UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

TypeCheckerDragon: Chequeo de Tipos Contextualizados en un Lenguaje de Tipado Dinámico

Tesis de Licenciatura en Ciencias de la Computación

Julián Francisco Gutiérrez Ostrovsky

Director
Hernán Wilkinson

Co-Director
Agustín Martínez

Buenos Aires, 2024

Resumen

El trabajo de esta tesis se basa principalmente en el desarrollo de una funcionalidad de chequeo de tipos para un lenguaje de tipado dinámico como Smalltalk [1].

Este chequeo consiste en poder verificar para luego informar al usuario de Smalltalk sobre inconsistencias en el código. Principalmente, advertir sobre mensajes que se envían a objetos que no tienen capacidad de responderlo, lo que podría incurrir eventualmente en errores en tiempo de ejecución.

También tiene en cuenta que esos objetos receptores de mensajes varían sus tipos de datos asociados -y por ende también qué mensajes pueden responder- dependiendo del contexto en el que se encuentren. Identificar, analizar y adaptarse a esos contextos también es parte de este trabajo.

Los resultados son presentados de forma unificada en una ventana que exhibe cada inconsistencia encontrada en cada envío de mensaje de cada método, junto con una breve explicación del problema, la previsualización del método en cuestión y la posibilidad de navegar directamente a ese método mediante el uso de browsing que posee Smalltalk.

La técnica utilizada para analizar todos los mensajes del método a verificar, se basa en recorrer el Árbol Sintáctico (AST) generado de cada método en busca de nodos de envíos de mensajes, para luego analizar si los tipos recolectados sobre el objeto receptor tienen la capacidad de responder el mensaje que está siendo enviado.

Agradecimientos

Al Estado argentino, presente.
Su educación pública, gratuita y de calidad
me educó en todos mis niveles
y me permitió llegar hasta acá.

A todos los docentes de la facu que me formaron.
Pero también en particular a León braunstein,
Fernando Schapachnik,
Melisa Scotti, Christian Roldán,
Nicolas Rinaldi, Javier Marenco,
Hernán Wilkinson, Gustavo Hurovich,
Isabel Mendez Diaz.
Los recuerdo especialmente.
Me motivaron e inspiraron pasión y amor
por la docencia, nuestra disciplina y nuestra facultad.

A mis amigos.
A mi novia.
Fundamentales en cada paso que doy en mi vida.

A mis viejos.
Daniel y Adriana.
Siempre.
Soy todo lo que soy gracias a ellos.

A la UBA, en particular al FCEyN y al DC.
Por tantas tardes pasadas en sus labos, pasillos y aulas.

Índice

1. Contexto e Introducción al Problema

Una de las grandes ventajas de los lenguajes de programación dinámicamente tipados es su flexibilidad a la hora de declarar y asignar variables. A diferencia de los lenguajes estáticamente tipados, donde el tipo de cada variable debe declararse explícitamente antes de su uso, en los lenguajes dinámicos los tipos se determinan en tiempo de ejecución. Esto permite que una misma variable pueda contener diferentes tipos de valores en distintos momentos, como un número entero en un instante y una cadena de texto en otro.
Este cambio de tipos en tiempo de ejecución, sin embargo, puede volverse complejo de detectar cuando no es intencional o cuidadosamente manejado, ya que una variable que cambia de tipo en un contexto inesperado podría causar fallos en la ejecución del programa.
Vale aclarar que un tipo de dato es una propiedad asociada a un valor que define qué operaciones se pueden realizar con ese valor (por ejemplo, los tipos numéricos pueden sumarse, las cadenas de texto concatenarse entre sí, las listas de elementos permiten acceder a sus ítems, etc).

Smalltalk [1] es un lenguaje de programación con tipado dinámico, y dentro de sus distribuciones Cuis [2] y Cuis University [3] se desarrolló **LiveTyping** [4], una técnica de anotación automática de tipos cuyo objetivo principal es mejorar la experiencia de desarrollo de los usuarios programadores.
Esta funcionalidad extiende la máquina virtual (VM) de Smalltalk para inspeccionar y registrar el tipo de las expresiones en los programas, tales como variables, argumentos y valores de retorno de los mensajes, mientras estos son ejecutados. La información de los tipos de una variable se obtiene a partir de todos los objetos asignados a ellas (independientemente de cuándo o en que contexto sucedan), mientras que la de los tipos de retorno de un mensaje se obtiene a partir de los objetos que son retornados por él.
Las anotaciones generadas se disponibilizan en tiempo real y sirven como base para desarrollar nuevas funcionalidades.
Este trabajo toma como punto de partida los tipos recolectados por LiveTyping, junto con el trabajo *Inferencia de Tipos Genéricos para Colecciones* (en adelante Generics) [5], que es una versión extendida que incorpora tipos genéricos y un nuevo lenguaje de clases para interactuar con ellos. A partir de ello se desarrolla en esta tesis una funcionalidad de chequeo de tipos llamada **TypeCheckerDragon**[1].
Los tipos genéricos son aquellos que están definidos a partir de su clase base y del uso que se les dé durante la ejecución del programa; un ejemplo típico son las colecciones, que primero son definidas como tal (clase base, una lista de elementos), y más adelante pueden ser definidos los tipos de sus elementos.

[1] Ver apéndice sobre uso de la keyword 'Dragon'

TypeChecker consiste en poder verificar inconsistencias en el código. Más precisamente, detectar envíos de mensajes sobre objetos que, debido a la información de tipos recolectada *hasta el momento* por LiveTyping, no tendrían capacidad de responder, lo que podría incurrir en errores en tiempo de ejecución.

El chequeo de tipos se puede realizar para uno o varios métodos a la vez, así como también para todos los métodos de una clase, o todos ellos dentro de todas las clases de una categoría. Si bien los resultados se presentan unificados, el análisis se hace de manera individual para cada método.

Esta funcionalidad se dispara bajo pedido explícito del programador sobre el/los método/s deseado/s, con la intención de ser utilizado como paso final del desarrollo en un lenguaje de tipado dinámico. La existencia de esta herramienta no interfiere ni actúa de ninguna manera sobre el proceso de compilación, ni modifica los tipos anotados por LiveTyping.

Este trabajo también incluye el desarrollo de la capacidad para detectar e interpretar los bloques de código dentro de condicionales "**IF**", analizando cómo las condiciones afectan a las variables dentro de su contexto. Esto es importante porque estas condiciones en cada rama de un condicional pueden potencialmente asignar un tipo a una variable, restringir tipos posibles, o fijar una serie de tipos dentro de una jerarquía posible. Aunque también puede suceder que las condiciones no afecten a las variables involucradas.
Todos estos casos mencionados, entonces, pueden modificar o incluso cambiar por completo los tipos originalmente asignados por LiveTyping, lo que podría lograr que un aparente error de tipos no lo sea.
Como es de esperar, esta funcionalidad está pensada para múltiples niveles de anidamiento de condicionales y combinaciones lógicas permitidas por el lenguaje.

Método con error de tipos ya que SmallInteger no responde 'isEmpty'.

Método sin error de tipos dentro de sus respectivos contextos.

Los ejemplos exhibidos aquí resultan triviales y puede determinarse a simple vista si el método posee errores de tipo o no. Sin embargo, son a modo ilustrativo para entender el problema a resolver, y debe tenerse en cuenta que el análisis también es realizado en variables de instancia -que pueden tomar tipos en distintos métodos de una clase e incluso en distintas clases de una jerarquía- y también sobre valores de retorno de otros mensajes.
Lo importante es visualizar que el primer ejemplo posee un error de tipos, porque en algún momento de la ejecución la variable var fue asignada con un tipo entero, y esa clase no tiene una implementación asociada al mensaje '#isEmpty'. Mientras que la segunda figura, debido al condicional que distingue si var está o no ante una colección, deja como única posibilidad para la rama '#ifFalse' que el tipo sea entero, y para la rama '#ifTrue' se queda con todos los tipos de la jerarquía Collection.

Cuis también contiene otras dos herramientas pensadas para asistir al programador en tiempo de diseño: balloon helps, cuya finalidad es realizar inspección de los tipos de variables, argumentos y valores de retorno; y code completion de mensajes.
La primera se utiliza para brindar los tipos de un objeto cuando se posa el cursor sobre él. Hoy en día estos tipos son obtenidos directamente desde LiveTyping para ser mostrados como un cartel de ayuda.
La herramienta de code completion es un recuadro que se abre con distintas posibles variantes para completar el código en base a lo escrito hasta el momento. En el caso de estar escribiendo un envío de mensaje, se toman los tipos del objeto receptor para generar el conjunto inicial de sugerencias.
Estas dos herramientas son mejoradas en este trabajo utilizando la detección de contexto para brindar información de tipos más precisa y acorde a cada bloque dentro de un condicional. El segundo ejemplo de los exhibidos arriba funciona como muestra de que se debe mostrar al inspeccionar la variable var posando el cursor sobre ella en esos 3 contextos distintos.

2. Estado del Arte en el Chequeo de Tipos

Contar con algún chequeo de tipos y consistencia en las variables utilizadas a través del ciclo de vida de un programa resulta un aliado estratégico para ayudar a desarrolladores a mejorar la calidad de su código, liberándolo de posibles errores en tiempos de ejecución y rápidamente brindando información de tipos que ya se haya calculado u obtenido.
En muchos lenguajes de programación dinámicamente tipados, estas herramientas a priori no vienen incorporadas y se deben incluir mediante alguna extensión.
A continuación se toman ejemplos que evidencian que no existe un estándar funcional sobre cómo abordar el problema o los resultados, sino que cada lenguaje utiliza una solución adaptada a su contexto, utilidades y facilidades que presenta. Es por eso que dentro de las diferencias hay herramientas que modifican o impiden la compilación; algunas que se basan exclusivamente en anotaciones estáticas, otras que utilizan una solución híbrida entre esas anotaciones y la inferencia; otras en donde el chequeo es obligatorio, y en otras opcional.

2.1 Typescript

TypeScript[2] es una extensión de JavaScript que agrega tipado estático, lo que permite que los tipos se verifiquen en tiempo de compilación. Esto contrasta con JavaScript propiamente, que es un lenguaje de tipado dinámico. Además, TypeScript provee opcionalmente anotaciones de tipos en la declaración de variables, lo que luego ayuda con la validación, ya que si los tipos están anotados no es necesario inferirlos.

El hecho que los tipos sean chequeados en tiempo de compilación es una gran diferencia con el planteo buscado para este trabajo, donde la idea principal es no interferir con el flujo natural de un lenguaje de programación dinámicamente tipado. Queda a decisión del usuario chequear o no los tipos, y poder decidir cuándo es aplicable hacerlo y cuándo no. De esta manera, la intención es conservar todas las ventajas del mundo dinámico de tipos y traer como herramientas las ventajas del mundo estático.

```
function someString() {
    return '10';
}
function main() {
    let num1 = someString();
    num1 = 10;
```

[2] https://www.typescriptlang.org/docs/handbook/

```
    let num2 = 2;
    let result = num1 ^ num2;
}
```
Error de tipos en Typescript, solo usando inferencia.

En el ejemplo, ese código en TypeScript no podrá ser compilado debido a que el motor de inferencia reconoce que la variable num1 toma un tipo que es incompatible con la función exponencial, a pesar de que el código en sí mismo no tendría errores ejecutándose.
Para contrastar, en el trabajo aquí desarrollado para un ejemplo similar, no habrían errores en tiempo de compilación (tampoco de ejecución por el orden de asignación), pero al correr TypeChecker se espera que informe que para la variable num1 el tipo String no responde el mensaje de exponenciación #raisedToInteger:.

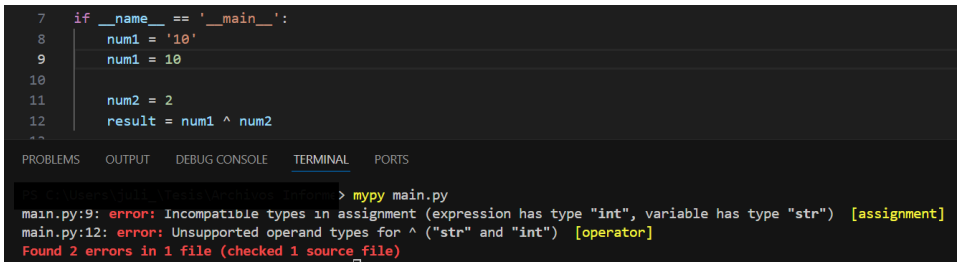
2.2 Python

Python[3] de forma nativa y al día de hoy no posee ninguna herramienta o característica que le permita chequear tipos ni advertir preventivamente sobre posibles errores de este estilo. Sin embargo, existen algunas aproximaciones.
Para empezar, desde la versión 3.5 se introdujeron *type hints* o anotaciones de tipos estáticas. Éstas permiten especificar los tipos de variables, parámetros de funciones y valores de retorno, pero **no son verificadas** en tiempo de ejecución por el intérprete de Python. Solo proporcionan una manera de enriquecer el código como documentación y facilitar la detección de errores manualmente o mediante la combinación con librerías externas.

MyPy[4]

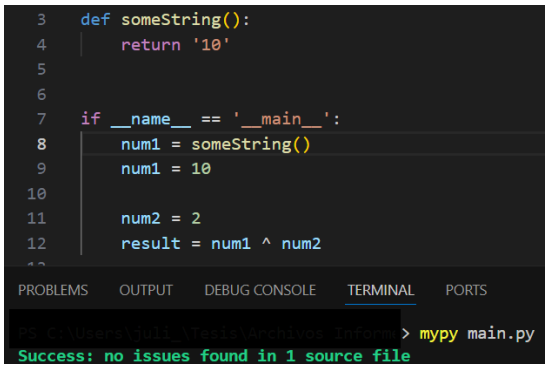
Es una librería de chequeo estático de tipos en Python que al igual que la propuesta de este trabajo, informa de posibles errores de tipos a pedido del programador, sin interferir con el proceso de compilación.
Está pensada para trabajar con type hints de python y su punto fuerte es detectar inconsistencias entre todos los tipos anotados.
De todas formas, también posee un motor de inferencias (aunque limitado), que le permite detectar posibles errores de tipo en un código sin anotaciones.

[3] https://wiki.python.org/moin/
[4] https://mypy.readthedocs.io/en/stable/index.html



Error de tipos en python solo usando inferencia.

Como se ve en la figura, sin anotaciones estáticas detecta incompatibilidad de tipos para num1 y, a pesar de que en tiempo de ejecución no se generarán errores, destaca que la operación puede ser inconsistente con los tipos inferidos.
Dentro de las limitaciones del motor, se puede ver que no hace inferencia sobre retornos de funciones, por lo que cambiando sintácticamente el ejemplo anterior, pero manteniendo la misma semántica, la librería ya no reconoce posibles errores de tipos.



Mismo error de tipos no detectado por el motor de inferencia.

Pydantic[5]

También es una librería de chequeo estático y orientada a la validación de modelos de datos, especialmente al momento de crear instancias de clases. A diferencia de las otras, aquí no existe forma previa de validar los tipos, sino que necesariamente sucede en tiempo de ejecución una vez evaluadas las expresiones.
La diferencia con el uso solo de type hints, es que pydantic realiza un paso previo de intento de conversión a los tipos esperados, antes de fallar. También, si esta conversión falla lanza excepciones que, usadas correctamente, podrían permitir que no se interrumpa el flujo del programa.

```
from pydantic import BaseModel

class Producto(BaseModel):
```

[5] https://docs.pydantic.dev/latest/

```
    id: int
    nombre: str
    precio: float

# Instancia con datos correctos
producto_valido = Producto(id=1, nombre="Laptop", precio=999.99)

# Instancia con un error de tipo (precio como string) que podrá
salvarse
producto_corregido = Producto(id=2, nombre="Mouse", precio="59.99")
# Instancia que no podrá ser creada. Esto lanzará un ValidationError
producto_invalido = Producto(id=2, nombre="Mouse", precio="precio
incorrecto")
```

Ejemplo de validación de datos anotados estáticamente para pydantic.

En la figura se observa un ejemplo básico de uso, donde el tercer caso generará una excepción en tiempo de ejecución. Si esta excepción es manejada adecuadamente entonces se puede generar un mecanismo de fallos inesperados en el sistema.
Esta librería no tiene ningún tipo de motor de inferencia de tipos ni de validador de consistencia de tipos ya cargados para mismas variables o instancias de clases, por lo que solo se apoya en las anotaciones estáticas del código.

2.3 Strongtalk[6]

Es un lenguaje de programación derivado de Smalltalk, diseñado “con un significante número de avances” en cuanto a performance de compilación, y con tipado estático opcional e incremental que funciona independientemente del compilador.
Este lenguaje fue creado en la década de 1990 y se destaca por ser uno de los primeros en introducir la idea de tipado estático en un lenguaje que tradicionalmente era dinámico. También se asegura de mantener la compatibilidad en ambas direcciones, ya que gracias a que el tipado es opcional cualquier programa de Smalltalk es un programa de Strongtalk; y cualquier programa de Strongtalk que se le retira la anotación de tipado se convierte en un programa de Smalltalk [6].
Si un programa en Strongtalk se encuentra estáticamente anotado con sus tipos, entonces cualquier inconsistencia de esta índole será notificada en tiempo de compilación, no permitiendo la ejecución, como en cualquier lenguaje de tipado fuerte.

[6] https://www.strongtalk.org/

Sin embargo, queda a decisión del usuario que métodos anotar y cuáles no, sobre qué métodos chequear tipos y sobre cuáles no y cuándo hacerlo.
En este trabajo, a diferencia de Strongtalk, no hay anotaciones de tipos ni interferencia con el proceso de compilación. La correctitud del análisis radica fuertemente en los tipos provistos por LiveTyping y en su poder de expresividad.

2.4 Smalltalk: LiveTyping

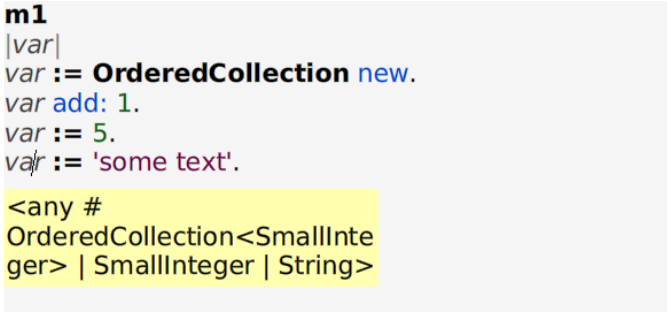
Dentro del universo de Smalltalk Cuis University, existe LiveTyping. Su finalidad es poder almacenar información de tipos recolectada de variables de instancia, temporales, argumentos y valores de retornos de mensajes. Luego se agregan abstracciones para acceder a esta información.

> *Como resultado, LiveTyping es un sistema de anotación automática de tipos para lenguajes dinámicos que busca mejorar la experiencia de desarrollo en estos lenguajes proveyendo información de tipos recolectada en tiempo de ejecución a medida que se evalúan mensajes dentro de métodos y se asignan valores a variables.*

La información de tipos provista por LiveTyping debe ser considerada **información parcial o incompleta**: para cada variable podría darse el caso de que contenga todas las clases posibles de esa variable según todas sus asignaciones (información completa), que contenga solo una parte debido a que no todas las trazas de ejecución que generen información de tipos para esa variable hayan sido evaluadas, o que no contenga información alguna debido a que las asignaciones a esa variable no hayan sido ejecutadas. También puede darse el caso de que una variable sea asignada con más tipos de lo que su objeto asociado *rawTypes*[7] puede almacenar, perdiéndose parte de la información.

Que la información de tipos provista por LiveTyping no sea extremadamente precisa o que pueda estar incompleta no es un inconveniente, ya que el objetivo es que sea suficiente para poder razonar sobre un programa o un fragmento de código que está siendo evaluado y/o construído.

[7] Array asociado a cada variable que almacena la información de tipos inferidas por LiveTyping. Al momento de realización de este informe, posee 10 posiciones fijas.



Inspección de tipos recolectados por LiveTyping para la variable var

A día de hoy algunas mejoras en Cuis Smalltalk incorporadas gracias a LiveTyping son:

1. Inspección de los tipos de variables, argumentos y valores de retorno mediante balloon helps (como muestra la figura).

2. Code completion[8] de mensajes más preciso basado en los tipos del receptor.

3. Búsqueda de senders (envíos de un mensaje) e implementors (métodos que implementan un mensaje) más precisa basada en información de tipos del objeto sobre el que se busca.

4. Refactorings como renombre de selectores, agregado y borrado de parámetros, entre otros, que utilizan información de tipos para determinar el scope de aplicación.

TypeChecker Existente

Aprovechando entonces la existencia de un motor de inferencia de tipos, resulta evidente que puede ser utilizado para chequear la consistencia de los envíos de mensajes en un método. Es decir, comprobar si efectivamente todos los tipos que LiveTyping recolectó para un objeto **obj** al que se le envía un mensaje **#m** tienen la capacidad de responderlo. Dicho de otra manera, si los tipos conocidos de **obj** tienen un método asociado en sus clases o superclases al mensaje **#m**.

[8] Funcionalidad que, a medida que un programador escribe, exhibe sugerencias a modo de autocompletar lo parcialmente escrito.

```
m1
|var|
var := OrderedCollection new.
var add: 1.
var := 5.
var := 'some text'.

var isEmpty.
```

Envío de mensaje isEmpty al objeto var. SmallInteger no sabe responderlo.

En la figura, var es el objeto receptor, que previamente tomó los tipos **OrderedCollection, SmallInteger y String**. **#isEmpty** es luego el mensaje enviado y tenemos una eventual inconsistencia de tipos ya que SmallInteger no tiene un método asociado a ese mensaje.
Vale mencionar que resulta evidente que en tiempo de ejecución en ese método no se producirá ningún error de tipos, ya que **la última** asignación a var es un String y resulta totalmente válido enviarle **#isEmpty**. Pero, por un lado, este es un ejemplo simplificado a fin de visualizar rápidamente los tipos, podría haber casos de métodos más complejos con composición de mensajes, uso de variables de instancia referenciada en otros mensajes, uso de colaboradores internos, etc.
Por otro lado y como se volverá a mencionar posteriormente[9], LiveTyping no recopila información extra sobre la cronología de las asignaciones de tipos que pueda permitir discernir sobre estas cuestiones. Por ende, dentro de cualquier herramienta que utilice como principal fuente de recolección de tipos a LiveTyping, sólo es posible limitarse a esa información y en este caso corresponde informar la inconsistencia.

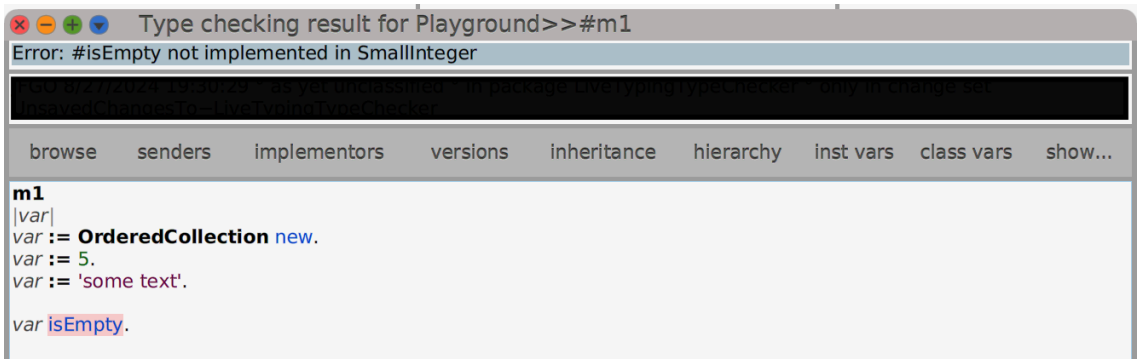
Efectivamente **esta funcionalidad de chequeo existía en Cuis** previo a la realización de este trabajo, y también se había desarrollado bajo el nombre de **TypeChecker**.
Para un método en particular (que no decide explícitamente no anotar tipos), familia de métodos dentro de una clase, o una categoría de clases, es posible realizar un *type check* con el objetivo ya descrito. Como resultado se ofrece una ventana que permite visualizar cada inconsistencia en cada envío de mensaje incurrida en cada método, catalogadas como un *error*, *un warning* o *un problem*, tal que:

- **Error**: Es cuando TypeChecker encuentra efectivamente una inconsistencia de tipos producto de la información provista por LiveTyping y del mensaje que se está analizando. Es decir, que el receptor del mensaje tiene al menos un tipo recolectado que no tiene implementado un método asociado al mensaje.

[9] Ver sección 4. Limitaciones de esta Propuesta

- **Warning**: Se genera cuando TypeChecker no tiene toda la información necesaria sobre el objeto receptor en el envío de un mensaje. Por ejemplo, al enviar un mensaje sobre un objeto que ya tenía previamente otro error de tipos.
- **Problem**: Se genera cuando no existe información para deducir, típicamente debido a que una variable nunca fue asignada. Esto puede deberse a que la traza de ejecución donde se asigna aún no se evaluó; o bien que no existen dichas trazas. Al día de hoy, LiveTyping no tiene capacidad de diferenciar esos dos escenarios.
- **MethodNotAnnotatingTypes**: Forma parte de un Problem. Pero se genera sencillamente porque se ha decidido que el método a evaluar no verifique tipos. Esto puede lograrse enviando el mensaje #removeProperties a cualquier método en cuestión.

Como parte del resultado se despliega una ventana donde se agrega una descripción del problema resaltando el mensaje que lo tiene, una previsualización de todo el método y otras funcionalidades inherentes a Cuis como la posibilidad de navegar hasta la implementación para realizar las modificaciones necesarias, buscar sender, implementors, etc.



Resultado de TypeChecker sobre el método m1, donde se informa que #isEmpty no puede ser respondido por SmallInteger, que es uno de los tipos recolectados de var por LiveTyping.

Limitaciones del TypeChecker Existente

Falta de Integración con Generics

Para empezar, esta versión de TypeChecker fue previa al desarrollo de *Generics* y por ende no contempla el refactoring de las interfaces por este propósito, ni la nueva jerarquía de clases que permite modelar tipos genéricos, tipos no genéricos, unión de tipos y la ausencia de información de tipos:

Ilustración de la jerarquía de clases para tipos que introduce Generics

Una de las mejoras directas que se pueden aprovechar del uso de Generics, es la inferencia de tipos cuando se accede a ítems de alguna colección.

TypeChecker informando un error de tipos por carecer de información sobre los elementos del array.

En un TypeChecker sin uso de Generics var es simplemente <Array>, y por ende no se puede luego inferir de qué tipo es el acceso a algún elemento como muestra el ejemplo, produciendo esto un error de tipos cuando no lo es. Mientras que con Generics incluido, el tipo de var pasa a ser <Array<SmallInteger>> y por ende el tipo de (var at: 1) es SmallInteger

Además la nueva jerarquía de tipos mediante el modelado de *LiveTypes* generó un nuevo lenguaje de mensajes y operaciones disponibles para realizar con ellos, que presentan mejoras respecto de las operaciones anteriores:

- #typesIn: addingIncompleteTypeInfoTo: castingWith:
- #receiverTypesIn: addingIncompleteTypeInfoTo: castingWith:
- #typesOfVariableNamed
- #returnTypes

Estos son algunos ejemplos de mensajes originales de LiveTyping. Para mantener la compatibilidad, el desarrollo de Generics optó por mantener los mensajes originales para obtener tipos de una variable, pero a su vez desarrolló los nuevos que devuelven objetos **LiveTypes** y que podrían aprovecharse en una reingeniería:

- #liveTypesIn: addingIncompleteTypeInfoTo: castingWith:
- #receiverLiveTypesIn: addingIncompleteTypeInfoTo: castingWith:
- #liveTypesOfVariableNamed:storingGenericsInfoIn:
- #returnLiveTypes

Además, Generics también agregó mensajes de enumeración: #liveTypesDo:, #classTypesDo: y #liveAndClassTypesDo:. Estos mensajes reciben bloques como argumentos, abstraen las características concretas de cada **LiveType** y en su lugar permiten razonar en base a los **ClassTypes** o directamente en base a las Clases de cada tipo.
Esto posibilita operar orgánicamente con los tipos de esta jerarquía, ya sea los base o los internos.

Adaptación al contexto incompleta y alto acoplamiento

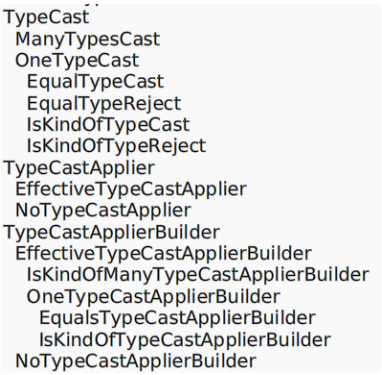
Para que un chequeador de tipos sea versátil debe poder interpretar correctamente cuando existen ramas dentro de un método que de alguna manera fijan o filtran tipos de determinadas variables y actuar en consecuencia. Las ramas en cuestión son las que surgen del uso de condicionales en combinación con los mensajes **#ifTrue:, #ifFalse:, #ifTrue:ifFalse:, #ifFalse:ifTrue**.

```
m1
|var|
var := OrderedCollection new.
var := 5.
var := Date today.
var := 'some text'.

var asLowercase .

var class = String ifTrue: [
        var asLowercase .
    ].

var isInteger ifFalse: [
        var class = OrderedCollection ifFalse: [
            var class = Date ifFalse: [var asLowercase].
        ].
    ].
```

Tres contextos distintos para la misma colaboración var asLowercase

De la figura se deduce que si bien se trata de la misma colaboración, solo en la primera corresponde informar un error de tipos. Ya que en la segunda, dentro del bloque #ifTrue:, var fue fijada directamente con tipo String. En la tercera colaboración, por su parte, mediante anidamiento de bloques #ifFalse: todos versando sobre var se le va filtrando de a un tipo a la vez hasta quedarse únicamente con String también.
Estos resultados son un comportamiento deseable y correcto, y funcionalmente estaban cubiertos por el TypeChecker existente.

Sin embargo la ingeniería realizada para lograr esto terminó siendo por demás compleja, y también incompleta.



Jerarquía diseñada originalmente para poder interpretar ramas dentro de cada método

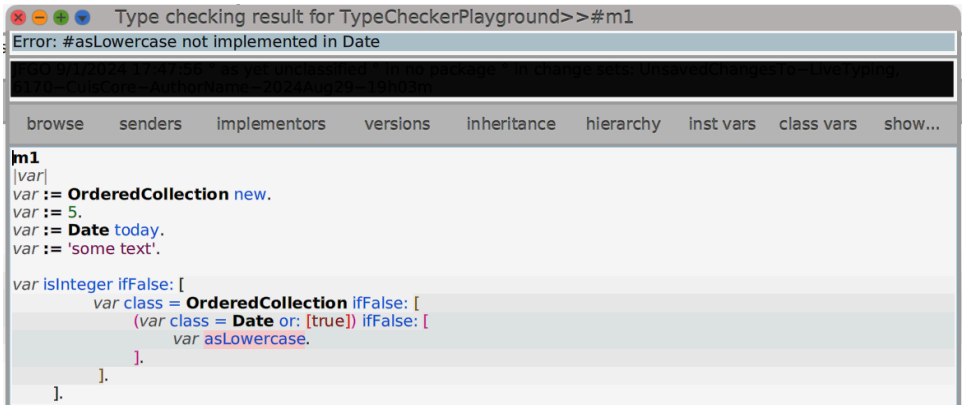
Compleja porque se utilizó una estrategia de construir clases de *Casts* y *Rejects* a aplicar, pero separada por tipos de casteo en Smalltalk (Equals, isKindOf, aunque no mensajes de estilo isType)[10]; a su vez combinada con los appliers efectivos de esas operaciones, y con builders para construir cada caso particular, siguiendo ese patrón de diseño. Además, esta lógica estaba dentro de TypeChecker y estos appliers debían pasarse como colaboradores a los mensajes de LiveTyping, generando un fuerte acoplamiento entre estas funcionalidades, sin una delimitación clara de responsabilidades.
El resultado es una sobre complejización del problema a resolver con una explosión de clases que a priori no parece necesaria.

Incompleta porque funciona correctamente para fijar tipos y también filtrarlos en múltiples niveles de anidamiento, pero no soporta operaciones booleanas en las condiciones de las ramas. Concretamente, no contempla los mensajes **#and:** y **#or**, y lo que sucede es que se ignora cualquier tipo de casteo o filtro, llevando a inconsistencias en los resultados presentados.

TypeChecker no cubre el mensaje #and:, y el resultado es el mismo que si no hubiera ningún IF.

[10] Se verá en la nueva solución que existen distintos tipos de consideraciones al fijar tipos como por ejemplo, si se contempla solo el tipo explícitamente escrito o toda la jerarquía que representa.

Se puede observar en un ejemplo ya mostrado anteriormente, como de manera muy trivial es posible generar una inconsistencia al chequear tipos, en este caso mediante el agregado de **#or**:

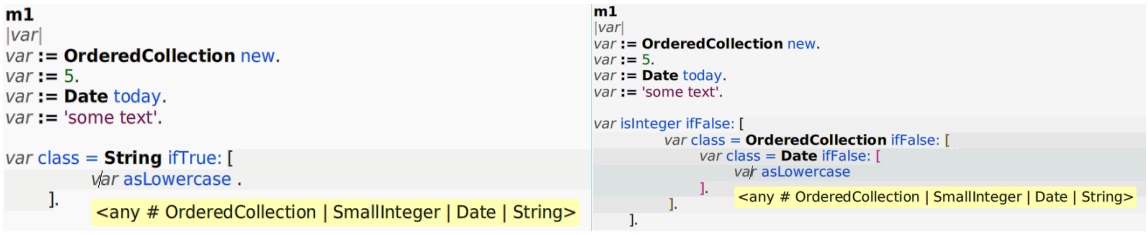


El filtro por Date queda excluido del análisis de TypeChecker y el resultado presentado no es correcto.

Falta de extensión a otras funcionalidades

Como consecuencia del acoplamiento mencionado, ese análisis de contexto e interpretación de los tipos en función de condicionales no pudo ser extendido a otras funcionalidades como inspección de variables y code completion, ya que debía replicarse toda la lógica de Cast Appliers nuevamente y, posiblemente, sobrecargar con éstos a otros mensajes de otras funcionalidades.

Esta falta de cobertura generaba diferencias en los resultados exhibidos y por ende no transmitía robustez, reduciendo la confianza de los resultados obtenidos del análisis para un programador que no esté familiarizado con esta situación



Ambos son casos sin errores de tipos, sin embargo la inspección sobre `var` contradice ese resultado.

3. Abordaje de un Nuevo Chequeo de Tipos en Smalltalk

Basados en las soluciones que se utilizaron en el pasado y que se utilizan en otros lenguajes, para este chequeo de tipos se eligió la inferencia por sobre la anotación, la opcionalidad de chequear, la no intervención del proceso de compilación y el foco en que debe ser una herramienta de apoyo que brinde seguridad, previsibilidad y confianza para utilizarla.

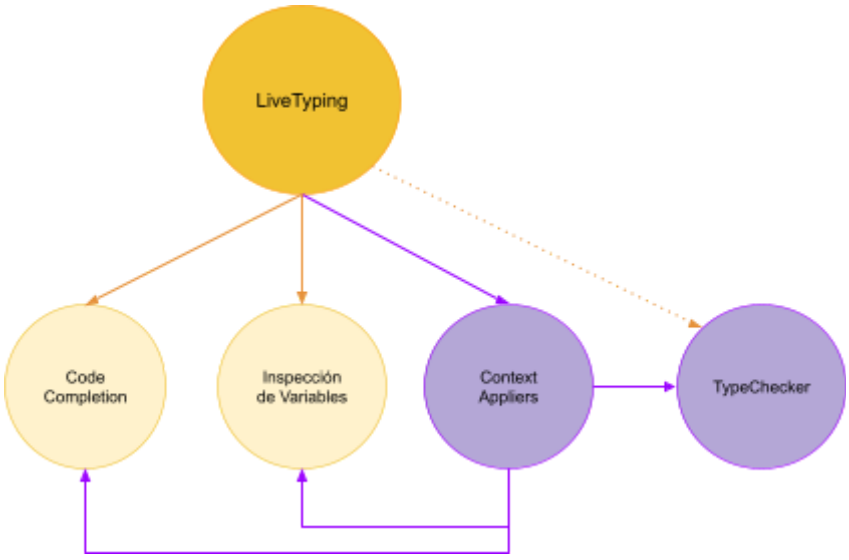
La propuesta consiste en el **diseño de TypeCheckerDragon,** un chequeador de tipos que posee las mismas capacidades principales que el anterior, pero con una reingeniería de sus responsabilidades que redunda exclusivamente en chequear consistencia de tipos, sin considerar detección de contexto, asumiendo que ya existe, o que, de alguna forma, los tipos siempre son los correctos.
Mientras que **esa adaptación de los tipos al contexto de condicionales, se rediseña como una funcionalidad adicional,** polimórfica con todos los mensajes relacionados a obtención de tipos de LiveTyping, pero con la particularidad de manipular esos tipos según amerite.

De esta manera, un chequeador de tipos que solo funciona correctamente en casos sin condicionales, puede rápidamente adaptarse a uno que sí los incluye con un esfuerzo extra mínimo de desarrollo.

Para la adaptación al contexto, también se contempló la solución original a fin de proveer una mejora en su diseño de clases y completar el alcance funcional que fue mencionado como un limitante en la sección anterior.

Por último, al desacoplar y desarrollar como un nuevo satélite la interpretación de contexto, ésta ahora puede ser utilizada por otras herramientas de LiveTyping -además de TypeCheckerDragon- minimizando el acoplamiento.

La idea a nivel global que sigue esta tesis y se plantea en la solución desarrollada, se puede esquematizar de la siguiente manera.



Esquematización de LiveTyping y algunas de sus funcionalidades satélites.
Lo marcado en violeta se desarrolló en este trabajo.

Se puede observar en el esquema que otros satélites de LiveTyping relacionados a este trabajo siguen teniendo relación directa, pero también agregan la posibilidad de consultar tipos adaptados al contexto de un condicional cuando corresponda.

En resumen y para cubrir todo lo mencionado, a nivel general la solución se compone de tres partes principales, donde cada una se analizará en detalle su respectiva sección:

- Desarrollo de TypeCheckerDragon sólo para envío de mensajes en métodos que no contengan bloques condicionales.
- Desarrollo de una nueva solución de análisis de condicionales, para generar adaptación al contexto y completar la funcionalidad de TypeCheckerDragon.
- Integración de esa adaptación al contexto a funcionalidades que también usan LiveTyping, como el autocomplete de Code Completion, y la inspección de tipos de variables con Balloon Helps.

3.1 TypeCheckerDragon

La idea básica de un chequeador de tipos consiste en **dado un método**, detectar envíos de mensajes sobre objetos que, debido a la información de tipos recolectada *hasta el momento* por LiveTyping para cada objeto receptor de esos mensajes, no tendrían capacidad de responder; o dicho de forma positiva, chequear que todos los mensajes de un método puedan enviarse satisfactoriamente en base a todos los tipos de sus respectivos objetos receptores.

En Smalltalk, esto se traduce en poder recorrer todo el código fuente de un **CompiledMethod**, detectar cada envío de mensajes y chequearlo. La mejor forma de recorrer código fuente es mediante la interpretación de su Abstract Syntax Tree (AST) asociado [11].

[11] Ver apéndice sobre qué es un Abstract Syntax Tree.

Jerarquía de ParseNode

Para poder realizar este análisis semántico en Smalltalk, el código fuente de un programa se descompone en instancias de subclases de la jerarquía **ParseNode**, lo que permite que el compilador manipule el código de manera abstracta y eficiente. Así, tomando el primer ParseNode que representa el punto de entrada en un programa o un método, **se tiene su representación como un AST en Smalltalk**.
Un ParseNode genera una abstracción del texto del código fuente ya que representa cada nodo de forma específica como una instancia de clase de su jerarquía. Por ejemplo, un nodo de tipo AssignmentNode representa una operación de asignación, sin importar los detalles específicos de cómo está escrita en el código.

```
a:= 5       +       14.
a := 5+14.
```

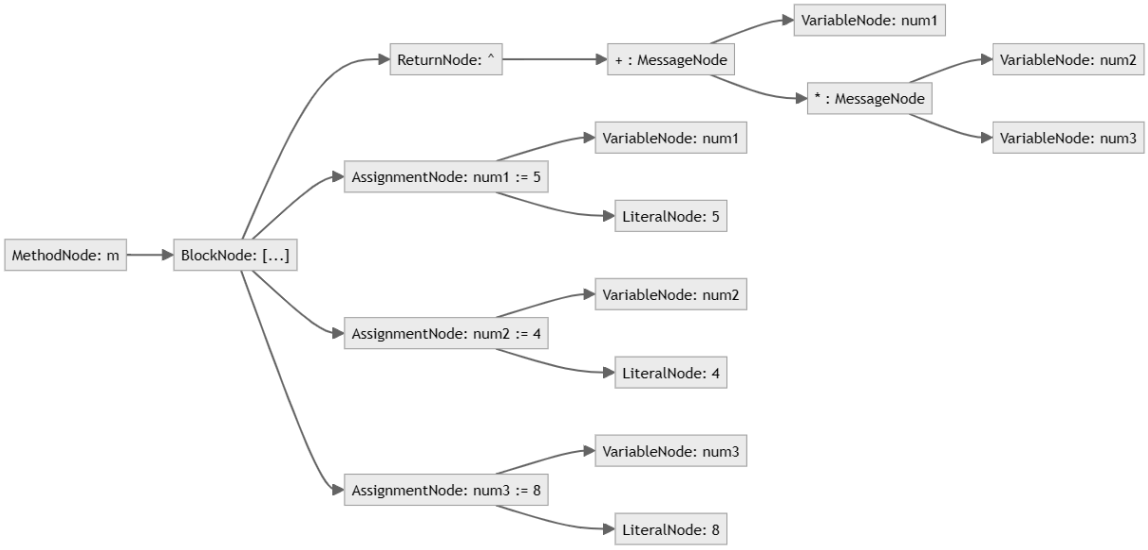
Dos asignaciones semánticamente iguales pero sintácticamente diferentes. La instancia de AssignmentNode que representa a ambas son indistinguibles.

A continuación una descripción de sus clases más relevantes para este trabajo.

Subclase de ParseNode	Descripción
MethodNode	Representa un método completo. Es el nodo raíz en el AST para un método
BlockNode	Representa un bloque de código, una secuencia de expresiones que pueden ser evaluadas. Los bloques son anónimos y se usan por ejemplo, para representar el conjunto general de expresiones de un método, y también las expresiones dentro de condicionales, entre otros.
AssignmentNode	Es la asignación de un valor a una variable. Básicamente, este nodo permite obtener la variable de asignación, y el valor asociado.
MessageNode	Nodo para el envío de un mensaje. Está compuesto por el receptor que es otro nodo; un selector que puede interpretarse como un símbolo, o también como otro nodo; y una colección de argumentos donde cada uno es otra instancia de ParseNode
LiteralNode	Representa un valor literal como un número, un string, una fecha.
ReturnNode	Es una declaración de retorno, que viene acompañado del nodo que se devuelve.

Estas clases de ParseNode se organizan jerárquicamente, formando un árbol. El nodo raíz podría ser un MethodNode, que contiene otros nodos como BlockNode, AssignmentNode, etc., que a su vez pueden contener otros nodos.
A continuación se exhibe un método y su descomposición como instancias de ParseNode.

```
m

|num1 num2 num3|
num1:=5.
num2:=4.
num3:=8.
^num1 + (num2 * num3)
```



ParseNodeVisitor

En Smalltalk es posible recorrer el AST de un método con un **ParseNodeVisitor**.
Como su nombre lo indica, sigue el patrón de diseño visitor [7] y permite usarse sobre instancias de ParseNode.

Esto es útil en este trabajo, para recorrer los nodos partiendo del MethodNode de un CompiledMethod, en busca de cada MessageNode, sin tener que cambiar la implementación de los nodos mismos, sino simplemente generando una nueva clase **TypeCheckerDragonMethodVisitor** que subclasifica a ParseNodeVisitor y redefina el mensaje **#visitMessageNode:** .

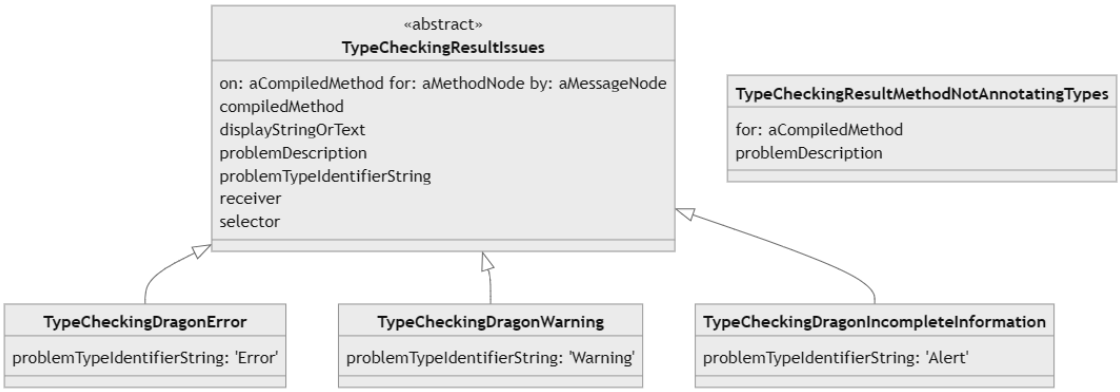
Retomando el desarrollo de TypeCheckerDragon, **la solución se desarrolló y cubrió funcionalmente mediante la técnica TDD**[12], partiendo de los casos más simples que permitieron explorar progresivamente el AST, hasta los más complejos de manera incremental.

Esta primera parte del abordaje se enfoca principalmente en realizar el chequeo de tipos **de un único método**, con el objetivo de buscar inconsistencias entre los tipos de los objetos receptores y los mensajes enviados, para todos los mensajes. Por supuesto entonces, TypeCheckerDragon podrá ser tan correcto como lo precisa que sea esa obtención de tipos.
Luego, esta funcionalidad es generalizada hacia chequear varios métodos e informarlos unificadamente. Parte de esas funcionalidades de extensión junto con mensajes necesarios para mantener la interfaz con la ventana de Smalltalk que muestran los resultados, se obtuvieron de la versión anterior.

Como estrategia de punto de entrada, el enfoque se puso en determinar qué pasa cuando se evalúa el mensaje **#typeCheck** sobre un método que tiene el typing deshabilitado, y luego lo mismo para un método completamente inocuo (vacío) pero que sí chequea tipos efectivamente. El primer caso generará un tipo de *issue* específico que simplemente informa que no corresponde chequear tipos para ese método. Mientras que el segundo permite empezar a desarrollar la interfaz y los mensajes adecuados para empezar con un chequeo de tipos, pero no más que eso ya que al no haber elementos en el AST para recorrer, la técnica de TDD no permite avanzar en ninguna construcción de búsqueda de errores.
Entonces se puede observar que para descubrir la estructura a recorrer del visitor y qué hacer en esas visitas, es necesario desarrollar por la negativa (reconociendo diversos errores) y no por el hecho de dar simplemente estructuras válidas de AST, con métodos sin inconsistencias de tipos, ya que si no se generan los distintos tipos de problemas, los tests pasan sin desarrollo mediante, invalidando el uso de TDD.

Al respecto de los tipos de inconsistencias, en esencia se conservaron los conceptos de *error* y *warning* mencionados y utilizados por el TypeChecker previo, pero luego se adaptó la jerarquía de manera de agruparlos bajo el concepto de *issue*, junto también con lo que originalmente se llamaba *problem*, pero que a partir de ahora se llama *alert*, y se crea como una instancia de TypeCheckingDragonIncompleteInformation.

[12] Esencia y Fundamentos de TDD (https://academia.10pines.com/topics/7/videos/22)



Nueva jerarquía de clases para informar inconsistencias de tipos.

Chequeo de Tipos en un MessageNode

Una vez abierto el camino mediante el uso de TDD para implementar el visitor como ya fue descrito, el método central es el de la **visita de un messageNode**. Es justamente en estos nodos donde luego de localizar el objeto receptor y el selector del mensaje que se está enviando, se deben buscar inconsistencias.
Más allá de considerar el carácter recursivo de un **messageNode** (donde el nodo receptor podría ser también otro **messageNode**), el aspecto fundamental aquí es poder **obtener el conjunto de tipos del objeto receptor del mensaje**.

```
visitMessageNode: aMessageNode
    | incompleteTypeInfo receiverLiveType |
    aMessageNode receiver accept: self. 1
    .
    .
    .                                   2
    receiverLiveType:=aMessageNode receiverLiveTypesIn: compiledMethod addingIncompleteTypeInfoTo: incompleteTypeInfo.
    .
    .
    .
    (receiverLiveType isEmptyType )                    3
        ifTrue: [issues add: (TypeCheckingDragonIncompleteInformation on: compiledMethod for: methodNode by: aMessageNode)]
        ifFalse: [ self checkMessageSentIn: aMessageNode isImplementedIn: receiverLiveType ]. 4

    self visitMessageNodeArguments:aMessageNode.
```

Extracto simplificado del método para obtener tipos en un messageNode.

1. Llamado a visitar el objeto receptor del mensaje que, recursivamente es otro nodo que debe ser explorado. Podría ser también incluso otro messageNode.
2. Mensaje que responde LiveTyping directamente informando los tipos del objeto receptor para el compiledMethod que se está verificando.
3. Primer caso de alerta a informar producto de que hasta el momento, LiveTyping no ha recolectado información de tipos sobre el objeto receptor, por lo que el análisis para ese mensaje no puede continuar.
4. Con los tipos del objeto receptor identificados, es posible verificar que el selector de mensaje se encuentre implementado en todos esos tipos.

```
checkMessageSentIn: aMessageNode isImplementedIn: receiverLiveType

    | implementorsFinder |

    implementorsFinder := AllActualLocalImplementors of: aMessageNode selectorSymbol forAll:
receiverLiveType.
    implementorsFinder liveTypeValue.

    implementorsFinder notImplementedIsEmpty ifFalse: [
        issues add: (TypeCheckingDragonError forNotImplementedTypes: implementorsFinder notImplemented
on: compiledMethod for: methodNode by: aMessageNode)].
```

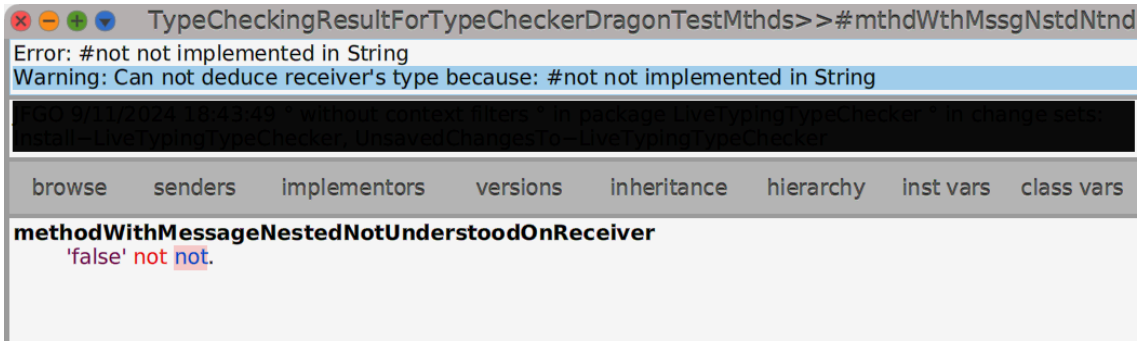
Extracto del método para evaluar la implementación del selector en los tipos del objeto receptor.

```
liveTypeValue

    notImplemented := OrderedCollection new.
    implementors := IdentitySet new.
    types liveTypesDo: [ :aLiveType |
        (aLiveType liveClass lookupSelector: selector )
            ifNil: [ notImplemented add: (NotImplementedMethod class: aLiveType liveClass selector: selector ) ]
            ifNotNil: [ :method | implementors add: method ].
     ].

    implementors := implementors collect: [ :method | method asMethodReference ].

    ^self
```

Extracto del método que busca el selector del mensaje en cada tipo.

AllActualLocalImplementors es un clase diseñada como *method object* cuya finalidad radica en recorrer todos los tipos recibidos buscando la implementación del selector. Luego, en el método #checkMessageSentIn:isImplementedIn: se verifica si algún tipo fue marcado como no implementado, y para todos ellos, **corresponde informar un error por falta de implementación del mensaje en esos tipos.**
Es en el método asociado a #liveTypeValue donde se recorre cada tipo y se verifica la existencia o no del selector en cada uno.

Habiendo cubierto errores y alertas tanto por falta de anotación como por falta de información, solo queda cubrir los warnings. Al momento de consultar los tipos en el **MessageNode** (punto 2 de la figura que los enumera), como resultado adicional se popula una lista (llamada **incompleteTypeInfo**) con todos los warnings, por lo que solo es necesario recorrerla para dejarlos disponibles en la instancia de TypeCheckerDragon y luego informarlos propiamente al usuario.
Actualmente, existen dos tipos de warnings: cuando no se puede deducir el tipo del objeto receptor (por ejemplo porque existe un error previo); y cuando se envía un mensaje sobre un objeto que no tiene tipos explícitos definidos de retorno.

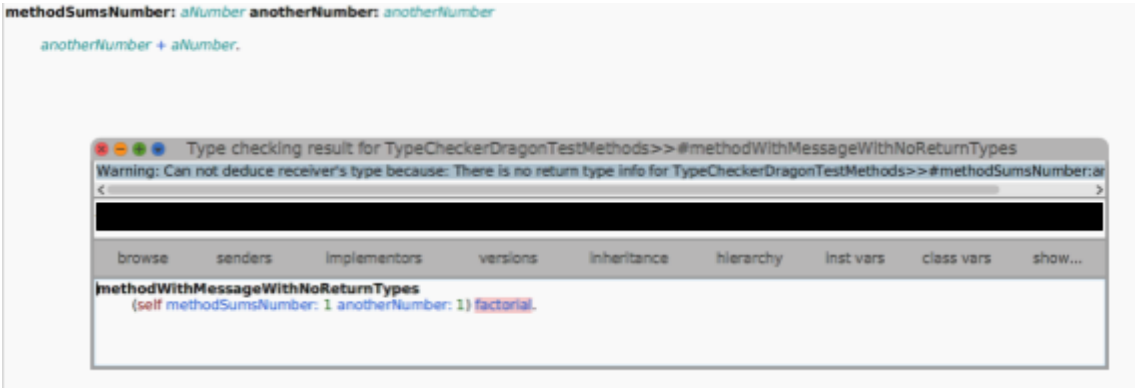


Warning producido por falta de información producto del error en el primer envío de mensaje.

Ambas estrategias para obtenerlos forman parte del hecho de interpretar los objetos receptores. En el caso del ejemplo observado el mensaje **#not** se envía al objeto **('false' not)**. Entonces es necesario obtener primero los tipos de ese objeto. En esa búsqueda se encontrará una inconsistencia producto de que el mensaje

#not, no puede ser respondido por el String **'false'.** Como esa inconsistencia sucede en la búsqueda de tipos de todo el mensaje como objeto receptor de otro, es que se genera un warning. Es decir que a nivel de mensaje interno es un error, pero a nivel del mensaje externo es falta de información. Es por eso que en el resultado final se observan ambos tipos de issues y mientras que el error resalta el **#not** interno, el warning, el externo.

De igual manera, esa interpretación de los objetos receptores, continúa por los tipos de retorno y en caso de no encontrarse ninguno, también se genera un warning.



Warning producido por falta de información del objeto receptor, que es un mensaje sin retorno explícito.

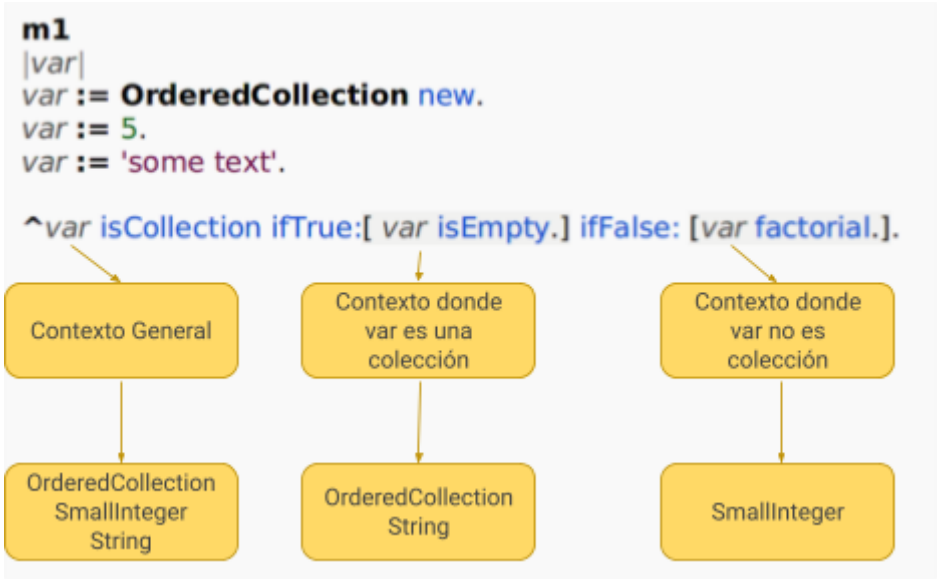
Aquí también hay falta de información por parte del objeto receptor del mensaje **#factorial,** y por eso que a nivel de ese mensaje, se produce un warning.

La versión actual de esos métodos de análisis son esencialmente las que existían en para el TypeChecker original, y en este trabajo fueron adaptadas al uso de LiveTypes. El código completo de la generación de warnings puede verse en el apéndice.

Llegado este punto está satisfactoriamente desarrollada la esencia de un chequeador de tipos para todos los mensajes de un método, aprovechando la implementación del visitor de ParseNode que utiliza object recursion para recorrer todo el AST.
De todas maneras, funcionalmente esto solo cubre los métodos más básicos y sin branches. En la siguiente sección se verá cómo se integran esos casos.

3.2 ContextAppliers para Adaptación al Contexto

La herramienta de ContextAppliers o de adaptación al contexto tiene como principal objetivo dentro de un CompiledMethod en primera medida detectar cualquier combinación de mensajes **#ifTrue:ifFalse:** y luego analizar la condición que versa sobre esos branches. Ese análisis realiza una serie secuencial de filtros que buscan quedarse con aquellas condiciones *conocidas* que puedan tener inferencia sobre los tipos de una variable (como ejemplo típico, la colaboración **v1 class = SmallInteger ifFalse: [...]** está generando una modificación de tipos sobre la variable **v1** dentro del scope **ifFalse**). Que un filtro detecte una condición, implica directamente también conocer cómo parsearla y *qué es lo que busca hacer* esa condición.
Como último paso **se genera un nuevo contexto cuyo scope de vida abarca al subárbol de el/los bloque/s en la rama ifTrue/ifFalse,** y que contiene la información sobre cómo adaptar tipos en la/s variables afectadas.



Ejemplo de distintos contextos y cómo afectan los tipos de la variable var.

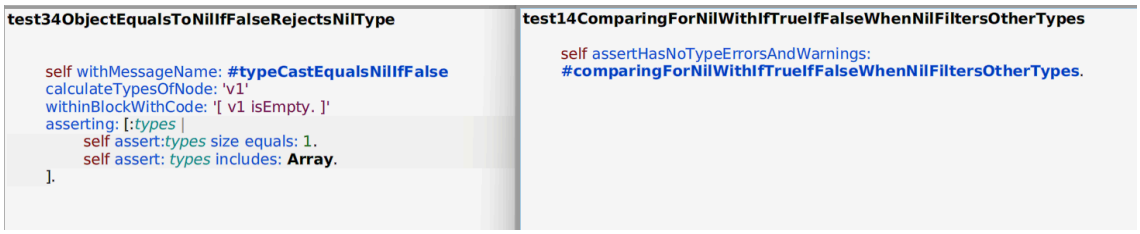
Desarrollo Funcional

La aproximación al desarrollo de interpretación de contexto que existía previamente, estaba en el scope del TypeChecker original, integrada como parte funcional de éste.
Retomando las limitaciones mencionadas en la sección 2 y ampliándolas a una cuestión funcional, seguir esa línea traía dos inconvenientes:

1. La posibilidad de integracion. Al carecer de una interfaz propia, la cohesión de los CastAppliers era baja ya que estaba mezclada con la del chequeo de

tipos. Esto no facilitaba que los CastAppliers puedan ser consumidos por otras funcionalidades de LiveTyping.

2. La declaratividad funcional y de los casos de test. Para ir cubriendo progresivamente nuevos filtros de condición, soporte para múltiples variables, o diversas ramas estando dentro del scope de TypeChecker, los casos de tests eran todos similares. Consistían en tomar un método nuevo para analizar con la condición que se queria cubrir y asertar en todos que el chequeo de tipos no debía tener ningún error. Si bien es técnicamente correcto, la declaratividad de los tests no era buena y solo era posible entender el caso cubierto observando el método a analizar y entendiendo explícitamente cuál era el valor agregado funcional que estaba aportando.



Ejemplo de declaratividad de dos tests que cubren casos similares. A la izquierda, el nuevo desarrollo de ContextAppliers. A la derecha, la implementación previa.

En observación de todo lo dicho, la estrategia de desarrollo funcional tuvo como eje generar **una nueva jerarquía de clases llamada TypeContextAppliers** mediante TDD con una interfaz en donde principalmente se puedan consultar los tipos de un objeto afectados (o no) por alguna condición, y también pueda actualizar a nuevos/restaurar a otros contextos.
Bajo esta idea no solo mejora la posibilidad de integración porque no necesita ni asume la existencia de otras funcionalidades satélites de LiveTyping, sino que también los casos funcionales de tests ganan más declaratividad. Ahora cada test incluye no solo el método con el condicional específico, sino también los tipos que son esperados para un objeto y un contexto dado.

Un detalle que no debe pasar desapercibido a la hora de comprender por completo qué significa *adaptarse al contexto*, es el hecho de que en un método dado, una variable puede tener distintos tipos dependiendo de en qué punto (nodo del AST) se consulte.

```
typeCastIfTrue
    |v1|
    v1:= OrderedCollection new.
    v1 := 'true'.

    v1 class = SmallInteger ifTrue: [v1 factorial].
```

En este ejemplo, los tipos de v1 **dentro del bloque** de #ifTrue (SmallInteger) son distintos a los tipos de la misma variable pero por fuera de ese bloque (SmallInteger y OrderedCollection). Esto quiere decir que el contexto está dado por cada uno de los BlockNode en donde se haga la consulta. Entonces para poder generarlo y luego consultar los tipos de una variable, **es necesario incluir no solo la variable sino también el BlockNode asociado a ella.** Más adelante en la sección de integración se explica cómo pueden obtenerse esos BlockNode en los casos de Inspección de tipos y Code Completion. Por ahora se asume que se tienen.

Un último aspecto a considerar es que no necesariamente las variables consultadas y las afectadas por un bloque contextual son las mismas. A fines prácticos en este informe, en la mayoría de los ejemplos coinciden, pero es importante saber que si se consulta una variable no afectada, entonces los tipos de esa variable son los tipos que existían previamente a ese contexto (que podría ser un contexto anterior, o la información directa de LiveTyping).

```
typeCastIfTrueIfFalseManyVariables
    |v1 v2 |
    v1:= OrderedCollection new.
    v1 := 'true'.
    v1 := 5.
    v2 := 5.
    v2 := Date today.
    v1 class = SmallInteger ifTrue: [v2 month] ifFalse: [v2 year].
```

Ambos contextos de los condicionales sólo afectan a v1. Por ende corresponde informar error de tipos para v2.

Detección y Aplicación de Condiciones

Siguiendo la línea de lo exhibido para TypeCheckerDragon, esta implementación vuelve a utilizar la estrategia del patrón visitor mediante la subclasificación nuevamente de ParseNodeVisitor, por lo que se explora el AST de similar a lo ya explicado. Igual que antes, se cuenta con un CompiledMethod como scope general de aplicación para, esta vez, obtener los tipos específicos **de un nodo dado** en ese método. Si bien parece que esto debiera agregar algún tipo de validación extra sobre la existencia del nodo en cuestión dentro del método, la realidad es que ya sea a través de la herramienta de chequeo, o por inspección de tipos, nunca sería posible obtener un nodo que no corresponda al método.
Por otro lado, se vio también que es necesario el BlockNode dentro del método sobre el cual debe realizarse la consulta, para definir el contexto.

Para abordar esta solución, al crear un nuevo visitor, también se agrega un colaborador interno de TypeContextApplier al que poder consultarle tipos. En el caso trivial (e inicial) este applier es una instancia de un GeneralContextApplier que se comporta similar a un patrón null object (donde el comportamiento correspondería a un "*no contexto*" o "*contexto general*"), redirigiendo los mensajes de tipos que le llega directamente hacia LiveTyping.

Cuando se dispara la consulta, el visitor recorre el AST empezando por el primer BlockNode correspondiente al método y pasando por todo el resto de colaboraciones, incluyendo visitar cada otro BlockNode de argumentos que pueda tener cada envío de mensaje, como sucede en particular con los argumentos de #ifTrue:ifFalse:.
Recorriendo cada uno, eventualmente se llegará al bloque de contexto sobre el que se estaba buscando realizar la evaluación. Allí entonces se realiza la obtención de tipos pedida a la instancia actual de ContextApplier.
Mientras eso no pase, la búsqueda continúa y **los contextos se van actualizando progresivamente** si corresponde.
Otra vez, pareciera necesario contemplar el caso de que el bloque buscado no exista dentro del método, pero igual que antes no es un escenario reproducible en los casos de uso para esta herramienta.

```
visitBlockNode: aBlockNode

    (aBlockNode = contextBlockNode) ifTrue:[ | liveTypes |

        liveTypes:= contextApplier liveTypesFor: parseNode in: compiledMethod addingIncompleteTypeInfoTo: incompleteTypeInfoReasons .
        liveTypes classTypesDo: [:aClass | types add:aClass].
        types:= types asArray .
        ^self.
    ].
    super visitBlockNode: aBlockNode.
```

Método de visita de cada BlockNode en el visitor de ContextAppliers.

La implementación de la visita de cada bloque exhibe todo lo descrito arriba. Mientras el bloque visitado no coincida con el buscado, el visitor sigue inspeccionando el AST (mediante el envío del mensaje *super*), hasta que una vez que lo hace obtiene los tipos del nodo, los convierte a un Array por cuestiones de compatibilidad y corta inmediatamente la búsqueda retornando self.

La actualización progresiva de contextos se da reasignando la instancia de contextApplier actual. Esas actualizaciones ocurren en efecto cuando se está visitando un MessageNode correspondiente a alguna combinación #ifTrue:ifFalse: o #ifNil, #ifNotNil, según se describe a continuación:

Mensaje Condicional	**Descripción**
#ifTrue:ifFalse:	Con el bloque asociado a la rama verdadera como primer argumento y el bloque asociado a la rama falsa como segundo argumento.
#ifFalse:ifTrue:	Que luego el compilador por cuestiones de optimización transforma el selector en #ifTrue:ifFalse: y por ende deja como primer argumento al bloque relacionado a #ifTrue: respetando el orden del caso anterior.
#ifFalse:	Si el argumento está escrito explícitamente como bloque, el compilador genera un primer argumento como bloque vacío, y el de éste queda segundo. Si en su lugar, es una variable representando un bloque, hay un único argumento.
#ifTrue:	El bloque contextual se encuentra como primer y único argumento.
#ifNil:	Ídem #ifTrue:
#ifNotNil:	El compilador lo convierte en un mensaje #ifTrue:ifFalse comparando por igualdad contra nil, y coloca como argumento del branch falso al bloque original.

También es importante notar que no todos los IFs interesan, sino **solo aquellos que versan sobre otro MessageNode como receptor de esos IF**. Esto se da así por la

naturaleza del problema. Para afectar a una variable en sus tipos, ya sea fijando o filtrándolos, es necesario compararla de alguna manera utilizando una operación de igualdad, o chequeando si pertenece a alguna categoría de clases. Ambos casos necesariamente implican un envío de mensaje.

```
typeCastClassSymbolEqualsObjectClassName
    |v1|
    v1:= OrderedCollection new.
    v1 := 'true'.
    v1 := 5.
    #SmallInteger = v1 class name ifTrue: [v1 factorial] ifFalse:[v1 isEmpty]
isIsMessageReject
    |v1|

    v1 := 5.
    v1 := 'test'.

    v1 isString ifFalse: [v1 factorial.]
notNilMessageCast

    | v1 |
    v1 := nil.
    v1 := Array new.

    v1 notNil ifTrue:[ v1 isEmpty.].
boolTesting
| someCheck |
someCheck := true.
someCheck ifTrue: [ ... ]
```

Tres ejemplos y un contraejemplo que ilustran que los condicionales a obtener deben como mínimo tener un envío de mensaje.

Recapitulando, un contexto solo puede ser actualizado ante la presencia de alguno de los mensajes condicionales listados arriba, y solo si además la condición es en sí misma otro envío de mensaje (con excepción de #ifNil, que el selector mismo ya indica la operación de tipo y puede versar sobre una variable directamente).

```
visitMessageNodeArguments:aMessageNode.
    aMessageNode argumentsInEvaluationOrder do: [:argumentBlock |
        contextApplier shouldUpdateFor: aMessageNode visiting: argumentBlock
            ifUpdated: [:newContext |
                contextApplier:= newContext.
                argumentBlock accept: self.
                self restorePreviousContext .
                ]
            ifNotUpdated:[ argumentBlock accept: self. ].
    ].
```

Extracto de visita de un mensaje con argumentos.

De ser así, se obtiene un nuevo contexto para visitar cada bloque (y esto puede volver a repetirse recursivamente con condicionales anidados) hasta encontrar aquel sobre el que se hace la consulta de tipos. Una vez visitado por completo el

bloque y si la búsqueda continúa, se restaura al anterior eventualmente volviendo siempre al general.

```
shouldUpdateFor: aMessageNode visiting: aBlockNode ifUpdated: aBlockWhenContextUpdated ifNotUpdated: aBlockWhenContextNotUpdated

    | newContext |

    aMessageNode isIfWithMessageNodeAsCondition ifTrue: [
        newContext := self buildContextForIfWithMessageNodeAsCondition: aMessageNode over: aBlockNode.
        aBlockWhenContextUpdated value: newContext.
        ^self.

    ].
    aMessageNode isIfNil ifTrue: [
        newContext := self buildContextForIfNilMessagesWith: aMessageNode over: aBlockNode.
        aBlockWhenContextUpdated value: newContext.
        ^self.

    ].
    aBlockWhenContextNotUpdated value.
```

Decisión sobre actualización de contexto. self representa aquí a alguna instancia de contextApplier.

Construcción de un Nuevo Contexto

Hasta este punto, se pudo observar cómo y cuándo se actualizan los contextos, y cómo y cuándo se consultan los tipos. Ahora el foco recae en la diversidad de contextos que pueden existir, y todos los filtros por los que pasa una condición para poder detectar cuál corresponde crear para cada caso.
Antes de eso, algo ya quedó determinado por la sección previa:

> *Cada instancia de un nuevo* ***ContextApplier*** *debe ser al menos polimórfica al mensaje* **#liveTypesFor:in:addingIncompleteTypeInfoTo:**. *Además también deben poder responder el mensaje de actualización a un nuevo contexto y de restauración al previo.*

Como existen diversas formas de poder escribir condiciones sobre variables que modifiquen potencialmente sus tipos, tanto en variedad de mensajes como en variedad de orden, se crearon entonces métodos capaces de detectarlas y también de poder extraer todos los elementos necesarios para crear un nuevo contexto.
Sin embargo, esto no significa que cada manera distinta de escribir estas condiciones, redunde en un *tipo* de contexto distinto. En realidad solo hay seis, con los que se crea una nueva jerarquía de clases que se exhibirá luego con su diagrama. El foco se puso en **minimizar la cantidad de tipos de contexto que se podían construir.** El motivo primario es el de poder mapear rápidamente los estilos de mensajes que se están cubriendo y, de aparecer alguno nuevo, agregar solamente un contexto nuevo, evitando así una innecesaria explosión de clases.
Aquí, la descripción y ejemplo de ellos:

Contexto	Descripción	Ejemplos
EqualsClassType	Representa los casos típicos de cast o rechazo de tipos comparando una variable contra un clase que representa un tipo.	#SmallInteger = var class name var class ~~ SmallInteger SmallInteger = var class nil ~= var
IsKindOfType	Representa los casos de comparación usando el mensaje #isKindOf:, que además tiene en cuenta las subclases de la jerarquía del tipo comparado.	var isKindOf: SequenceableCollection

IsTypeMessage	Representa casos de comparación mediante **cualquier** mensaje que arranque con el sufijo “is”. Además, las implementaciones de esos mensajes deben retornar booleanos.	var isInteger var isCollection var isNil
EqualsToLiteral	Cubre todos los casos de casteo implícito cuando una variable es comparada contra un literal.	var = 5 'text' ~~ var var == false
Logical	Es una colección de otros contextos que son agrupados por conectores lógicos como #and: y #or:	(v1 isCollection and: [v1 class = String]) ((v1 class = Date) not and: [(v1 class = SmallInteger) not] and: [(v1 class = String) not])
General	Representa la ausencia de un contexto específico ya sea por inexistencia o por falta de cobertura.	N/A

Como elementos al momento de determinar el contexto a construir, se cuenta con el condicional completo (un MessageNode con selector IF en alguna de sus combinaciones según ya se vió) y el bloque que se está visitando actualmente (un BlockNode).
Con estos colaboradores se obtiene la condición en sí misma del condicional (un MessageNode como los vistos recientemente en los ejemplos de los tipos de contextos); y se determina si el bloque visitado se encuentra en el bloque de casteo, o en el bloque de rechazo.
Esto último, depende no solo del bloque donde se consulte, sino también del mensaje aplicado en la condición.
Como ejemplo básico, al observar el siguiente mensaje

```
var class = SmallInteger ifTrue:[ ... ] ifFalse: [ ... ]
```

La condición es una comparación por igualdad, y se ve que si la consulta se estuviera realizando sobre el bloque que rechaza (correspondiente a ifFalse), entonces en ese bloque **var** tendrá todos sus tipos, **menos SmallInteger**. Es decir, **los tipos deben ser filtrados.** En caso contrario, el tipo de **var** debería ser fijado (cast) únicamente a SmallInteger.
Ahora bien, si la comparación no fuera por igualdad tal que

```
var class  ~~ SmallInteger ifTrue:[ ... ] ifFalse: [ ... ]
```

Al mismo bloque de antes ahora le correspondería fijar el tipo en lugar de filtrarlo, y viceversa.
Analizando todas las posibles combinaciones en general se tienen los siguientes resultados:

Consulta sobre bloque en ifFalse	Condición pregunta por Igualdad	Debe Filtrar Tipo
V	V	V
F	V	F
V	F	F
F	F	V

Obtener estos resultados es consistente con la operación booleana de equivalencia en Smalltalk y se envía como colaborador a cada contexto construido que lo necesite para determinar qué camino debe tomar cuando sea consultado por los tipos de la variable afectada (casteo o rechazo).

Adicionalmente, otros análisis de la condición deben ser realizados. Para conocer *qué forma* tiene una condición se deben ir haciendo consultas al MessageNode que la representa, en ocasiones, más de una. Luego se debe poder extraer el objeto sobre el que versa la condición y el tipo contra el que está comparando, que puede estar al principio o al final (como se ve en los ejemplos del EqualClassType). Por último y ya dentro de un contexto construído, para poder consultar los tipos se debe tener en cuenta si se lo está haciendo sobre el objeto que afecta el contexto o sobre otro, y si el tipo debe ser filtrado o casteado.
Todo esto redunda en la existencia de varias consultas utilizando mensajes IF sobre las condiciones para determinar todo lo descrito.

Entonces, cada ContextApplier tiene la capacidad de determinar cuál es el siguiente contexto a construir dadas las condiciones descritas anteriormente.

```
contextFor: condition withBlockRejecting: blockIsOnRejectingCondition
    condition isMessageNode ifFalse:[
        ^GeneralContextApplier new.
    ].

    condition isNegated ifTrue:[
        ^self contextFor: condition receiver withBlockRejecting: blockIsOnRejectingCondition not.
    ].

    (condition isOr or: [condition isAnd]) ifTrue:[
        ^self contextFor: condition whenLogicalOperatorWithBlockRejecting: blockIsOnRejectingCondition .
    ].

    (condition isManyOr or: [condition isManyAnd ]) ifTrue:[
        ^self contextFor: condition whenManyLogicalOperatorWithBlockRejecting: blockIsOnRejectingCondition .
    ].

    condition isComparingEqualityForClassWithObjectClass ifTrue:[
        ^self contextFor: condition whenComparingClassWithObjectClassWithBlockRejecting: blockIsOnRejectingCondition.
    ].

    condition isComparingEqualityForClassNameWithObjectClassName ifTrue:[
        ^self contextFor: condition whenComparingClassNameWithObjectClassNameWithBlockRejecting: blockIsOnRejectingCondition .
    ].

    condition isComparingEqualityForClassSymbolWithObjectClassName ifTrue:[
        ^self contextFor: condition whenComparingClassSymbolWithObjectClassNameWithBlockRejecting: blockIsOnRejectingCondition .
    ].

    condition isComparingEqualityForLiteralWithObject ifTrue:[
        ^self contextFor: condition whenComparingLiteralWithObjectWithBlockRejecting: blockIsOnRejectingCondition.
    ].

    condition isIsKindOfWithLiteralBehavior ifTrue: [
        ^IsKindOfTypeContextApplier
            to: condition arguments first key value
            on: condition receiver
            shouldFilterType: blockIsOnRejectingCondition
            withPreviousContext: self.
    ].

    condition isComparingToNil ifTrue:[
        ^self contextFor: condition whenComparingObjectToNilWithBlockRejecting: blockIsOnRejectingCondition .
    ].

    condition isIsTypeMessageWithBooleanReturning ifTrue:[
        ^self contextFor: condition whenSendingIsTypeMessageWithBlockRejecting: blockIsOnRejectingCondition.
    ].


    ^GeneralContextApplier new.
```

Método en ContextApplier que analiza una condición para determinar cómo deben extraerse sus elementos.

Por simplicidad se exhibe uno como ejemplo ya que esencialmente el resto de los contextos tendrán el mismo comportamiento, y solo diferente manera de extraer los elementos. Siguiendo con el ejemplo de la colaboración

```
var class = SmallInteger ifTrue:[ ... ] ifFalse: [ ... ]
```

el mensaje que detecta ese caso es **#isComparingEqualityForClassWithObjectClass**, ya que verifica que haya algún tipo de comparación por igualdad (en este caso el selector **#=** da cuenta de eso), y también que por un lado haya un envío de mensaje class a un objeto y por otro haya un literal (SmallInteger en este ejemplo). Luego solo queda extraer adecuadamente los elementos y crear el nuevo contexto.

```
contextFor: condition whenComparingClassWithObjectClassWithBlockRejecting: blockIsOnRejectingCondition

    | typeToApplyOnCast objectToCast |

    condition isObjectClassComparedToClass ifTrue: [
        typeToApplyOnCast:= condition arguments first key value.
        objectToCast:= condition receiver receiver.
    ] ifFalse:[
        typeToApplyOnCast:= condition receiver key value.
        objectToCast:= condition arguments first receiver.
    ].


    ^EqualsClassTypeContextApplier
        to: typeToApplyOnCast
        on: objectToCast
        shouldFilterType: (self shouldFilterTypeFor: condition knowing: blockIsOnRejectingCondition )
        withPreviousContext: self.
```

Construcción de un EqualsClassType

La forma de extraer los objetos es verificando el orden de apariencia. Entonces si no es que se está haciendo un *object class* contra el nombre de una clase (como en el ejemplo), es porque se está haciendo al revés, es decir que en primer lugar aparece la clase y luego *object class*. Eso determina cómo obtener el tipo a aplicar y cuál es el objeto sobre el que se está aplicando. Para finalizar, también es necesario saber si el tipo debe ser filtrado o casteado según se explicó arriba, y ya se puede construir el nuevo contexto pasando también al actual para luego poder restaurarlo.

Volviendo unos pasos hacia atrás para retomar desde la sección “Detección y aplicación de Condiciones” donde se describía el visitor y su recorrido hasta encontrar el bloque buscado para obtener los tipos de la variable que se deseaba, es posible ahora que fueron descritos los distintos contextos, enfocarse en la resolución del mensaje de pedido de tipos para un contexto que no es el general.

```
visitMessageNodeArguments:aMessageNode.
    aMessageNode argumentsInEvaluationOrder do: [:argumentBlock |
        contextApplier shouldUpdateFor: aMessageNode visiting: argumentBlock
            ifUpdated: [:newContext |
                contextApplier:= newContext.
                argumentBlock accept: self.
                self restorePreviousContext .
                ]
            ifNotUpdated:[ argumentBlock accept: self. ].
    ].
```

Punto de actualización un contexto. Si lo hace, **ese** bloque será visitado bajo **ese** nuevo contexto.

```
visitBlockNode: aBlockNode

    (aBlockNode = contextBlockNode) ifTrue:[ | liveTypes |

        liveTypes:= contextApplier liveTypesFor: parseNode in: compiledMethod addingIncompleteTypeInfoTo: incompleteTypeInfoReasons .
        liveTypes classTypesDo: [:aClass | types add:aClass].
        types:= types asArray .
        ^self.
    ].
    super visitBlockNode: aBlockNode.
```

Método de visita de cada BlockNode en el visitor de ContextAppliers. Los tipos son consultados al contexto actual.

Siguiendo el caso de un EqualsClassType, esta es su implementación para determinar los tipos de una variable dada.

```
liveTypesFor: aParseNode in: compiledMethod addingIncompleteTypeInfoTo: incompleteTypeInfo

    (self shouldApplyTo:aParseNode) ifTrue:[
        self shouldFilter
            ifTrue: [^self liveTypesRejectingObjectFor: aParseNode in: compiledMethod addingIncompleteTypeInfoTo: incompleteTypeInfo ]
            ifFalse: [  ^self liveTypesCastingObjectFor: aParseNode in: compiledMethod addingIncompleteTypeInfoTo: incompleteTypeInfo ].
    ] ifFalse: [
        ^previousContext liveTypesFor: aParseNode in: compiledMethod addingIncompleteTypeInfoTo: incompleteTypeInfo.
    ]
```

Esquema general de obtención de tipos para condiciones de igualdad entre clases y objetos.

Aquí es posible ver que, si la consulta es sobre algún otro objeto que no sea para el que este contexto fue construido, entonces los tipos corresponden al contexto anterior, sea cual fuere éste (en última instancia uno general, es decir, lo que indica LiveTyping).
Caso contrario la resolución depende de si hay que castear un tipo, o filtrarlo.

Castearlo es directo, en el ejemplo el único tipo habilitado para **v1** independientemente de cuáles sean los anteriores, es SmallInteger.
Filtrarlo involucra tomar todos los tipos originales de **v1** y, si existía previamente, eliminar SmallInteger.

```
liveTypesCastingObjectFor: aParseNode in: compiledMethoda addingIncompleteTypeInfoTo: incompleteTypeInfo

    ^FixedType for: type.

---

liveTypesRejectingObjectFor: aParseNode in: compiledMethod addingIncompleteTypeInfoTo: incompleteTypeInfo


    | originalTypes filteredLiveTypes liveType |
    originalTypes := previousContext liveTypesFor: aParseNode in: compiledMethod addingIncompleteTypeInfoTo: incompleteTypeInfo.
    liveType := FixedType for: type.

    filteredLiveTypes := originalTypes asArray reject: [:aLiveType | aLiveType typeName = liveType typeName].
    ^RawToLiveTypesAdapter new adapt: filteredLiveTypes .
```

Extracto de resolución de tipos de un EqualsClassType.

Así queda completamente cerrado el círculo de visita de un método, de actualización progresiva de contextos evaluando sus condiciones y de, una vez encontrado el bloque de interés, pedir los tipos para una variable.

Si bien los otros tipos de contextos siguen una estructura similar, también poseen particularidades que los caracterizan y por eso se crea una distinción.
Tanto IsKindOfTypeContextApplier como IsTypeMessageContextApplier siguen la lógica funcional de poder filtrar o castear. Por este motivo es que, junto con el EqualsClassTypeContextApplier ya visto, se los jerarquiza como subclases de CastingRejectingContextAppliers.

IsKindOfContextApplier

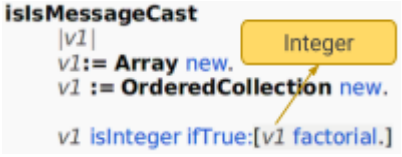
La particularidad que tiene IsKindOf es que contempla las jerarquías de clases. Por ejemplo, ´se puede decir que "cualquier instancia de una OrderedCollection *isKindOf* Collection" (es del estilo o es también una). Lo mismo para cualquier String, que pertenece a la *familia* Collection.
Entonces para determinar los tipos resultantes se deben considerar todos los subtipos del tipo contra el que se está comparando (Collection en el ejemplo). En el caso del casteo, los tipos resultantes serán los originales que formen parte de esos subtipos y en el caso del filtrado los que no formen parte de esos.

Ejemplos de resultados de contexto usando IsKindOf. El segundo debe filtrar ambos subtipos de SequenceableCollection

---

IsTypeMessageContextApplier

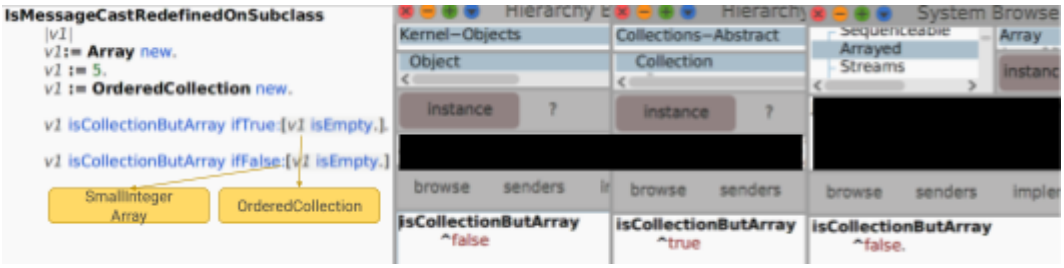
Para el caso de IsTypeMessageContextApplier por un lado se sigue una lógica de casteo propia del EqualsClassType donde, si el tipo fijado por el mensaje *is* no forma parte de los originales, se agregan aquellos tipos que responden **true** a ese mensaje. En el ejemplo, **v1** dentro del bloque será Integer.



Por otro lado, **también** sigue una lógica de filtrado de tipos similar a IsKindOf contemplando jerarquías, pero con la particularidad de que los tipos candidatos ahora son todas las clases que tengan implementado ese mensaje 'is....' y devuelvan **false** o **true** como resultado final, según se esté en el bloque de filtrado o casteo.

Resultados de contexto usando isCollection

En el ejemplo se observa que la clase Collection es quien devuelve **true** para ese mensaje, por ende todas las subclases de collection son tipos posibles para **v1**, y en particular Array y OrderedCollection porque formaban parte de los tipos previos.
En el ejemplo anterior a este eran todas las subclases de Integer, pero como no había ninguna en los tipos previos de **v1** entonces se fija ese tipo a modo de casteo.
Para el caso de ifFalse, SmallInteger devuelve false en la respuesta de ese mensaje por su implementación en Object, por ende es el único tipo de los originales que sobrevive.
Una particularidad de esta implementación es que toda la jerarquía es evaluada desde las subclases hacia arriba en busca del mensaje en cuestión. Por ende si en el ejemplo la clase Array se redefiniera isCollection como **false**, dejaría de aparecer como tipo en el bloque #ifTrue: para aparecer en el #ifFalse:.



EqualsToLiteralContextApplier

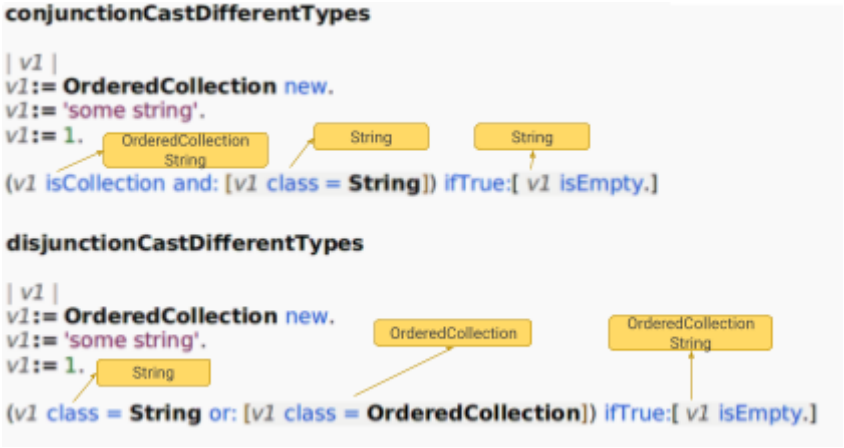
A diferencia de los anteriores **solo funciona con lógica de casteo,** y resolverá en ese caso de la misma manera que un EqualsClassType. La particularidad de este contexto se da porque al comparar un objeto contra un literal

```
var = 'hola' ifTrue: [...] ifFalse:[...]
```

solo es posible afirmar para ese ejemplo que en el bloque verdadero el tipo de **var** es el mismo que el tipo de 'hola' (String). Pero si la igualdad no se cumple, no se puede afirmar nada.

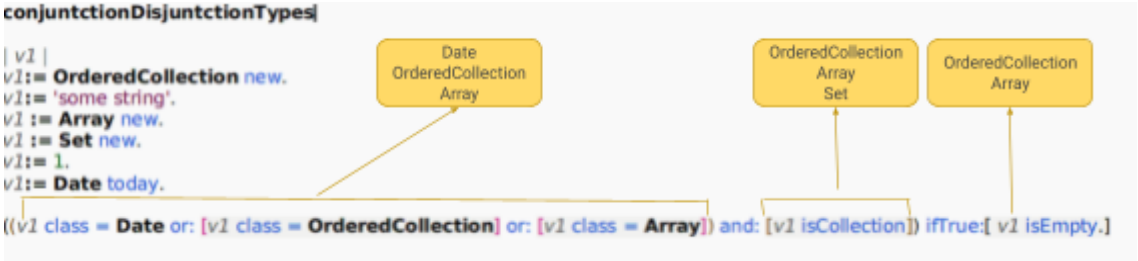
LogicalContextApplier

Este último contexto es, en realidad, una colección de otros contextos unidos por algún conector lógico #and: u #or:.
Básicamente cuando se tienen dos contextos unidos por #and:, el resultado final es la intersección de tipos de ambos contextos. Mientras que si están unidos por #or:, es la unión.



Ejemplos básicos de uso de conectores lógicos.

Esto luego puede extrapolarse a *n* contextos unidos. Lo interesante aquí es que hasta el momento de realización de este trabajo, CUIS University no soporta mensajes combinando #and: y or: en un mismo envío. Sino que soporta hasta #and:and:and:and: y #or:or:or:or:. En caso de querer combinarlos tienen que ser envíos de mensajes distintos agrupados y separados por paréntesis. Por ende, **dentro de un mismo mensaje lógico siempre se hace intersección o siempre se hace unión.**



Combinación de varios conectores lógicos.

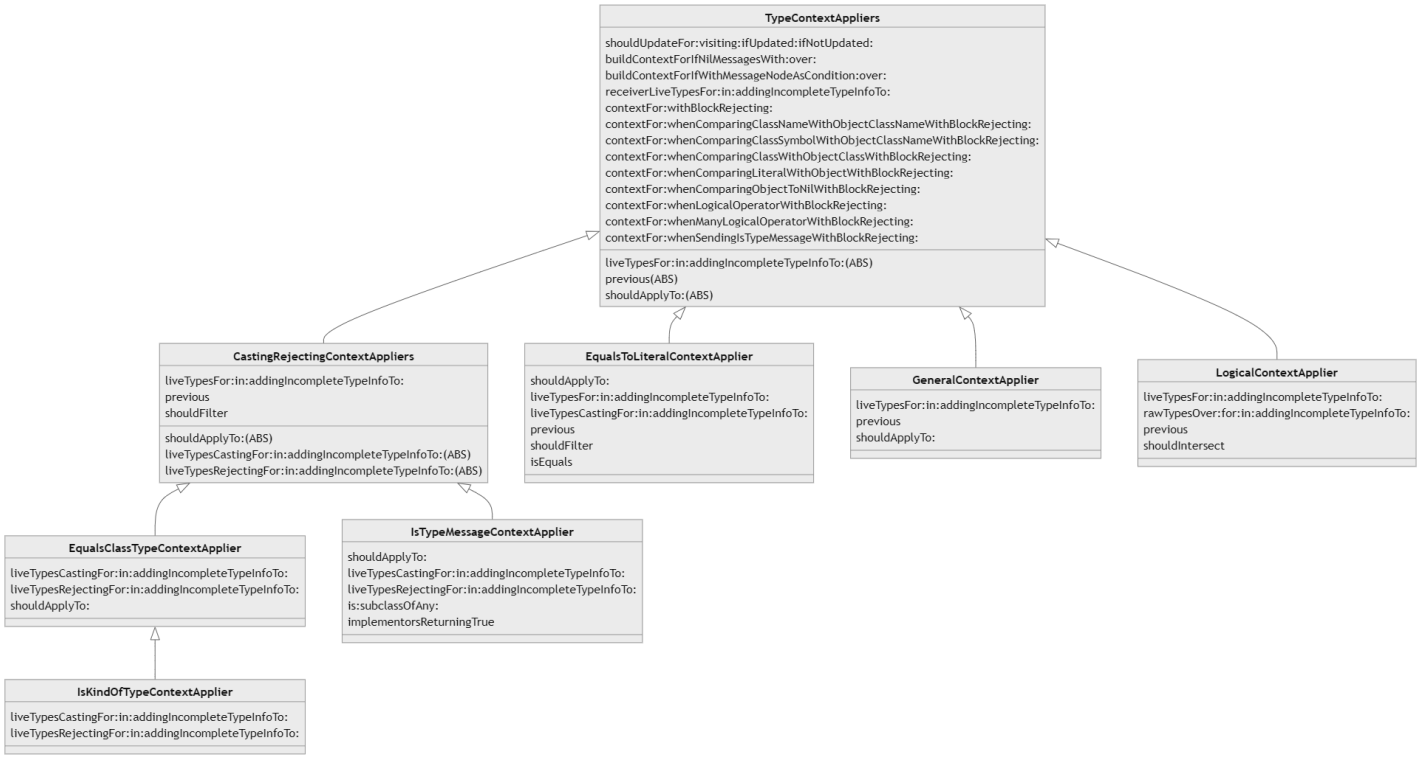
En el ejemplo, el mensaje #and: principal combina 2 contextos, uno lógico y uno de tipo *is*, donde el resultado final será la intersección de los tipos de cada uno. Luego, el lógico interno es una combinación de 3 contextos EqualsClassType unidos mediante #or:, por lo que el resultado parcial de tipos será la unión de cada uno de los tres.

Otro operador lógico es **#not** que, como su nombre indica tiene la particularidad de negar lo que la condición que lo precede está indicando. El efecto que tiene la negación en toda esta construcción de contextos es en realidad la de alternar si un contexto debe filtrar o castear. Se comporta similar a enviar un mensaje por distinción en lugar de igualdad. Es por esto que este caso no es cubierto como un contexto lógico en sí.
Como además este mensaje está siempre sintácticamente al final (por ende es el que primero se detecta en una visita), para cubrir esta funcionalidad la solución será invertir el indicador de si el bloque está en el argumento de rechazo o casteo y recursivamente enviar ese indicador a la construcción de contexto sobre la del receptor de la condición (la condición original sin negar). De esta manera no solo se cubre una, sino todas las negaciones encadenadas que pueda tener un mensaje.

```
contextFor: condition withBlockRejecting: blockIsOnRejectingCondition

    condition isNegated ifTrue:[
        ^self contextFor: condition receiver withBlockRejecting: blockIsOnRejectingCondition not.
    ].
```

Solución recursiva para eliminar negaciones en condiciones.

Luego de haber cubierto todos los tipos de contexto, a continuación se exhibe el diagrama final de clases[13].



[13] Algunos mensajes privados auxiliares se omiten a fines prácticos, ya que no aportan información relevante para comprender el modelo.

Casos con nil

Dentro de TypeContextAppliers se puede encontrar un mensaje especial para casos con nil. Esto es porque existe el mensaje **#ifNil:** que no es parte del set de condicionales cubiertos con **#ifTrue:ifFalse:** y entonces debe contemplarse particularmente.
Si bien pareciera que esto puede generar alguna complejidad extra, la realidad es que son simplemente casos particulares de un EqualsClassType (salvo por el mensaje **#isNil** que seguirá naturalmente el camino de los IsTypeMessage). La única salvedad es que el tipo no se extrae directamente de la condición, sino que se fija en UndefinedObject.

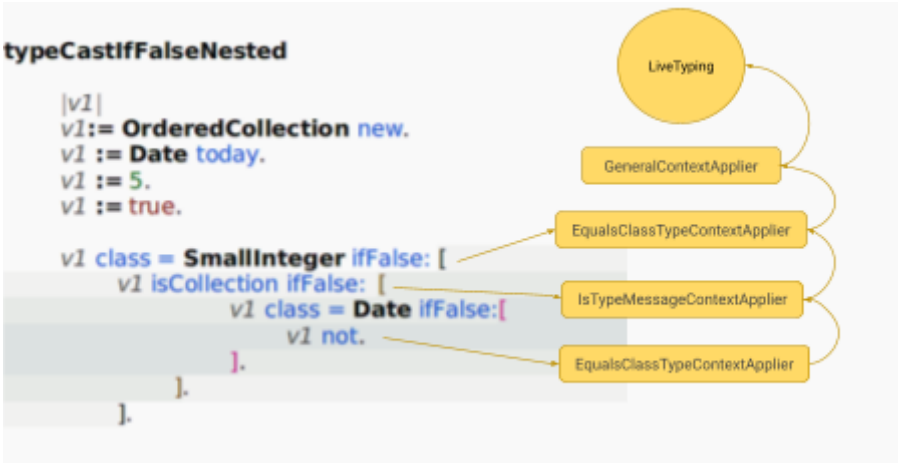
```
contextFor: condition whenComparingObjectToNilWithBlockRejecting: blockIsOnRejectingCondition

    | objectToCast |

    condition isNotNil ifTrue:[
        ^EqualsClassTypeContextApplier
            to: UndefinedObject
            on: condition receiver
            shouldFilterType: blockIsOnRejectingCondition not
            withPreviousContext: self.
    ].


    condition isFirstArgumentNil ifTrue: [
        objectToCast := condition receiver.
    ] ifFalse:[
        objectToCast := condition arguments first.
    ].

    ^EqualsClassTypeContextApplier
        to: UndefinedObject
        on: objectToCast
        shouldFilterType: (self shouldFilterTypeFor: condition knowing: blockIsOnRejectingCondition )
        withPreviousContext: self.
```

Construcción de contexto para casos de comparación contra nil.

Condicionales Anidados

A modo de cierre de esta sección y como resumen de todo lo descrito, todo método empieza con una instancia de GeneralContextApplier que, al consultar los tipos para cualquier objeto, se hace *forwarding* de la consulta a LiveTyping.
Luego, por cada bloque visitado correspondiente a algún tipo de condicional se crea un nuevo contexto con referencia al previo, se visita el bloque con ese como contexto principal, y luego de la visita se restaura al anterior.
Por recursión esto puede crear varios contextos enlazados, aunque no todos versan sobre las mismas variables necesariamente. Lo que se obtiene es una representación como una lista enlazada con un contexto por cada condicional, donde cada nuevo elemento en esa lista representa un nivel más de anidamiento de condiciones en el método.
Para resolver los tipos de alguna variable en el contexto dado se recurre nuevamente a la recursión orientada a objetos, tal que:

- Si la variable consultada no corresponde con la afectada por el contexto, los tipos corresponden a los del contexto previo.
- Si la variable consultada corresponde con la afectada por el contexto, los tipos resultan de tomar como base los previos y realizar la operación necesaria de filtrado y/o agregado pertinente para ese contexto.
- Recursivamente, cada contexto previo realiza lo mismo.
- Eventualmente la recursión llega hasta GeneralContextApplier, caso base.



Ejemplo de contextos encadenados por cada condicional.

3.3 Integración de ContextAppliers a Funcionalidades Satélites

Retomando el gráfico de la Esquematización de LiveTyping y sus funcionalidades satélites, queda ahora cubrir las relaciones entre ellas. Es decir, integrar los ContextAppliers con TypeCheckerDragon -desligando la comunicación directa que tiene con LiveTyping-, con CodeCompletion y con Inspección de Variables.

TypeCheckerDragon

El punto de partida para esta integración es el momento en que se consultan los tipos del objeto receptor de cada mensaje que acepta su visitor, explicado en la sección "Chequeo de Tipos en un MessageNode".

```
visitMessageNode: aMessageNode
    | incompleteTypeInfo receiverLiveType |
    aMessageNode receiver accept: self.
    .
    .
    .
    receiverLiveType:=aMessageNode receiverLiveTypesIn: compiledMethod addingIncompleteTypeInfoTo: incompleteTypeInfo.
    .
    .
    .
```

Consulta de tipos directa al MessageNode.

Esa consulta original obtiene la información directamente desde LiveTyping. En la solución de adaptación al contexto se vio que es el contexto general quien hace *forwarding* del pedido a LiveTyping. Por lo que un primer refactor que no modifica ni agrega funcionalidad podría verse así.

```
visitMessageNode: aMessageNode
    | incompleteTypeInfo receiverLiveType |
    aMessageNode receiver accept: self.
    .
    .
    .
    receiverLiveType:=(GeneralContextApplier new) receiverLiveTypesFor: aMessageNode in: compiledMethod addingIncompleteTypeInfoTo: incompleteTypeInfo.
    .
    .
    .
```

La consulta de tipos ahora pasa por un GeneralContextApplier, que reenvía el pedido a LiveTyping.

Esto claramente no es suficiente. Para contemplar contextos, mediante TDD se agregan casos testigos de los utilizados en el desarrollo de ContextAppliers, ya que muchos de ellos son ejemplos en donde TypeCheckerDragon debería no informar errores. Esos casos testigos servirán para conseguir agregar una variable de instancia en el visitor con el ContextApplier actual (empezando por el General) que se irá actualizando y restaurando progresivamente a medida que se detecta un condicional que versa sobre alguna variable y sus tipos.
Este comportamiento de detección y actualización de un contexto, visita del bloque y restauración al contexto anterior, es exactamente el mismo que se realizó para el ContextAppliersMethodVisitor, por lo que amerita la abstracción hacia una

nueva jerarquía de visitors adaptados a contexto (con la restauración de contexto sucede lo mismo).

```
visitMessageNodeArguments:aMessageNode.
    aMessageNode argumentsInEvaluationOrder do: [:argumentBlock |
        contextApplier shouldUpdateFor: aMessageNode visiting: argumentBlock
            ifUpdated: [:newContext |
                contextApplier:= newContext.
                argumentBlock accept: self.
                self restorePreviousContext .
                ]
            ifNotUpdated:[ argumentBlock accept: self. ].
    ].
```

Esquema general de actualización y visita para visitors adaptados al contexto.

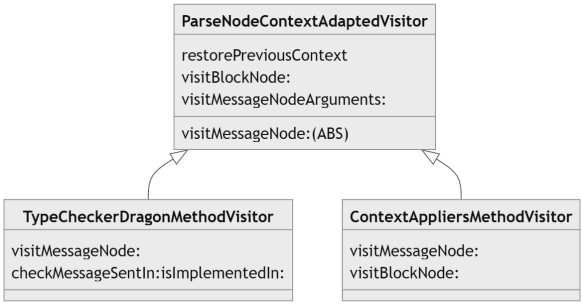


Diagrama final de responsabilidades y jerarquía de los Visitors.

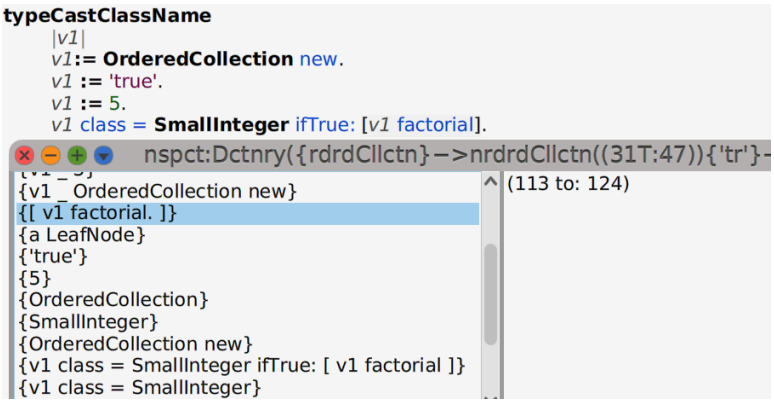
Finalmente, en la visita de mensaje, se termina de reemplazar la instancia fija de GeneralContextApplier, por la de la variable de instancia que está actualizada.

```
visitMessageNode: aMessageNode
    | incompleteTypeInfo receiverLiveType |
    aMessageNode receiver accept: self.
    .
    .
    .
    receiverLiveType:=contextApplier receiverLiveTypesFor: aMessageNode in: compiledMethod addingIncompleteTypeInfoTo: incompleteTypeInfo.
    .
    .
    .
```

Implementación final de consulta de tipos al objeto receptor de un mensaje en TypeChecker.

Inspección de Variables

Esta funcionalidad es la encargada de visualizar mediante balloon helps la información de tipos sobre un objeto que recolectó LiveTyping hasta ese momento. El objeto es aquel que se encuentra debajo del cursor.
La integración aquí consiste en poder detectar si el objeto que está bajo el cursor y se desea inspeccionar, está o no dentro de un bloque en el método, ya que eso es lo que determina un posible cambio de contexto. Si es así, es necesario obtenerlo para luego construir el visitor que recorre todos los nodos hasta encontrar ese bloque y dar los tipos adaptados a su contexto.
La única diferencia con la inspección de variables original es la búsqueda del bloque contextual del objeto, si tal existe. El resto de los elementos, como el método, el objeto a inspeccionar en sí y la llamada a la obtención de tipos ya está resuelta por la versión original.
El dato principal con que se cuenta para obtener todos estos elementos es la posición del cursor en el método al momento de disparar la consulta. El objeto por debajo del cursor se obtiene comparando todos los elementos del diccionario sourceRanges de un método. Ese diccionario contiene todos los elementos del AST asociado al rango en donde se encuentran, con su posición inicial y final.



Ejemplo de diccionario sourceRanges para un método. El bloque factorial ocupa las posiciones 113 a 124.

Lógicamente, no es cierto que siempre haya un bloque contextual para obtener, por ejemplo cuando se inspeccionan variables fuera de cualquier condicional.
La estrategia entonces es la inspección por todos los bloques en el objeto sourceRanges y la comparación entre su rango y la del cursor. Si ningún bloque coincide, implica que el objeto a inspeccionar no está dentro de ningún condicional por lo que se procede con la lógica original de simplemente obtener el objeto y consultar tipos sin contexto.

```
balloonTypeInfoInMethodAt: mousePositionInText

    ^self
        withMethodNodeAndClassDo: [ :methodNode :class |
            methodNode
                withParseNodeIncluding: mousePositionInText
                do: [ :aNodeUnderCursor | self balloonTypeInfoOf: aNodeUnderCursor in: methodNode definedAt: class ]
                ifAbsent: [
                    mousePositionInText <= methodNode selectorLastPosition
                        ifTrue: [ self balloonTypeInfoOf: methodNode in: methodNode definedAt: class ]
                        ifFalse: [ '' ]]]
        ifErrorsParsing: [ :anError | '' ]
```

Método original con la estrategia de obtención del objeto debajo de la posición del cursor.

```
balloonTypeInfoInMethodAt: mousePositionInText

    ^self withMethodNodeAndClassDo: [ :methodNode :class |
            methodNode withParseNodeAndBlockNodeIncluding: mousePositionInText
                do: [ :aNodeUnderCursor :aBlockNodeUnderCursor|
                    self balloonTypeInfoOf: aNodeUnderCursor within: aBlockNodeUnderCursor in: methodNode definedAt: class
                ]
                ifBlockNodeAbsent:[:aNodeUnderCursor | self balloonTypeInfoOf: aNodeUnderCursor in: methodNode definedAt: class]
                ifParseNodeAbsent: [ mousePositionInText <= methodNode selectorLastPosition
                        ifTrue: [ self balloonTypeInfoOf: methodNode in: methodNode definedAt: class ]
                        ifFalse: [ '' ]
                ]]
        ifErrorsParsing: [ :anError | '' ]
```

Nuevo método para intentar obtener el BlockNode y propagarlo, o continuar con el camino original de no haberlo.

De haberse efectivamente obtenido el bloque, se envía un mensaje para poder generar el balloon help que muestre el resultado, con ese bloque como nuevo colaborador, generando una bifurcación con el método original. De no haberse encontrado ningún bloque, se toma el camino original y por ende todos los mensajes originales se mantienen, por lo que no supone ningún esfuerzo extra seguir manteniendo la compatibilidad previa.

```
balloonTypeInfoOf: aNodeUnderCursor in: methodNode definedAt: class

    ^[ ((ParseNodeTypesDisplay of: aNodeUnderCursor in: methodNode definedAt: class)
        calculateTypes;
        initializeTypeInfo;
        typeInfo) printTypesUpTo: 5 ]
        on: MethodNotAnnotatingTypes
        do: [ :anError | 'Could not get type because: ', anError messageText ]
```

```
balloonTypeInfoOf: aNodeUnderCursor within: aBlockNode in: methodNode definedAt: class

    ^[ ((ParseNodeTypesDisplay of: aNodeUnderCursor within: aBlockNode in: methodNode definedAt: class)

        calculateTypesWithContext;
        initializeTypeInfo;
        typeInfo) printTypesUpTo: 5 ]
        on: MethodNotAnnotatingTypes
        do: [ :anError | 'Could not get type because: ', anError messageText ]
```

Bifurcación en cálculo de tipos con y sin bloque obtenido.

Finalmente, dentro del mensaje calculateTypesWithContext se tiene el objeto sobre el que calcular tipos, el método y el bloque a buscar, por lo que están dadas todas las condiciones para generar la búsqueda de tipos contextual.

Code Completion

Bajo este escenario, la premisa es similar a la que había con inspección de variables. Se tiene una lógica funcionando donde se abre un cuadro con sugerencias de posibles mensajes a enviar una vez que se escribió un objeto existente en el método y el programador desea seguir escribiendo. Los mensajes sugeridos resultan de la unión de todos los mensajes que saben responder todos los tipos que tiene ese objeto según LiveTyping.
Nuevamente la integración aquí consiste en detectar si ese mensaje se está escribiendo dentro de algún bloque contextual para obtenerlo y generar el cálculo de tipos adaptado. Si no hay bloque, se procede como originalmente.
Uno de los puntos de entrada de esta funcionalidad es el mensaje computeMessageEntriesFor:in:and: que nuevamente utilizando los rangos del sourceCode del editor en Smalltalk identifica el objeto receptor. Este objeto puede ser self, super, nil, symbol, valores literales y también variables de algún tipo. El camino que sigue luego de la detección de estas últimas es el que debe ser modificado, ya que en los otros, los resultados están determinados por el objeto en sí y no por su contexto.

```
computeMessageEntriesFor: allSource in: contextClass and: specificModel

    | id rangeType |

    canShowSelectorDocumentation := true.
    id := allSource copyFrom: currentRange start to: currentRange end.
    rangeType := currentRange rangeType.

    rangeType == #globalVar
        ifTrue: [ ^self computeMessageEntriesForClass: (Smalltalk at: id asSymbol) class ].
    rangeType == #self
        ifTrue: [ ^self computeMessageEntriesForClass: contextClass ].
    rangeType == #super
        ifTrue: [ ^self computeMessageEntriesForClass: contextClass superclass ].
    rangeType == #true
        ifTrue: [ ^self computeMessageEntriesForClass: True ].
    rangeType == #false
        ifTrue: [ ^self computeMessageEntriesForClass: False ].
    rangeType == #nil
        ifTrue: [ ^self computeMessageEntriesForClass: UndefinedObject ].
    rangeType == #number
        ifTrue: [ ^self computeMessageEntriesForClass: (self classOfLiteral: id in: contextClass) ].
    rangeType == #string
        ifTrue: [ ^self computeMessageEntriesForClass: (self classOfLiteral: id in: contextClass) ].
    .
    .
    .
    rangeType == #instVar
        ifTrue: [ ^specificModel computeMessageEntriesIn: self ofInstVarNamed: id inRange: currentRange ].
    rangeType == #methodArg
        ifTrue: [ ^specificModel computeMessageEntriesIn: self ofTempVarNamed: id inRange: currentRange ].
    rangeType == #tempVar
        ifTrue: [ ^specificModel computeMessageEntriesIn: self ofTempVarNamed: id inRange: currentRange ].
    .
    .
    .
```

Para el caso de estar consultando variables, se agrega un colaborador extra con el rango de la posición en el método para ese objeto.

La forma de obtención del bloque contextual si lo hubiera es la misma que en inspección de variables, y en este caso, los mensajes se modifican para eso de la

siguiente manera, donde el bloque de ifNoContextFoundDo: es la manera original de proceder sin bloque contextual.

```
typeInfoForInstVarNamed: aName inRange:aRange withClass: aClass

    ^self typeInfoForVarNamed: aName inRange: aRange ifNoContextFoundDo: [^aClass
typeInfoOfInstanceVariableNamed: aName ifAbsent: [ nil ].].

---

typeInfoForTempVarNamed: aName inRange:aRange

    ^self typeInfoForVarNamed: aName inRange: aRange ifNoContextFoundDo: [^currentCompiledMethod
typeInfoOfVariableNamed: aName ifAbsent: [ nil ].].
```

Luego, los distintos escenarios donde no hay contexto son lógicamente cuando no es encontrado porque no existe, pero también si aún no hay un método compilado. Este último caso ocurre cuando se está escribiendo un método por primera vez.

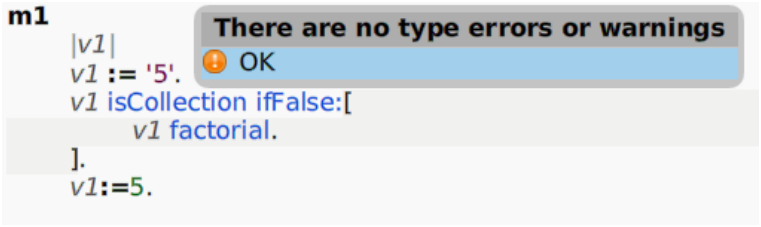
```
typeInfoForVarNamed: aName inRange:aRange ifNoContextFoundDo: aBlock

    | methodNode blockNodeContext variableNode |
    currentCompiledMethod ifNil:[^aBlock value].
    methodNode := currentCompiledMethod methodNode.
    blockNodeContext:=methodNode parseBlockNodeWhenBranchIsIn: aRange start ifAbsent:[^aBlock value].

    variableNode := methodNode variableNodeNamed: aName.
    ^currentCompiledMethod typeInfoOfVariable: variableNode withNode: methodNode withinBlockNode: blockNodeContext.
```

Lógica de mejor esfuerzo para buscar un bloque.

Si existe un bloque de contexto, sumado al método y al objeto sobre el que hacer la consulta, se dispara la búsqueda de tipos contextual.

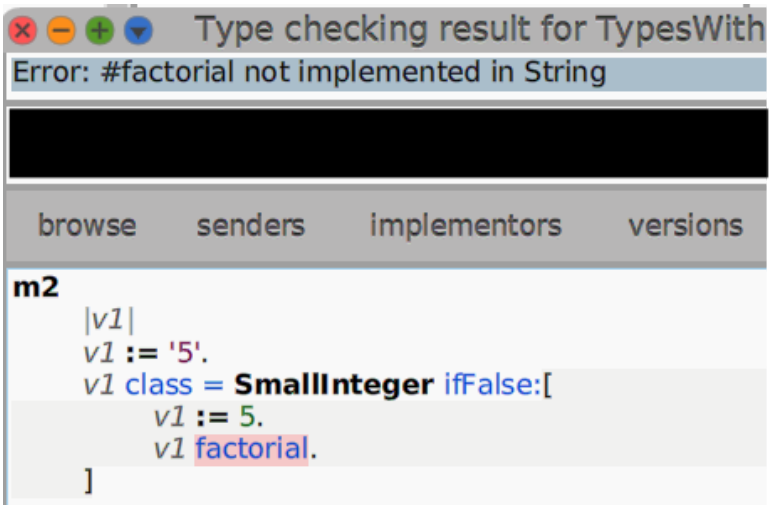
4. Limitaciones de esta Propuesta

1. Una limitación importante en el funcionamiento de TypeCheckerDragon tiene que ver con la falta de información de orden al almacenar los tipos de LiveTyping. Entonces los tipos de una variable son todos los tipos recolectados, sin importar cuando. Esto permite generar errores en el resultado del análisis tal que un mensaje específico de un tipo pueda enviarse a un objeto, **pero ese tipo le sea asignado más adelante en la ejecución.** TypeCheckerDragon no puede distinguir estos casos y falla por no informar el error.



Error de tipos no detectado. Al momento de enviar #factorial, v1 aún no fue asignado con un entero.

2. De la misma manera, si se excluye algún tipo por construcción de contextos, aún si luego se asigna nuevamente más adelante dentro del contexto, no se reconocerá. Esto a diferencia del ejemplo anterior, hará que TypeCheckerDragon informe de un error cuando no existe.

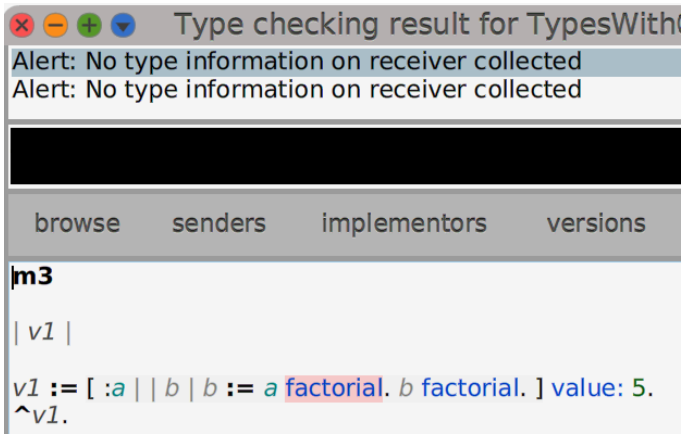


Incorrecto error de tipos informado por TypeCheckerDragon

De todas formas es debatible que esto sea una limitación, ya que el bloque está explícitamente escrito con la intención de eliminar el tipo SmallInteger.

3. Otra limitación se da al evaluar tipos para variables locales y colaboradores dentro de bloques particulares. Al día de hoy, LiveTyping no recolecta esta información. Luego, bajo estas circunstancias, el resultado de TypeCheckerDragon será generar

alertas por falta de información. De todas formas, al depender el problema totalmente de LiveTyping, cuando esa funcionalidad se soporte, no se requerirá desarrollo extra en esta herramienta de chequeo.



Resultado de falta de información para las variables a y b.

4. Ante la eventual aparición de un nuevo formato condicional que predique sobre tipos de variables, es posible que TypeContextAppliers falle en su detección. Esto generará que esa información contextual no sea aplicada y se proceda directamente con un contexto general, que es la misma información que se brindaba previo a la realización de este trabajo.

5. Por la forma en que funciona el ParseNodeVisitor de CUIS recorriendo bloques **por orden de aparición** en el CompiledMethod y no por orden de uso o contexto de uso, es fácil "engañarlo" asignando bloques a variables en algún orden, y luego evaluando esas variables en algún otro orden. Concretamente relacionado a este trabajo, puede crearse el siguiente método

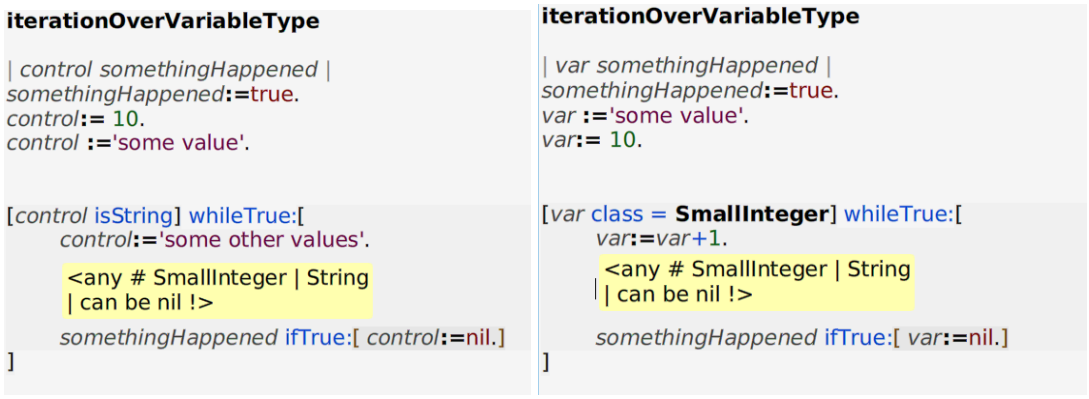
```
m1

| factorialBlock v1 |

v1:='5'.
v1:=5.
factorialBlock:=[v1 factorial.].

v1 class=SmallInteger ifTrue:factorialBlock.
```

Desde el punto de vista de evaluaciones, se ve que no hay ningún error de tipos. Sin embargo, cuando el visitor de TypeCheckerDragon recorre todos los nodos, llega al bloque y pide los tipos de v1, el contexto es el general porque no hay ningún IF mediando. Por ende los tipos de **v1** dentro del bloque son String y SmallInteger generando un error de tipos que debe ser informado.
Además, cuando el visitor efectivamente llega al IF, el contexto se actualiza adecuadamente pero luego el visitor interpreta a **factorialBlock** como una variable temporal, donde no hay nada que chequear ya que no hay envíos de mensajes. Es simplemente una variable.

5. Trabajos Futuros

1. Del punto 3 de las limitaciones, se puede generar una mitigación visual. Para mejorar la experiencia de usabilidad se podría sutilmente cambiar el color de fondo de los balloon help en la inspección de variables cuando la información provista pasó por algún tipo de adaptación de contexto; y conservar el color original si es que la información viene netamente de LiveTyping. De esta manera, sería rápido y efectivo para el programador interpretar la calidad de la información que está recibiendo. Para Code Completion podría hacerse algo similar.

2. Todo el trabajo de adaptación de contextos fue realizado sobre un tipo de *branch* en un método: los condicionales IF. No fue explorado el branch generado por mensajes **#whileTrue:** o **#whileFalse**. Si bien son casos más acotados, es a priori factible escribir una condición de while que también altere los tipos de alguna variable.



3. Al momento de consultar tipos de cualquier manera, resulta evidente en este punto que no es lo mismo hacerlo directamente a LiveTyping que pasando primero por toda la creación contextual necesaria primero, para luego hacer la consulta que, también en última instancia recurre a LiveTyping. En el segundo caso el overhead es mayor y es una constante asociada a la cantidad de branches que tiene un método antes de llegar al buscado. Ningún trabajo de performance fue realizado en esta tesis. **Tampoco hay evidencias empíricas que ameriten hacerlo.**
   Dicho esto y con mayor certeza futura de cómo son utilizados estos contextos, y cómo podrían escalar, podrían considerarse tareas de mejoras respecto de instancias de contexto creadas.
   Por ejemplo, al inspeccionar una variable se sabe de antemano cuál es. Sin embargo, cualquier contexto que verse sobre otras variables también será construido para tener una biyección entre branches y contextos como se detalló en su explicación. Esto no es estrictamente necesario ya que a ojos de la variable

consultada, cualquier contexto sobre otra variable al consultar tipos va a ser ignorado.
Implementar esto podría redundar en la eliminación de la pregunta en cada contexto por si el nodo a consultar coincide con el de cada instancia.

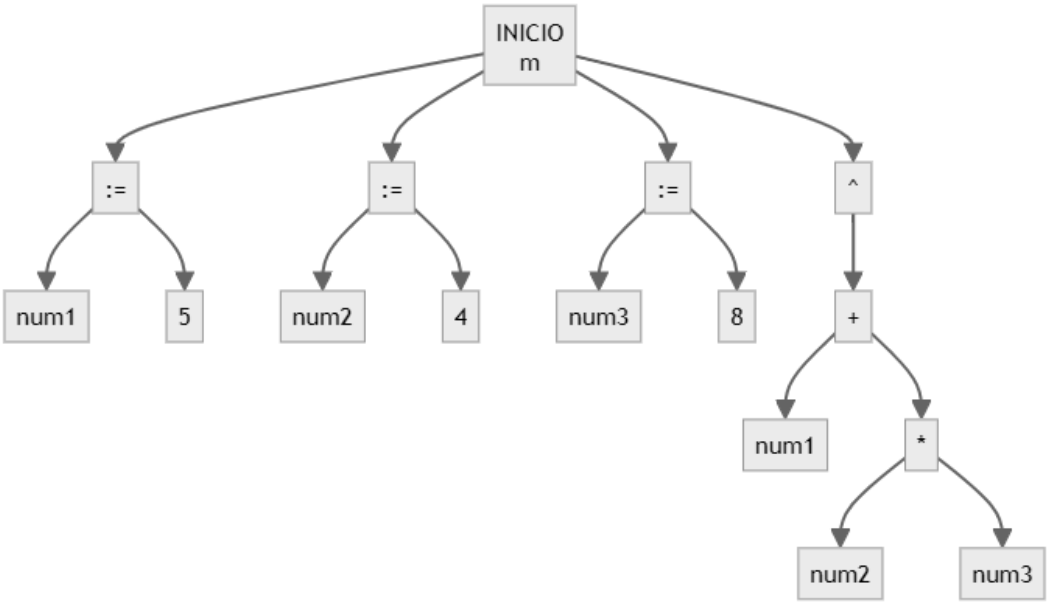
4. Hilando más fino y aún aceptando pagar un overhead por tener información contextual, también es cierto que si se desea inspeccionar el valor de una variable contextualizada apoyando el cursor sobre ella, luego se quita el cursor y se vuelve a posicionar sobre la variable, exactamente los mismos contextos serán construídos dos veces. Por lo que, si fuera necesario, para un compiledMethod (mientras no haya sido recompilado por modificaciones) se podrían tener asociados a un diccionario hash la lista enlazada de cada contexto construido para cada variable e intentar primero recuperarlos de ahí, antes de reconstruirlos.

6. Apéndice

Abstract Syntax Tree

También conocido por sus siglas como AST, es una **representación jerárquica en forma de árbol** de un programa (en este caso, será de un método) que refleja su estructura, sus instrucciones y operaciones, más allá de la sintaxis textual.
El AST organiza el código en nodos que corresponden a las construcciones del lenguaje, como métodos, expresiones, asignaciones, operaciones, y otros bloques de código. Cada nodo puede o no estar conformado por subnodos que también son elementos de un AST, e incluso subárboles en sí mismos.
A modo esquemático y abusando un poco de la notación con fines de acercar la idea, el siguiente diagrama representa un AST para el método:

```
m

|num1 num2 num3|
num1:=5.
num2:=4.
num3:=8.
^num1 + (num2 * num3)
```



Sobre Uso de Keyword *Dragon*

Como puede observarse en la implementación final de este trabajo, todas las clases relacionadas a TypeChecker (su clase en sí, pero también el visitor y las de test) contienen la palabra "*Dragon*" en su nombre. Como sucede muchas veces al diseñar software, fue pensada como una solución temporal que terminó persistiendo en el tiempo.

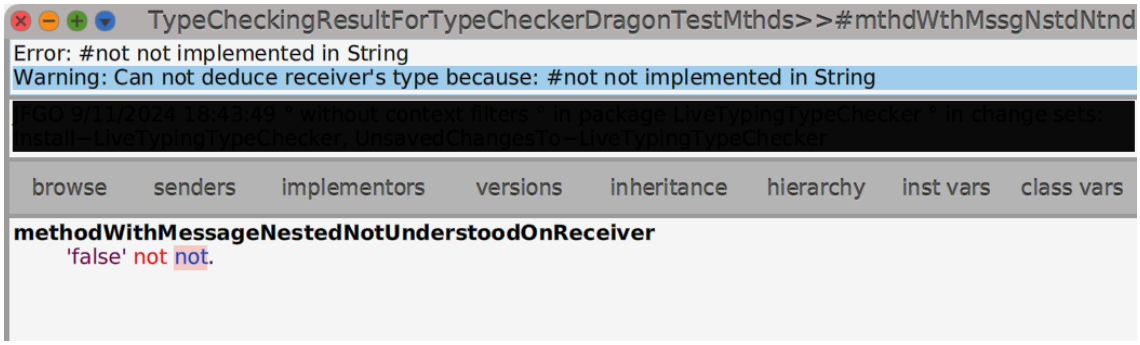
La razón de su existencia es debido a que durante la primer fase de desarrollo, era útil mantener el TypeChecker original para poder usarlo como referencia de comportamiento (tanto deseado como no), y para no tener que lidiar con borrados problemáticos de código cada vez que fuera necesario instalar el paquete nuevo en una imagen limpia.

La palabra Dragon fue elegida totalmente al azar con la idea de luego poder con un editor de texto, reemplazarla por vacío (eliminarla) y de esa manera tener automáticamente integrado el nuevo TypeChecker.

Sin embargo, se mantuvo hasta la versión final por la imposibilidad de, mediante una mera actualización de paquetes de smalltalk, eliminar código de otros paquetes existentes.
Es decir que mediante el nuevo paquete "LiveTypingTypeChecker" no es posible eliminar el TypeChecker original que viene con cada imagen limpia de CUIS. Tenerlos distinguidos por nombre permite en todo momento evitar cualquier tipo de colisión hasta que el original sea definitivamente eliminado de las versiones oficiales y el nuevo tome su lugar, ahora sí eliminando Dragon de todas las clases y referencias con un simple refactor de rename. 🐉

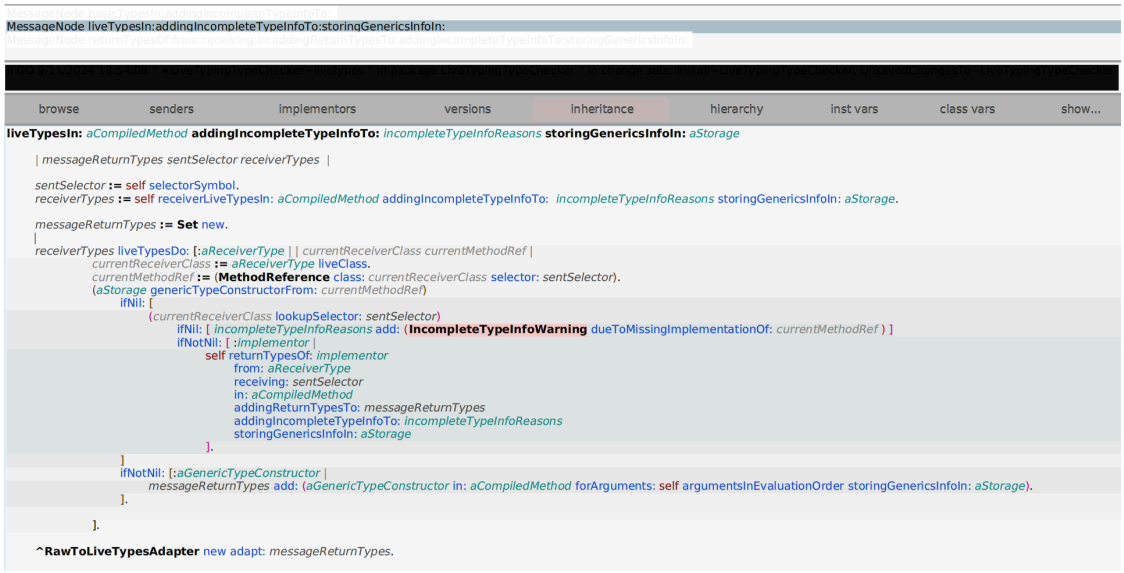
Sobre generación de Warnings en TypeChecker

Para comprender mejor cómo funcionan los métodos que buscan warnings, se toma nuevamente el siguiente ejemplo.



Se tienen dos mensajes **#not.** Uno interno que se envía al String **'false'** y otro externo que se envía al objeto **('false' not)**

El warning debe ser detectado cuando se analicen los tipos del objeto receptor del **#not** externo **('false' not)**, que es un MessageNode. Por ende el mensaje de obtención de tipos para ese objeto, lo responde esa clase.

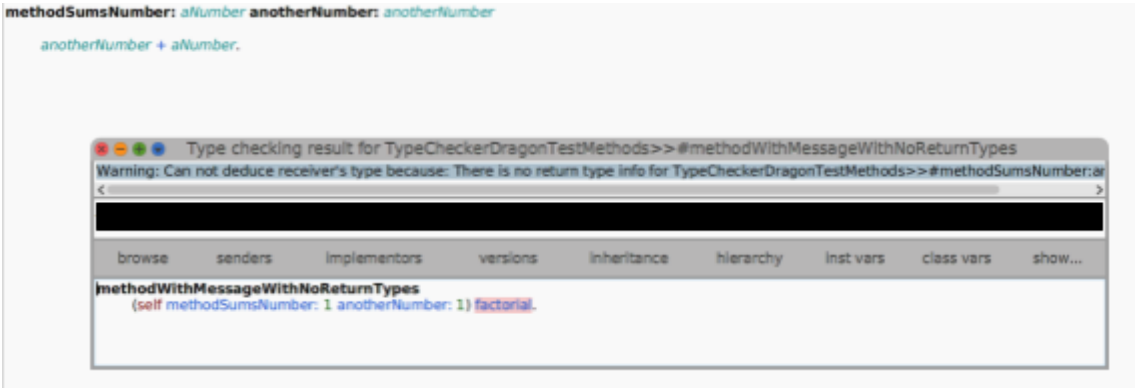


Método de obtención de tipos de un MessageNode. En este caso se deben obtener los tipos de 'false' not.

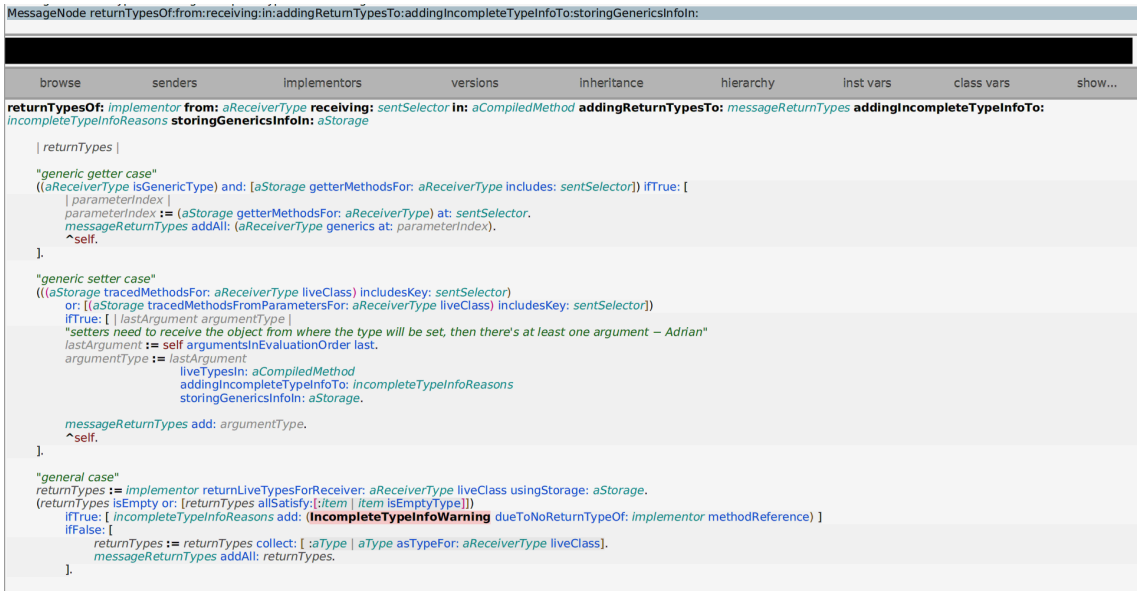
Al ser un MessageNode, obtener los tipos de todo el objeto, implica nuevamente obtener los tipos del objeto receptor del mensaje (que si nuevamente fuera un MessageNode, recursivamente se repetiría la lógica).
En el ejemplo, la lista de receiverTypes tiene solamente String como único elemento. Lo que hace el método luego es buscar la implementación del mensaje enviado en cada tipo. Como **#not** no es respondido por String, se genera el warning.

Luego, al seguir con el análisis, éste mismo mensaje **'false' not** volverá a ser analizado, pero no en contexto de objeto receptor de otro mensaje, sino como mensaje en sí mismo. La misma inconsistencia vuelve a ser detectada pero en ese caso será un error como se vió en la sección "**Chequeo de Tipos en un MessageNode".**

Parte de la búsqueda de tipos de un objeto receptor es también verificar si corresponde los tipos de retorno, si es que no se agregó el warning por el caso anterior.
Para comprender este escenario, resulta conveniente observar el siguiente ejemplo.



A diferencia de antes, el objeto receptor a analizar aquí es un envío de mensaje.
Los tipos de un envío de mensaje, son los de retorno del mismo.
En este caso no se puede determinar los tipos del objeto receptor del mensaje **#factorial**, ya que es un mensaje sin tipos de retorno explícito. Ese chequeo arranca de la misma manera que lo explicado arriba, intentando obtener ahora los tipos del objeto **(self methodSumsNumber: 1 anotherNumber: 1)**. Esa búsqueda de tipos detecta el envío de mensaje y resuelve sus tipos de la siguiente manera.



Método de búsqueda de tipos de retorno en un envío de mensaje.

Aquí el implementor es el método del mensaje y el receiverType es la clase en sí misma. En el caso general de búsqueda se obtienen todos los tipos explicitos de retorno del mensaje (que son todos los ReturnNode del AST). Si no hay tipos, o bien son instancias de EmptyType, corresponde informar el warning ya que el análisis no podrá continuar con su intención original de chequear si #factorial se estaba enviando a un objeto que supiera responderlo.

7. Referencias

[1] A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc, 1983
Available: http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf

[2] H. Fernandes and J. Vuletich, “The Cuis-Smalltalk Book,” 2020.
Available: https://cuis-smalltalk.github.io/TheCuisBook/index.html

[3] “Cuis University: ambiente creado especialmente para la enseñanza de la Programación Orientada a Objetos,” 2020.
Available: https://sites.google.com/view/cuis-university/inicio

[4] H. Wilkinson, “Live Typing: Automatic Type Annotation for Dynamically Typed Languages,” 2018
Available: https://github.com/hernanwilkinson/LiveTyping

[5] R. A. Castiglione, “Inferencia de tipos genéricos para colecciones en ambientes con LiveTyping,” FCEyN, Universidad de Buenos Aires, 2023.

[6] G. Bracha, “The Strongtalk Type System for Smalltalk,” 1993.
Available: https://bracha.org/nwst.html

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994, pp. 331–344.