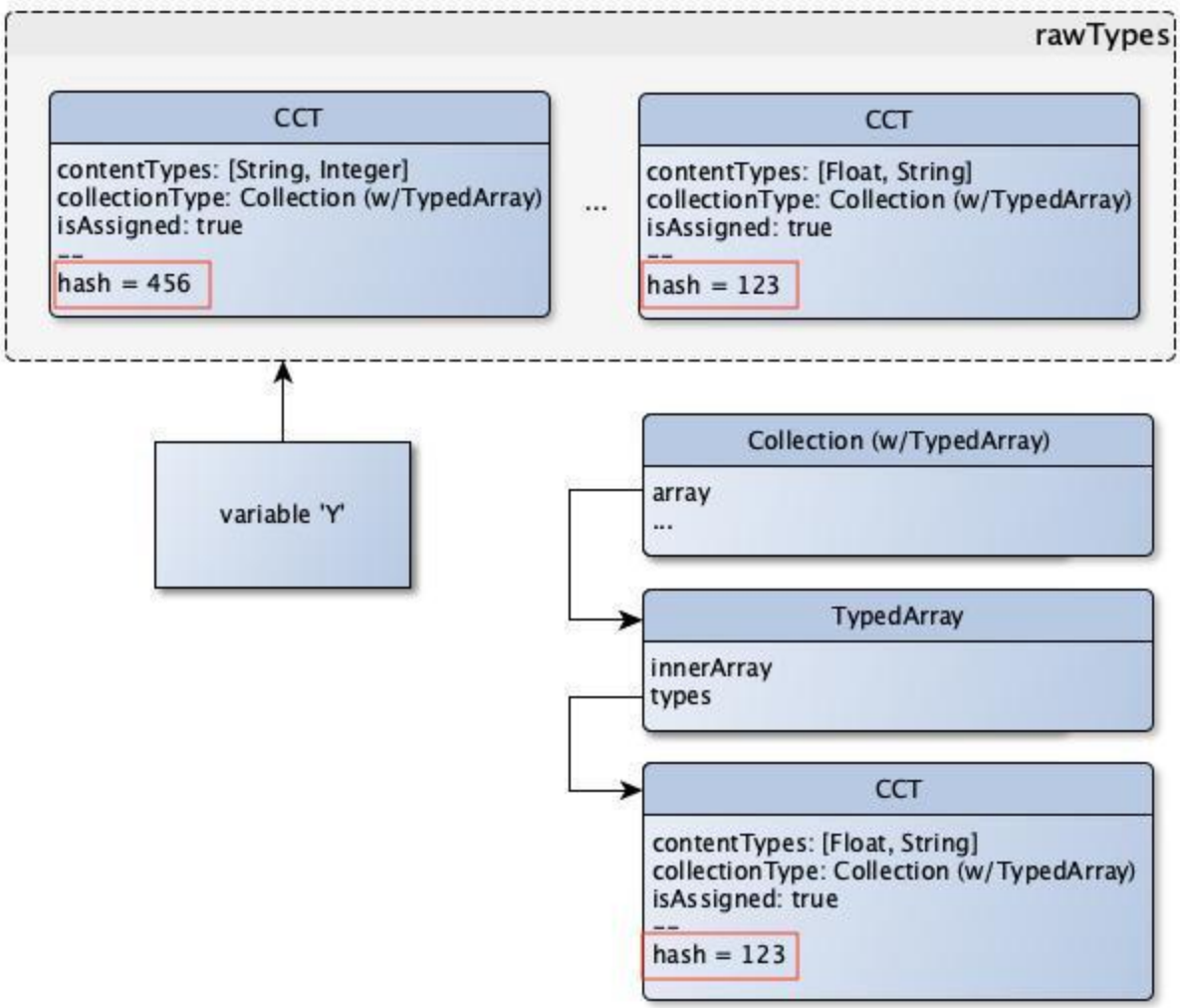
Si la colección del ejemplo anterior luego se asigna a otra variable *(supongamos variable 'Y')*, su *rawTypes* agrega la referencia al CCT sin unificar con otros CCTs que pueda contener.

Assigned → Assigned



Asignación de una colección ya asignada a otra variable.

Luego de efectuar la asignación, el *rawTypes* hace referencia al nuevo CCT, sin realizar el *merge* entre los otros CCTs que posee.



Estado resultante del *rawTypes* luego de la asignación de una colección ya asignada a otra variable.

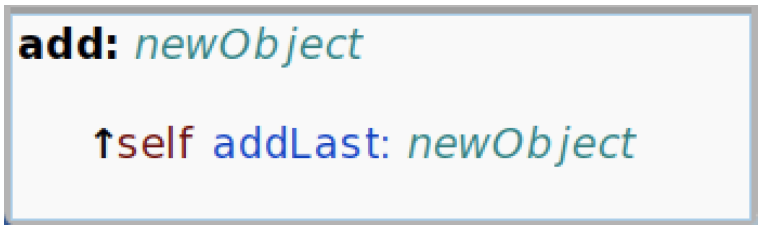
Apéndice II: Smalltalk 80

Se distinguen dos componentes en un sistema basado en la especificación de Smalltalk-80 (Goldberg & Robson, 1983), la imagen virtual y la máquina virtual:

1. La imagen virtual *(virtual image o image)* consiste en todos los objetos del sistema.
2. La máquina virtual *(virtual machine ó VM)* consiste en los dispositivos de *hardware* y en las rutinas en lenguaje máquina que genera objetos dinámicos en la imagen virtual.

En las implementaciones de Smalltalk 80, además de objetos y mensajes, se resaltan al menos cuatro conceptos: **métodos**, **compilador**, **intérprete** y **memoria de objetos**.

El código de los métodos que escribe un programador se representan como instancias de String, los cuales deben conformar con la sintaxis del lenguaje:



Ejemplo de código fuente (OrderedCollection>>#add:)

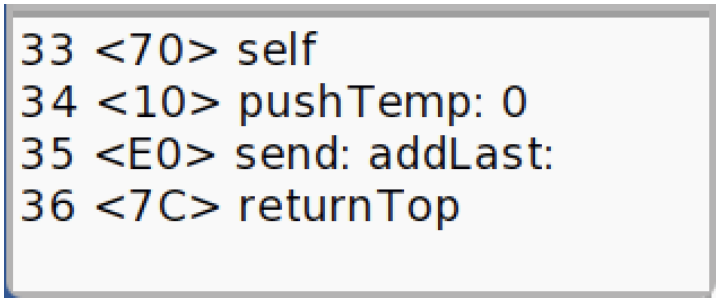
El **compilador** traduce el String representando al método en una secuencia de instrucciones para el **intérprete de stack.** Estas instrucciones las llamamos ***bytecodes.*** Los *bytecodes* junto con metadata componen el ***CompiledMethod.*** En un ambiente de Smalltalk, cada clase deberá generar sus métodos para que puedan ser ejecutados por el intérprete, para ello, deberán pedirle al compilador que genere el *CompiledMethod* a partir del código fuente.

Un *CompiledMethod* contiene no sólo los *bytecodes* que representan el código fuente sino también un conjunto de objetos referenciados desde el *literal frame.* El *literal frame* contiene a todos los objetos que no pueden ser automáticamente mapeados a *bytecodes*, como por ejemplo variables compartidas (globales, de clase), literales constantes y la mayoría de los selectores de los mensajes que se envían en el método (existen algunos selectores con un *bytecode* asociado por lo que no aparecen en el *literal frame*).

La cantidad de *bytecodes* disponibles dependen de la implementación, pero suelen caracterizarse en 5 tipos:

- *Push bytecodes:* indica que un objeto debe agregarse al *stack.*

- *Store bytecodes:* indica qué variable debe actualizarse con el resultado de una o más operaciones.
- *Send bytecodes:* especifica el selector de un mensaje a ser enviado y cuántos argumentos debe tener.
- *Return bytecodes:* indican que el *CompiledMethod* terminó de ejecutar y qué variable deberá retornar.
- *Jump bytecodes:* indican que el próximo *bytecode* a ejecutar no es el siguiente al *jump,* pueden ser condicionales y se utilizan para implementar eficientemente estructuras de control.



Ejemplo de bytecodes para el método presentado anteriormente (OrderedCollection>>#add:)

Intérprete de *stack*

El intérprete ejecuta los *bytecodes* que se encuentran en los *CompiledMethods*. El proceso es un ciclo de tres pasos que requiere la siguiente información:

- El *CompiledMethod* que contiene los *bytecodes* a ejecutar.
- La dirección en el *CompiledMethod* del próximo *bytecode* a ejecutar (también llamado *instruction pointer*).
- El receptor y los argumentos del mensaje que invocó el *CompiledMethod.*
- Las variables temporales necesarias por el *CompiledMethod*.
- Un *stack.*

Con esta información, el ciclo del intérprete queda definido por:

- Traer el *bytecode* del *CompiledMethod* indicado por el *instruction pointer.*
- Incrementar el *instruction pointer.*
- Ejecutar la función especificada por el *bytecode*
    - La mayoría de las funciones involucran el *stack.* Los *push bytecodes* dicen dónde encontrar objetos para agregar al *stack*. *Store bytecodes* define dónde poner los objetos en el *stack*. *Send bytecodes* remueve el receptor y los argumentos del mensaje del *stack*. Los resultados de la ejecución del mensaje son guardados en el *stack.*

Contextos de ejecución

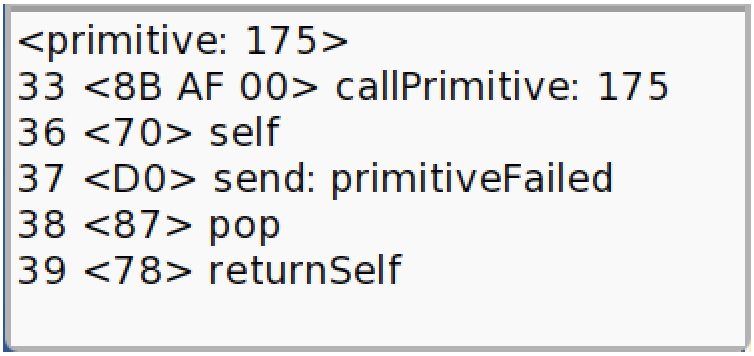
Los *bytecodes* de *push, store* y *jump* requiere únicamente cambios simples en el estado del intérprete: se altera el stack ya sea al agregar o quitar objetos y/o se modifica el *instruction pointer*, manteniendo el resto del estado. En el caso del *send* y *return bytecodes* puede necesitar cambiarse el estado completo del intérprete. Cuando un mensaje se envía, las 5 componentes del estado del intérprete pueden necesitar cambiar para que pueda ejecutarse el *CompiledMethod* correspondiente al nuevo mensaje. El estado anterior del *interpreter* debe recordarse para resumir la ejecución cuando el nuevo mensaje termine.

El intérprete guarda su estado en objetos llamados *context*s. En la ejecución del sistema, van a existir múltiples contextos al mismo tiempo, siendo el *active context* el que representa al estado actual del *interpreter.*

Cuando un *bytecode* de *send* en el *active context* de un *CompiledMethod* requiere la ejecución de un nuevo *CompiledMethod*, el contexto activo se suspende y uno nuevo se crea y marca como activo. Cada contexto que se activa debe recordar qué contexto suspendió para poder resumir la ejecución una vez que el activo termine. El contexto suspendido se llama el *sender* del nuevo contexto.

Métodos primitivos

Cada *CompiledMethod* además de exponer al intérprete su *bytecode,* pueden indicar que existe una primitiva que responde el mensaje, permitiendo al *interpreter* responder el mensaje sin ejecutar *bytecode* de la imagen. Estas primitivas se encuentran en la VM y no requieren la creación de un nuevo contexto. La ejecución de una primitiva puede fallar, por ejemplo, porque los tipos de los parámetros no corresponden con los requeridos por la primitiva. Cuando el fallo sucede, el *intérprete* ejecuta el *CompiledMethod* como si la primitiva no hubiera existido.



Ejemplo de bytecode con primitiva (*Behavior*, *identityHash*)

Object memory

El *object memory* es un objeto que provee al *interpreter* una interfaz a los objetos de la imagen. Cada objeto es asociado con un identificador unívoco llamado *object pointer*. La cantidad de *object pointers* accesibles queda sólamente restringida por la implementación de Smalltalk que se use y no el lenguaje.

Los *object pointers* pueden estar asociados entre sí:

- Cada *object pointer* de un objeto se encuentra asociado con el *object pointer* de la clase del objeto.
- Si el objeto tiene variables de instancia, su *object pointer* se encuentra asociado a los *object pointer* de los valores de las variables.

El *object memory* provee entonces las siguientes capacidades al *interpreter:*

- Acceso al valor de una variable de instancia de un objeto.
- Modificar el valor de una variable de instancia de un objeto.
- Acceder a la clase de un objeto.
- Crear un nuevo objeto.
- Sabe cuántas variables de instancia un objeto posee.

Arrays

Un detalle importante a considerar sobre Smalltalk y en particular su implementación en OpenSmalltalk VM, que es la utilizamos en este trabajo de investigación, es que si bien la clase Array pertenece a la jerarquía de *Collection*, es que su estructura se accede por índices en vez de variables nombradas. Además, gran parte de la implementación se encuentra en forma de primitivas, es decir, escritas en la VM.

Apéndice III: LiveTyping

Raw Types

Llamamos *raw types* a los tipos capturados por la VM durante la ejecución de distintas operaciones. Estos tipos son las clases de los objetos observados y se insertan en arrays limitados a 10 *slots* (por defecto). Cuando un tipo es observado, se compara contra los tipos presentes en los *slots* ocupados y sólo de ser nuevo se coloca en el primer *slot* libre. Si el arreglo se encuentra lleno, se procede a descartar el nuevo tipo.

Estos arrays de tipos recolectan la información de *LiveTyping* para algún componente deseado, como pueden ser variables de instancia o temporales, parámetros o tipos de retorno.

En caso de no querer recolectar información para algún componente, basta con definir en *nil* la variable en la que se encuentra el *raw types* asociado al componente.

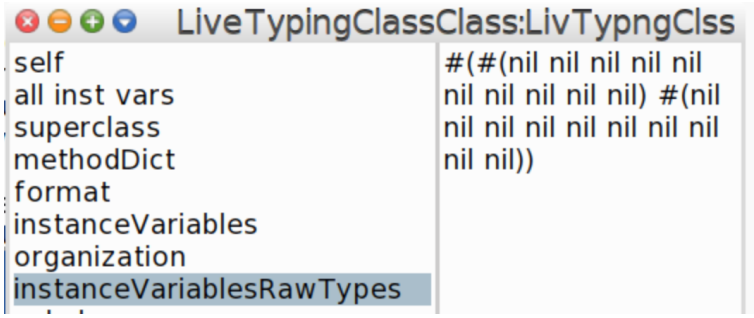
Variables de Instancia

Las variables de instancia son tipadas con la introducción de la variable de instancia *instanceVariablesRawTypes* en *ClassDescription*. *ClassDescription* forma parte del metamodelo que define el comportamiento y estructura de todas las clases. Se utiliza esa variable para recolectar la información de tipos de sus variables de instancia. *instanceVariablesRawTypes* es instancia de *Array* y cada posición contiene el *raw type* de la i-ésima variable de la clase: las *instance variables* se numeran por la posición en la que aparecen en la definición de la clase.

Supongamos que tenemos la clase *LiveTypingClass* con dos *instance variables*:

```
Object subclass: #LiveTypingClass
    instanceVariableNames: 'instVariable anotherInstVariable'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'LiveTyping'
```

Al crearla por primera vez, el *instanceVariablesRawTypes* va a tener dos elementos, el primero para *instVariable* y el segundo para *anotherInstVariable*:



Ambos arreglos se generan con 10 *slots* por defecto como mencionamos y cada uno de los elementos del array se inicializan con *nil*. A medida que a las variables de instancia se le asignen objetos, *LiveTyping* irá actualizando los *rawtypes* con las clases de la cual son instancia esos objetos. Supongamos que tenemos el siguiente mensaje que actualiza ambas variables, a modo de ejemplo*:*

```
setVariableWith: aValue andAnotherVariableWith: anotherValue
    instVariable := aValue.
    anotherInstVariable := anotherValue.
```

Si lo ejecutamos una vez con un *String* y un *SmallInteger*:

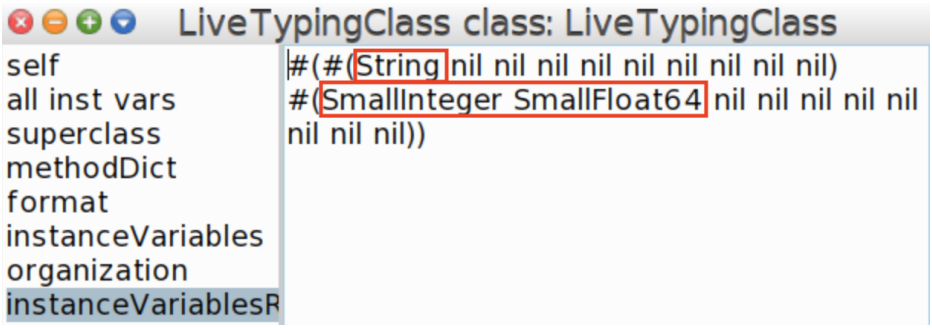
```
(LiveTypingClass new)
    setVariableWith: 'aString'
    andAnotherVariableWith: 1.
```

Al volver a revisar el *instanceVariablesRawTypes* veremos que el primer *rawTypes* contiene a la clase *String* y el segundo a la clase *SmallInteger:*

Si ahora, en la ejecución del sistema se vuelve a ejecutar el mensaje con un *String* y un número flotante, por ejemplo:

```
(LiveTypingClass new)
    setVariableWith: 'someOtherString'
    andAnotherVariableWith: 1.0.
```

Entonces veremos que el *instanceVariablesRawTypes* no se actualizó para el *rawTypes* de *instVariable* ya que se le asignó el mismo tipo de variable, *String*. En cambio para la segunda variable *(anotherInstVariable)* sí, ya que en esta ejecución se le asignó un *SmallFloat64.*



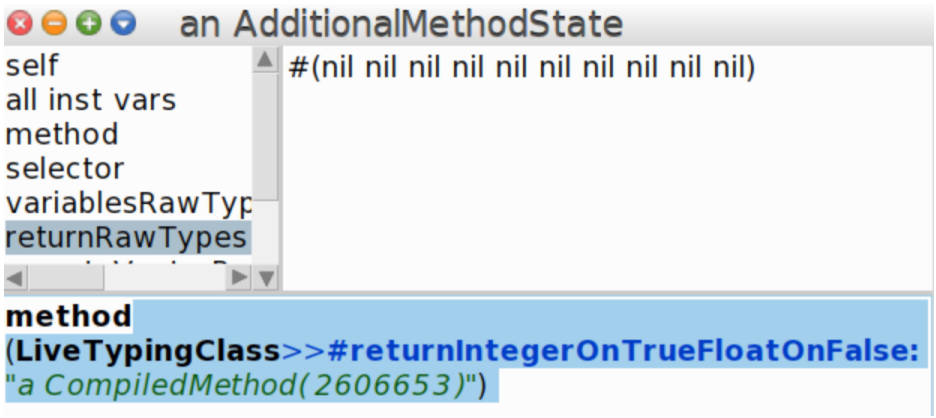
Tipo de Retorno

Un método puede devolver instancias de distintas clases. Es por ello que *LiveTyping* recolecta todos los tipos de retorno de un método en un array de *raw types*. El array se encuentra dentro del *AdditionalMethodState* de un *CompiledMethod* en la variable de instancia *returnRawTypes.*

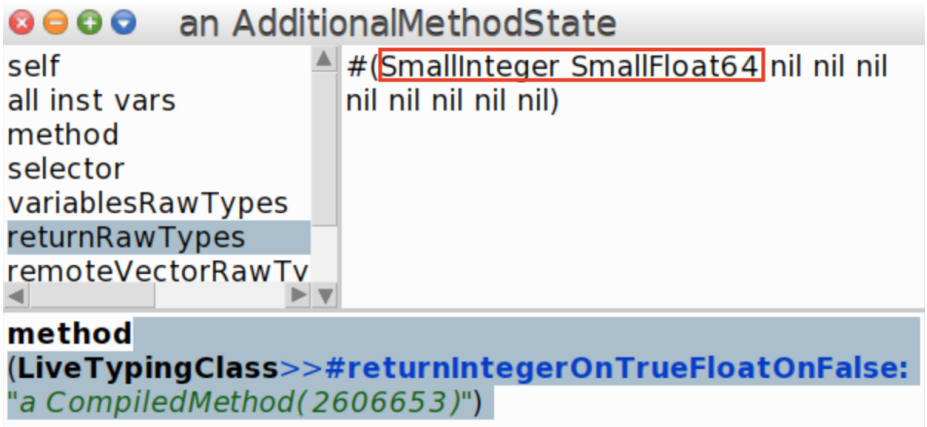
Supongamos que creamos el siguiente mensaje en nuestra *LiveTypingClass*:

```
returnIntegerOnTrueFloatOnFalse: condition
    condition
        ifTrue: [ ↑ 1 ]
        ifFalse: [ ↑ 1.0 ]
```

Al principio, como el método nunca se ejecutó, el *returnRawTypes* de su *AdditionalMethodState* se va encontrar vacío:



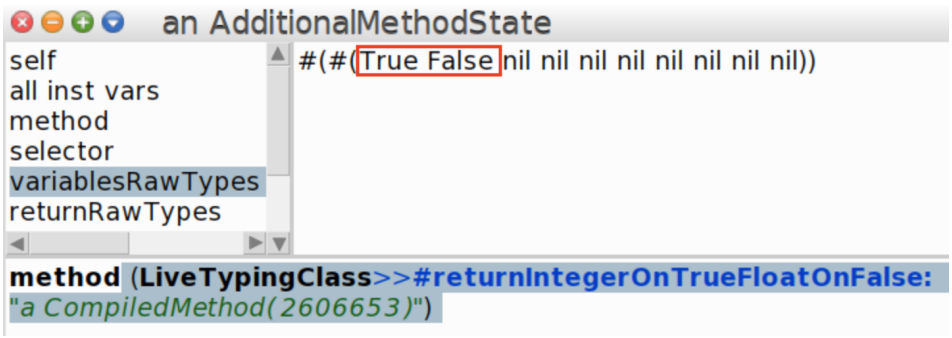
Si luego el sistema ejecuta el envío del mensaje dos veces, la primera vez con *true* y luego con *false* el *returnRawTypes* se verá modificado de la siguiente manera:



Parámetros y Variables Temporales

Utilizando la misma estructura que para variables de instancia, el *AdditionalMethodState* contiene en *variableRawTypes* las referencias a parámetros y variables temporales, las que se encuentran numeradas por cómo quedan definidas en el código fuente de un método. Se utiliza entonces el índice de la variable para acceder al *rawTypes* correspondiente.

Si inspeccionamos el mismo método que en el ejemplo de tipo de retorno, veremos que se referencian los dos objetos booleanos: *True* y *False*. Dado que el método recibe un único parámetro y no hay variables temporales, estas clases corresponden a las dos llamadas a *returnIntegerOnTrueFloatOnFalse:*



LiveTyping como Herramienta

A partir de los *rawTypes*, se puede construir herramientas *(tooling)* que mejoren la experiencia del desarrollador. La imagen de Cuis actualmente posee las siguientes herramientas (Wilkinson, 2019):

- Buscar los *senders* e *implementors* de un mensaje dependiendo del tipo del receptor del mensaje sea variable, resultado de retorno de un método, etc *(actual implementors)* y no de una simple búsqueda del mensaje que de ser implementado en distintas clases puede generar resultados inválidos.
- Mostrar la información de tipos del nodo del AST sobre el cual el cursor está posicionado.
- Mejoras en los refactoring *rename method, add parameter y remove parameter* para que se aplique únicamente a los *actual senders y actual implementors*.
- Mejorar el *autocomplete* para sugerir mensajes en contextos estáticos como el editor de código.

De esta manera, la información de tipos recolectada permite al desarrollador navegar el código y aplicar refactorings eficientemente, ver el tipo de variables en el editor y obtener sugerencias de *autocompletion* a partir de un contexto extendido y específico. Adicionalmente, se permite listar los tipos, agregar o eliminar tipos, rápidamente navegar a sus definiciones y verificar que los tipos coincidan con los requeridos por los métodos utilizados (*type checking*).

Bibliografia

Dolby, J. (2005). Using Static Analysis For IDE's for Dynamic Languages. *The Eclipse Languages Symposium*.

Ebraert, P., D'Hondt, T., Vandewoude, Y., & Berbers, Y. (2005). Influence of type systems on dynamic software evolution. *The Electronic Proceedings of the 21st International Conference on Software Maintenance*.

Endrikat, S., Hanenberg, S., & Robbes, R. (2014). How Do API Documentation and Static Typing Affect API Usability? *International Conference on Software Engineering*.

Fischer, L., & Hanenberg, S. (2015). An Empirical Investigation of the Effects of Type Systems and Code Completion on API Usability using TypeScript and JavaScript in MS Visual Studio. *Proceedings of the 11th Symposium on Dynamic Languages*.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc.

Haupt, M., Perscheid, M., & Hirschfeld, R. (2011). Type Harvesting: A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages. *Proceedings of the ACM Symposium on Applied Computing*.

Hoppe, M., & Hanenberg, S. (2013). Do Developers Benefit from Generic Types? An Empirical Comparison of Generic and Raw Types in Java. *SIGPLAN*.

Kleinschmager, S., Robbes, R., & Stefik, A. (2012). Do Static Type Systems Improve the Maintainability of Software Systems? An Empirical Study. *IEEE International Conference on Program Comprehension (ICPC)*.

Koeppe, I. V. (2018). An Investigation into the Imposed Cognitive Load of Static & Dynamic Type Systems on Programmers.

Petersen, P., Hanenberg, S., & Robbes, R. (2014). An Empirical Comparison of Static and Dynamic Type Systems on API Usage in the Presence of an IDE: Java vs. Groovy with Eclipse. *International Conference on Program Comprehension*.

Pierce, B. C. (2002). *Types and Programming Languages*. The MIT Press.

Röthlisberger, D., Greevy, O., & Nierstrasz, O. (2008). Exploiting Runtime Information in the IDE. *IEEE International Conference on Program Comprehension*.

Tratt, L. (2009). *Advances in Computers* (Vol. 77). Elsevier.

Wilkinson, H. (2019). VM support for live typing: automatic type annotation for dynamically typed languages. *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*.