Universidad ORT Uruguay
Facultad de Ingeniería

InterfLan: Interface Language

Lenguaje de especificación de interfaces para ingreso y validación de datos

Entregado como requisito para la obtención del título de Licenciado en Ingeniería de Software

Emil Santurio - 180754
Joaquín Silveira - 187102

Tutor: Álvaro Tasistro

2017

Declaración de autoría

Nosotros, Emil Santurio y Joaquín Silveira, declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos el Proyecto;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Emil Santurio
6-2-2017

Joaquín Silveira
6-2-2017

"A mi madre, siempre apoyándome a alcanzar todas mis metas. A mi padre, va por vos y siempre te tendré presente a mi lado. A Tato, por estar siempre presente cuando necesitábamos una mano."

-Emil

"A nuestro tutor, Tato, por recibirnos hasta en los lugares más peculiares cada vez que necesitamos evacuar dudas. A mi familia y mi novia por entender el espacio y el tiempo necesario para poder trabajar. Y finalmente a mi compañero por acompañarme en este proceso durante los momentos buenos y malos. Gracias a todos."

-Joaquín

Abstract

InterfLan es un lenguaje de especificación de interfaces para entrada de datos en la forma de formularios. Es una tarea difícil la de traducir una especificación descrita en forma narrada a código que cumpla con todas las reglas especificadas, tanto es así que existen en otras áreas de la programación, lenguajes intermedios para poder realizar una especificación de alto nivel y luego ser analizada y construir la implementación en base a dicha especificación. Eso es lo que InterfLan intenta resolver. Este nivel de abstracción para la generación de interfaces es relevante para las personas que diseñan sistemas críticos y también para todos aquellos que realizan formularios en base a especificaciones narradas.

La solución consta de la gramática con las reglas para poder escribir en el lenguaje, un verificador de sintaxis, un verificador de tipos y junto con el último un validador que verifica que algunas de las propiedades necesarias para comprobar la correctitud del formulario. En el proceso de desarrollo investigamos los lenguajes que más se utilizan hoy en día para definir formularios y poder identificar que carencias existían en dicha área. La etapa de desarrollo estuvo guiada principalmente por el proceso de desarrollo de un compilador sugerido en el curso "Lenguajes de programación" de este título. Luego se estudiaron ejemplos realistas de formularios como el formulario de solicitud para una tarjeta de crédito y un formulario para la configuración de marcapasos extraído de la tesis de Sebastián Degrandi [1].

Concluímos que la implementación de nuestro lenguaje y el validador tienen muchas ventajas en cuanto a la usabilidad con respecto a las soluciones existentes, pero que queda mucho por hacer para poder demostrar la correctitud de los formularios. Además se puede usar el resultado de nuestra solución para poder convertir una especificación en de código InterfLan a otros lenguajes o *frameworks*.

Palabras clave

lenguaje de especificación de interfaces; lenguaje específico de dominio; compilación

Índice

1 Introducción

Es un problema conocido por la comunidad científica el hecho de que la especificación de datos de entrada y validaciones complejas de sistemas críticos suele ser una tarea difícil. Sin embargo, para tareas similares como la *especificación o modelado* de estructuras de datos, contamos con lenguajes que lo expresan, no sólo de manera correcta y analizable o demostrable, sino también de una manera que es simple de comunicar. Éstos son los atributos que han hecho que esos lenguajes de *especificación o modelado* se hayan convertido en casi estándares dentro de la comunidad de los diseñadores y desarrolladores. Algunos ejemplos de estos pueden ser UML, MER, MR y otros más que son ampliamente usados en varios ámbitos industriales y académicos. Existen lenguajes (especialmente algunos *frameworks* de desarrollo de aplicaciones web) como *KnockoutJS* y *AngularJS* que permiten especificar dependencias entre campos y restricciones de datos de muchas formas. Sin embargo, el nivel de abstracción de los mismos no permite hacer muchas inferencias o validaciones sencillas en cuanto a propiedades sobre las especificaciones.

1.1 Motivación

Inspirados en el trabajo de Degrandi [1], que analiza el problema de la especificación de interfaces complejas para sistemas como el de la programación de marcapasos cardíacos, nos dispusimos a generar una nueva solución que se adecue más a los estándares que hoy en día los programadores exigen de los lenguajes de especificación. Es interesante investigar la posibilidad de generar una nueva manera de expresar las complejas restricciones que un formulario de ingreso de datos puede llegar a tener, de manera más sencilla y sin ambigüedades. Nos pareció que a esta área relevante de la programación le faltaba algo importante. Además, este problema no sólo concierne a las personas que especifican formularios de programas críticos como el de los marcapasos, u otros sistemas en donde se pone en riesgo vidas humanas, sino a cualquier programador que tiene que generar ingreso de datos en una aplicación regular. La tarea de transformar el mensaje de un cliente o interesado

del español (lenguaje natural) a un lenguaje de programación no es sencillo y muchos son los problemas que la traducción trae. Por lo tanto creemos que, así como los demás lenguajes de especificación ayudan con esa transición de algo que puede entender un cliente a algo que la máquina puede utilizar, nuestra solución facilitaría el proceso de la especificación de formularios.

Entonces nos enfocaremos en la creación de un lenguaje de especificación con este propósito y de un nivel de abstracción mayor que el de los anteriormente mencionados. El mismo debe cumplir con poder expresar las mismas restricciones que los otros lenguajes de una manera más sencilla de manera que toda estructura expresada en el mismo sea validada. Como extensión del proyecto podríamos generar un compilador de nuestro lenguaje a *KnockoutJS*, *AngularJS* u otros *frameworks* tanto web como de aplicaciones de escritorio. Sin embargo, cabe destacar que el propósito de la tarea es la posibilidad de especificar las restricciones de una manera completa pero a la vez sencilla y fácilmente comunicable.

1.2 ¿Qué existe?

Por un lado tenemos los lenguajes de carácter general que permiten generar interfaces gráficas, tanto de escritorio como aplicaciones web. De éstos nos quisimos concentrar en los web, ya que son los más populares hoy en día, la gran mayoría del desarrollo se mueve en esa dirección. No por alguna característica en particular de éstos sino por el hecho de que son simples de integrar en arquitecturas web favoreciendo atributos de calidad tales como disponibilidad, accesibilidad y portabilidad. Pero no queremos quedarnos con los lenguajes de carácter general porque en ellos suele ser necesario tener que declarar estructuras complejas o propias de la práctica de la programación y mucho código procedural para poder darle el comportamiento deseado a un formulario. En cambio queremos quedarnos con los lenguajes y *frameworks* que nos permiten de manera natural, y mayormente declarativa, describir el comportamiento que el formulario tiene cuando recibe estímulos. Los que hemos encontrado relevantes y que cumplen con estas características son *UIML* (*User Interface Markup Languaje*), *AngularJS* y *KnockoutJS*. Todos permiten especificar la mayoría de las restricciones o el comportamiento que un formulario podría llegar a tener de manera declarativa y natural. También consideramos estudiar *RoR* (*Ruby On Rails*) y *Django*, ya que parecían prometedores, pero descubrimos que los formularios en esos *frameworks* naturalmente están atados a modelos de datos con las validaciones en ellos, y por lo tanto no eran declarativos. Uno para poder utilizarlos debería especificar,

en general, uno o varios objetos que representan la entrada de datos o caer en especificar el comportamiento del formulario de la misma manera que en un lenguaje de carácter general. También debemos hablar de la tesis de Degrandi, que propone una solución al mismo problema que estamos atacando pero pone en prioridad aspectos distintos que nosotros. Es cierto que su solución es declarativa, pero no tiene una forma natural de expresar las restricciones. De todos modos una comparación más extensa se presentará en la siguiente sección.

1.3 Comparación y Contribución

Para empezar a comparar las opciones que ya existen, necesitamos hablar de qué características son deseables en un lenguaje de este estilo, o por lo menos de aquellas que nosotros consideramos son las más valiosas. El lenguaje que necesitamos debe ser:

- Fácil de entender.
- Sencillo de escribir.
- Declarativo.
- Con suficiente potencial y flexibilidad como para poder expresar cualquier tipo de relación que tenga sentido entre campos.
- Se debe poder comprobar que las relaciones entre los campos tienen sentido (por lo menos en su gran mayoría).

Partiendo de esto podemos analizar qué ventajas y desventajas tienen los lenguajes y *frameworks* antes mencionados.

1.3.1 UIML

```
...
<behavior>
  <rule>
    <condition>
      ...
    </condition>
    <action>
      ...
    </action>
```

```
  </rule>
</behavior>
...
```

Este lenguaje [2] permite la especificación de interfaz de usuario mediante etiquetas. Pero a diferencia de HTML, no es un formulario con controles estáticos, sino que se puede definir comportamiento frente a estímulos dentro de la etiqueta `<behavior></behavior>`. El desarrollador define en una parte de la estructura los campos, los atributos que tiene, y luego el comportamiento que va a tener mediante reglas, cada una de las cuales es dada por una condición y una acción. Las acciones suelen cambiar alguna propiedad de uno o más campos, permitiendo así cambiar su valor, su visibilidad, etc. Al parecer todos los tipos de restricciones se pueden expresar en él, además es declarativo y es muy parecido a HTML, por lo que aprender a escribir en él no es una tarea compleja. Sin embargo tiene algunos problemas. Es complejo entender las relaciones que hay entre los campos ya que todas van en la sección de comportamiento y no junto a los campos en sí. Se necesita conocer muchas etiquetas para poder completar la estructura y además ésta es muy larga. En el fragmento de código de arriba se muestran solamente las etiquetas que rodean la sección de comportamiento, pero para especificar solamente la condición es necesario conocer al menos cinco etiquetas más que estarán anidadas dentro de la condición, sin contar las de la acción. Otro problema que presenta es que debido a que todo el comportamiento va en la misma sección, pero no con un orden en particular, puede ser que se redefina comportamiento o que hayan conflictos que no son simples de visualizar por la separación de los campos y el comportamiento. También está el hecho de que es necesario usar demasiados símbolos para poder declarar pocas cosas debido a la estructura de las etiquetas.

1.3.2 *AngularJS*

```
...
<table>
  <tr><th>Row number</th></tr>
  <tr ng-repeat="i in [0, 1, 2, 3, 4, 5, 6, 7]"><td>{{i}}</td></tr>
</table>
...
```

Este es un *framework* para *JavaScript* y HTML, que permite agregar controladores con características particulares como el *two-way-binding* y etiquetas con comportamiento en HTML. La idea de *AngularJS* [3] es mantener un modelo que esté en un controlador, asociado a una vista. Los valores de

la vista se ven afectados por los cambios en el modelo y viceversa (característica conocida como *two-way-binding*). Esto hace que definir comportamiento condicional en *AngularJS* sea muy simple y que los cambios en el formulario se visualicen al instante. Debido a que los valores están en sincronía en el modelo y en la vista, es muy simple manejar relaciones del lado del modelo. Entonces se puede realizar habilitaciones y calcular campos en base a otros valores. Además de eso, la librería básica de *AngularJS* provee muchas validaciones comunes a todos los formularios, por ejemplo de la presencia de valores en un campo, validaciones de *regex* conocidos, etc. Pero tiene algunas desventajas. Por ejemplo, las estructuras o modelos que se declaran en el controlador no suelen ser las más intuitivas y uno necesita de un poco de experiencia para darse cuenta de cuál es su funcionalidad o para idearlas. Otro inconveniente es que la estructura se compone de HTML con anotaciones y de uno o más archivos de *JavaScript*, pero las validaciones no se concentran en uno solo, sino que algunas de ellas van como anotaciones en el HTML y otras van implícitas en el código *JavaScript*. Y a pesar de tener la gran mayoría de las relaciones posibles entre valores, no encontramos una manera nativa del *framework* para poder expresar una relación bidireccional entre dos campos.

1.3.3 *KnockoutJS*

```
...
var ViewModel = function(first, last) {
    this.firstName = ko.observable(first);
    this.lastName = ko.observable(last);

    this.fullName = ko.pureComputed(function() {
        return this.firstName() + " " + this.lastName();
    }, this);
};
...
```

KnockoutJS [4], al igual que *AngularJS*, es un *framework* de *JavaScript* y HTML. No sólo eso, sino que tiene muchas similitudes en cuanto al manejo de los valores. Presenta también *data bindings*, como el *two-way-binding* de *AngularJS*, pero en *KnockoutJS* uno debe especificar qué valores tienen el *binding* y qué tipo de relación tienen. Por ejemplo, un valor puede tener *binding* con un campo y que su valor dependa del definido en el controlador y en la vista o puede ser que sea un valor computado en base al valor de otros campos (caso en el cual sería un computado). Esto tiene sus ventajas y desventajas, ya que el programador tiene que incorporar más palabras

para poder definir lo mismo que en el otro *framework* y es necesario entender qué significado tiene cada tipo de relación, pero tiene más claro cómo es la relación de ese valor con quienes lo observan. La otra ventaja que tiene *KnockoutJS* es la variedad de tipos de validaciones o *bindings* que pueden tener los campos en el HTML. Con todos los *bindings* que tiene *KnockoutJS* podemos expresar todos los tipos de relaciones entre campos que queríamos expresar. Pero al igual que *AngularJS* tiene el problema de que a veces es necesario hacer estructuras como modelos que no tienen una extrapolación directa con algún concepto del formulario, sino que son artilugios para poder generar el comportamiento deseado. A pesar de eso es uno de los que tienen menor curva de aprendizaje y es bastante intuitivo, pero no llega al nivel que deseamos. Una de las diferencias importantes que tiene *KnockoutJS* con respecto a Angular es que en *AngularJS* los cambios de *binding* se generan con cada evento disparado sobre un campo, pero en *KnockoutJS* sólo se disparan las validaciones y las relaciones cuando uno pierde el foco sobre un campo.

1.3.4 *Ruby On Rails - Django*

```
... (Ruby On Rails)
class Article < ApplicationRecord
  validates :title, presence: true,
                    length: { minimum: 5 }
end
...
... (Django)
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
...
```

Finalmente están los *frameworks RoR* [5] (*Ruby On Rails*) y *Django* [6] que se forman sobre *Ruby* y *Python* respectivamente, pero que comparten muchas características. Estos son *frameworks* diseñados para generar soluciones de manera muy sencilla y rápida. Construir aplicaciones web complejas es muy simple en estos lenguajes, pero por ello mismo no se concentran en la parte gráfica en cuanto al ingreso de datos, sino más bien en los modelos que representan las entidades de la aplicación. A pesar de que algunas de las validaciones que creemos necesarias parecen poder expresarse de forma sencilla, esto no es tan así porque la parte gráfica en estos lenguajes se genera en el servidor de manera estática y luego se envía al navegador la vista

del formulario. En ésta se suelen utilizan otros lenguajes o *frameworks* para hacer dinámica la interacción con el usuario, por lo que no podemos generar cambios en el formulario a medida se van ingresando datos. Sin embargo, queremos destacar que la sintaxis de ambos lenguajes es muy buena. Es simple de entender, clara, amigable para el desarrollador y para el cliente (ya que se puede leer de la misma forma que un texto en prosa) y utiliza pocos símbolos.

1.3.5 Degrandi

El lenguaje propuesto por Degrandi en su tesis tiene características muy positivas. Por un lado es un lenguaje con un poder expresivo muy grande debido a que todos los campos tienen un dominio y ése es el cual se puede calcular en base a otros, permitiendo así generar todas las validaciones posibles. También es interesante que todos los valores en los campos aparezcan como "valores seleccionables", ya que impide que el usuario ingrese cualquier valor en el input. Sin embargo tiene características que no nos parecieron positivas.

El código necesitaba de mucho conocimiento de *C* para poder generar los dominios y hasta a veces se debían generar de manera procedural en lugar de declarativa. Otro aspecto negativo es que los controles son solamente declarados en un sector y luego en otra parte se declaran las restricciones, lo cual no establece un vínculo claro entre el campo y todas sus validaciones o restricciones. Además las estructuras necesarias para poder acceder a los valores de otros campos o para declarar algunas expresiones son muy largas y aumentan el tamaño del código de manera innecesaria.

Además observamos que todas las decisiones de diseño que fueron tomadas en su lenguaje apuntaban únicamente a la especificación del formulario de los marcapasos y no pensó en un público más amplio.

En conclusión no encontramos un lenguaje o *framework* que se adecue completamente a lo que queremos, por lo que planeamos crear un lenguaje que combine el poder de expresividad de *KnockoutJS* y *AngularJS*, con la simpleza de sintaxis de *Ruby* y *Python*, más los otros atributos mencionados anteriormente.

2 Descripción del lenguaje

Para poder analizar la estructura del lenguaje y para que el lector comprenda el uso del mismo decidimos documentar el proceso de especificación de los siguientes formularios, los cuales no contienen todas las cosas que se pueden especificar en el lenguaje, pero sí la gran mayoría de ellas.

2.1 Primer Ejemplo - Línea de crédito

Este formulario (**Figura 2.1**), como sus campos delatan, es para una solicitud de una línea de crédito personal. Consta de algunos campos sencillos, como el nombre y dirección, y de otros que no lo son tanto, como la fecha de nacimiento, la fecha de casamiento, la ciudad y el país de nacimiento y el ingreso total para la solicitud del monto de la línea de crédito. Es importante que expliquemos los campos complejos antes de empezar con la especificación, ya que ése sería el proceso natural por el cual la especificación pasaría. Una redacción de las restricciones que debe poseer el formulario anterior podría ser la siguiente:

- Ninguno de los campos del formulario puede ser vacío, a excepción de la fecha de matrimonio o concubinato de no ser alguno de los recién mencionados la situación actual del solicitante.
- El correo electrónico debe ser un correo electrónico sintácticamente bien formado.
- La cédula de identidad debe pasar por la función que verifica el último dígito de la misma.
- La fecha de nacimiento debe corresponder con la de una persona mayor de edad (o sea mayor de 18 años).
- La edad no se podrá ingresar manualmente, sino que será calculada en el momento en el que se ingrese la fecha de nacimiento.

Figura 2.1: Formulario de ejemplo.

- En caso de que el estado civil del solicitante sea"Concubino" o "Casado", se deberá habilitar el campo de la fecha de matrimonio o concubinato, de otra forma el campo estará deshabilitado.
- Una vez que se seleccione el país de nacimiento, se mostrarán las ciudades correspondientes a dicho país en la opción de "Seleccionar Ciudad".
- Tanto si se ingresa el límite de crédito o los ingresos nominales, será calculado el otro valor. En caso de ingresarse los ingresos nominales, el máximo de la línea de crédito será el doble de dicho valor. En cambio si se ingresa el valor máximo de la línea de crédito deseada, se calculará los ingresos nominales mínimos necesarios para obtener el crédito, como la mitad de la línea de crédito.

Ahora, teniendo la especificación en español, podemos pasar a crear el formulario. Debemos definir la estructura que contiene los campos para la entrada de datos. Podríamos ponerle como nombre `linea_de_credito`.

```
form linea_de_credito let

end
```

Código 2.1: Definición del formulario.

Sin embargo, un formulario sin campos no tiene sentido, así que agreguemos un control para cada uno de los campos del formulario. La declaración de los campos es similar a la del formulario, sólo que se deben especificar algunos datos más. Por ejemplo, lo siguiente constituiría una declaración de un control para ingresar el nombre del solicitante.

```
input "Nombre" as name:string let

end
```

Código 2.2: Definición de un control.

La definición de un control está compuesta por:

- **Tipo de Control:** Aquí se seleccionará la "forma" del control según su interacción con el usuario. Se puede seleccionar entre: `input` para definir un campo de ingreso de datos, `label` para definir un campo que solo muestra determinado dato sin que el usuario pueda editarlo, y `select` para definir un campo donde el usuario podrá seleccionar únicamente un valor del dominio; a su vez este tipo de control tiene dos variantes según como se desee mostrar: puede ser tanto como una lista desplegable de opciones (`select combo`), o exponiendo todas las opciones como un grupo de *radio buttons* (`select radio`).

- **Nombre:** Se definirá el nombre del control para constituirlo como variable dentro del formulario y opcionalmente se podrá especificar un nombre distinto para la vista al usuario. Esto último se lo conoce como *caption* (`"Texto a mostrar" as variable`) el cual se puede observar su uso en el **Código 2.2**.

- **Tipo de dato:** Todo control deberá tener acompañado de su nombre el tipo de datos que contendrá. Este puede ser uno de los primitivos (`integer`, `bool`, etc.) o un enumerado (que veremos más adelante).

- **Sentencias:** Por último tendremos las sentencias que especificarán qué restricciones, validaciones y dependencias tendrán, las cuales iremos agregando a continuación.

Continuando con la definición de todos los campos, tendríamos lo siguiente:

```
form linea_de_credito let
    input "Nombre" as nombre:string let
    end
```

```
    input "Apellido" as apellido:string let
    end

    input "Dirección de correo  electr ónico" as email:string let
    end

    input "Cédula de identidad" as cedula:integer let
    end

    input "Fecha de Nacimiento" as f_nacimiento:datetime let
    end

    label "Edad" as edad:integer let
    end

    select combo "Estado civil" as ecivil:<?> let
    end

    input "Fecha concubinato o matrimonio" as f_mat:datetime let
    end

    select combo "País de nacimiento" as p_nacimiento:<?> let
    end

    select combo "Ciudad de nacimiento" as c_nacimiento:<?> let
    end

    input "Mínimo ingreso nominal" as ingresos:integer let
    end

    input "Máximo línea de crédito" as credito:integer let
    end
end
```

Código 2.3: Definición de todos los controles.

Nótese que la edad no se declaró como un `input` sino como un `label` para cumplir con la especificación de que no debe ser un campo que se pueda modificar manualmente. También es importante ver que no hemos podido declarar el tipo de los controles `select` ya que su tipo no se corresponde con ninguno de los primitivos. Por lo cual debemos declarar los tipos para los `select`. Para el estado civil y los paises simplemente necesitamos un enumerado con los valores válidos de la siguiente manera:

```
enum estado_civil [("Casado/a",CAS),("Soltero/a",SOL),
                   ("Concubino/a",CON),("Viudo/a",VIU),
                   ("Divorciado/a", DIV)] ;

enum paises [("Uruguay",UY), ("Brasil", BR), ("Argentina", AR)] ;
```

Código 2.4: Definición de enumerados.

Los "valores" de los enumerados están compuestos por un par ordenado (*caption*, valor), para poder diferenciar la referencia interna en el código y lo que se muestra al usuario.

Para las ciudades podríamos hacer lo mismo, sólo que tendríamos el inconveniente de que no importa que país se elija, aparecerán todas las ciudades. Así que debemos crear un enumerado que dependa de el país que se ha seleccionado.

```
enum ciudades let
    domain (case p_nacimiento of
                paises.UY -> [("Montevideo", MVD), ("Rocha", ROC),
                              ("Canelones", CAN)] ;
                paises.BR -> [("Brasilia", BRS),("Fortaleza", FRT),
                              ("Recife",RCF)] ;
                paises.AR -> [("Buenos Aires", BAS), ("Santa Fe",STF),
                              ("Córdoba", CBA)]
            end) ;
end
```

Código 2.5: Definición de enumerado con dependencia

Analizando la declaración anterior, podemos observar la primera sentencia dentro de un bloque en nuestro lenguaje. Esa sentencia define que el dominio del tipo enumerado ciudades se define por casos en base al valor del control `p_nacimiento`. Luego se define qué es lo que pasa en cada uno de los casos posibles del `select p_nacimiento`. Cada una de las opciones se llama **rama** y dan un nuevo enumerado para cada caso posible, definiendo así los valores que va a tomar el enumerado en cada instante. Repasando un poco podemos llegar a que la sección de los `select` se define así:

```
...
select combo "Estado civil" as ecivil:estado_civil let
end
...
select combo "País de nacimiento" as p_nacimiento:paises let
end
```

```
select combo "Ciudad de nacimiento" as c_nacimiento:ciudades let
end

enum estado_civil [("Casado/a",CAS),("Soltero/a",SOL),
              ("Concubino/a",CON),("Viudo/a",VIU),
              ("Divorciado/a", DIV)] ;

enum paises [("Uruguay",UY), ("Brasil", BR), ("Argentina", AR)] ;

enum ciudades let
   domain (case p_nacimiento of
              paises.UY -> [("Montevideo", MVD), ("Rocha", ROC)
                  ,
                       ("Canelones", CAN)] ;
              paises.BR -> [("Brasilia", BRS),("Fortaleza", FRT),
                       ("Recife",RCF)] ;
              paises.AR -> [("Buenos Aires", BAS), ("Santa Fe",
                  STF),
                       ("Córdoba", CBA)]
          end) ;
end
...
```

Código 2.6: Definición de los select y enums

Lo que resta es generar las validaciones para los demás controles. Para poder controlar que ninguno de los campos se encuentre vacío luego de ingresar los datos, debemos agregar la siguiente sentencia en todos los campos:

```
input "Nombre" as nombre:string let
   not_empty;
end
```

Código 2.7: Utilización de sentencia not_empty.

Para cumplir con la restricción de la validación del correo electrónico, introducimos la sentencia `regex`, de la siguiente manera:

```
input "Dirección de correo  electr ónico" as email string let
   not_empty ;
   regex "(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+
      $)" ;
end
```

Código 2.8: Utilización de la sentencia regex.

Para validar la fecha de nacimiento podemos hacer uso de un rango que deje claro que la fecha es válida y el solicitante es mayor de 18.

```
input "Fecha de Nacimiento" as f_necimiento:datetime let
    not_empty ;
    range (subs_years(now, 150))..(subs_years(now, 18)) ;
end
```

Código 2.9: Utilizacón de la sentencia range.

De esta manera nos aseguramos que la persona no sea mayor que 150 años ni menor de 18. Las operaciones de fechas se realizan de la siguiente manera: `subs_` o `add_`, la unidad sobre la que se va a operar (*years*, *months*, *days*), la fecha base de la operación y la cantidad que se va a aumentar o disminuir.

Continuando con las restricciones, debemos introducir las funciones para poder otorgarle a quien realice la especificación el potencial de especificar cosas que se encuentran por fuera de nuestro lenguaje, permitimos realizar funciones con cuerpo escrito en *JavaScript*, que es un lenguaje popular y flexible. La estructura de las funciones es la siguiente:

```
function verificar_cedula:bool (cedula:integer) let
    "cuerpo de la función en JavaScript"
end
```

Código 2.10: Definición de una función.

El nombre de la función está acompañado luego de los ":" por el tipo que retorna la función, luego continúa la lista de parámetros, si los tiene, que recibe la función con sus correspondientes tipos y finalmente el cuerpo que contiene el código *JavaScript* entre comillas. Ahora podemos realizar las validaciones del cálculo de la edad y la verificación de la cédula.

```
function verificar_cedula:bool (cedula:integer) let
    "cuerpo de la función en JavaScript"
end
input "Cédula de identidad" as cedula:integer let
    not_empty ;
    check (verificar_cedula(self)) ;
end
...
function calcular_edad:integer (fnac:datetime) let
    "cuerpo de la función en JavaScript"
end
```

```
label "Edad" as edad:integer let
    calculate (calcular_edad(f_nacimiento)) ;
end
```

Código 2.11: Definición de funciones de cédula y edad.

Veamos qué cosas se han agregado a la sintaxis. Además de las funciones que verifican la cédula y que calculan la edad del solicitante, necesitamos declarar la validación de la cédula en el cuerpo haciendo uso de la sentencia `check` que contine una expresión booleana. También debemos declarar que el valor del `label edad` se calcula mediante la expresión que le sigue. Es importante destacar también la expresión *self*, la cual hace referencia al valor del campo en el cual la expresión se encuentra. Continuando con las validaciones para la fecha de matrimonio o concubinato, debemos declarar que se debe habilitar solamante cuando el estado civil del solicitante sea concubino o casado. Y sería conveniente también que esta fecha sea coherente con la fecha de nacimiento.

```
input "Fecha concubinato o matrimonio" as f_mat:datetime let
    enable (ecivil in [CAS,CON]) ;
    range (add_years f_nacimiento 14)..(now) ;
end
```

Código 2.12: Utilización de la sentencia enable.

Finalmente lo que resta es la relación entre la línea de crédito y los ingresos. Debido a que la relación es bidireccional, se debe establecer una manera de calcular el ingreso mínimo en base a la línea de crédito y otra forma de calcular la línea de crédito en base al ingreso.

```
input "Mínimo ingreso nominal" as ingresos:integer let
    not_empty ;
    equity credito (self * 2) (credito / 2) ;
end

input "Máximo línea de crédito" as credito:integer let
    not_empty ;
end
```

Código 2.13: Utilización de la sentencia equity.

Se deja en el **anexo C.1** el formulario completo al finalizar este ejemplo.

2.2 Segundo Ejemplo - Problemas con dependencias

El siguiente ejemplo refiere a errores comunes que podrían suceder relacionados a las dependencias entre campos.

Supongamos que deseamos declarar un formulario con tres campos de tipo *integer* con los nombres A, B y C.

```
form dependencias let
    input A:integer let
    end

    input B:integer let
    end

    input C:integer let
    end
end
```

Estos campos actualmente no tienen ninguna restricción ni relación entre ellos. Sin embargo, deseamos agregar una relación entre los campos A y B. En esta nueva relación declaramos que A es el doble de B, y por lo tanto B es la mitad de A. En una primera instancia intentaremos declarar la relación como dos `calculate`, porque tiene sentido decir que A se calcula en base a B y viceversa.

```
    ...
    input A:integer let
        calculate (B * 2) ;
    end

    input B:integer let
        calculate (A / 2) ;
    end
    ...
```

Esto produce un ciclo ente las relaciones de `calculate`, la cual podría generar inconsistencias. Esto es porque si no fueran inversas una de la otra, al ingresar el valor en uno de los campos se generaría una secuencia infinita de cálculos para determinar el valor de ambos campos. Por lo tanto el programa nos indica el error con el siguiente mensaje:

"There is an error in the Calculate dependency # there is a cycle among ("A" "B")."

Para corregir el error anterior basata con sustituir las dos relaciones de `calculate` por una de `equity`. Esta nueva dependencia que se presenta, establece una relación bidireccional de actualización. Es conveniente contar con ella para que el desarrollador sea consciente que quiere establecer esta relación y no haya sido por error que el haber colocado dos relaciones `calculate` en un ciclo. Esta relación se puede declarar en cualquiera de los los campos, pero vamos a declarar en A por ahora.

```
...
input A:integer let
    equity B (self / 2) (B * 2) ;
end

input B:integer let
end
...
```

Ahora supongamos que deseamos poner una relación de `calculate` entre B y C. Pero en lugar de hacerla de C a B lo hacemos en el sentido contrario.

```
...
input B:integer let
    calculate (C * 2) ;
end

input C:integer let
end
...
```

Esto también genera conflictos, ya que con el nuevo cambio estamos diciendo que B se calcula en base a la fórmula $C*2$, pero el valor de B ya estaba determinado como $A/2$, porque aunque B no tenga la sentencia `equity` en su cuerpo, de todos modos forma parte de la relación. Eso significa que B tiene 2 valores lo cual no es consistente. En esta situación el mensaje de error sería:

"There cannot be an Equity and Calculate Dependency in the Input B."

Cuando ponemos la sentencia en el control correcto el error desaparece. Pero para analizar otro caso de error más intentemos poner una relación de `equity` entre C y B.

```
...
input B:integer let
end
```

```
input C:integer let
    equity B (self * 23) (B * 23) ;
end
...
```

El error en este caso sería que B se encuentra en más de una relación `equity`, porque aunque no tenga ninguna de las sentencias en su cuerpo, está unida a A y a C por una relación de este tipo.

Con estos ejemplos cubrimos muchos de los escenarios que podrían generar problemas. Otro escenario interesante es discutido en la **sección 3.2.4**.

3 Proceso de Desarrollo

Haremos un repaso por las tres secciones que componen el compilador de InterfLan; dos de ellas han sido implementadas correctamente mientras que discutiremos la justificación de por qué no se ha logrado la implementación de la generación de código.

Para la implementación de todo el lenguaje nos hemos basado en su construcción de la misma manera que lo hemos realizado en el curso "Lenguajes de Programación" de este título junto al libro *"Implementing Programming Languages"* [7] donde se proveen herramientas y fundamentos para el diseño e implementación de un lenguaje.

3.1 Gramática

Diseñar un lenguaje desde cero no es una tarea trivial, más aún cuando se trata de establecer una forma de escribir código lo más natural posible para expresar conceptos complejos, como puede ser una dependencia entre los valores de dos o más campos. Para esto la idea principal de este trabajo es que el código en InterfLan no esté tan lejano a lo que es la especificación en lenguaje natural.

El desafío fue no utilizar símbolos en la sintaxis del lenguaje, o ahorrarse lo más que se pueda. Para esto nos basamos ampliamente en la sintaxis que utiliza *Ruby* la cual presenta una forma de leer código muy similar a la de leer un texto en prosa. De allí, por ejemplo obtuvimos la inspiración para la declaración de bloques, funciones de nuestro lenguaje con su nombre sin abreviaturas, acrónimos, etc. Tal es el caso de `not_empty;` para verificar que un campo no sea vacío, o las operaciones sobre las fechas (`datetime`).

Para la definición de la gramática utilizamos la notación *BNF* por lo cual los ejemplos que brindaremos a continuación serán expresados en esta notación. Se deja también a disposición del lector la gramática entera en

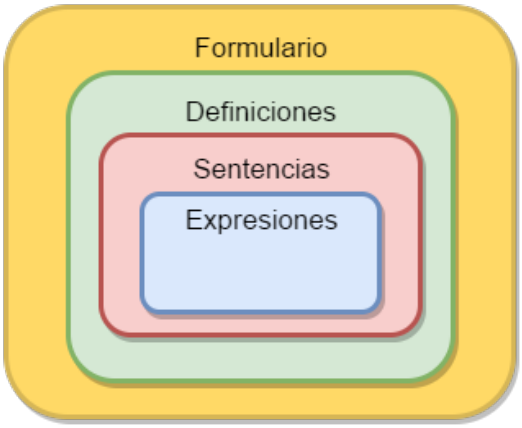

Figura 3.1: Estructura de la gramática de InterfLan.

formato natural en el **anexo A**.

Como podemos ver en la **Figura 3.1** la gramática del lenguaje está dividida en cuatro niveles:

- **Formulario:** Es la definición de todo el "programa" en sí. No se podría definir un campo si no está dentro de un formulario. Contiene todas las definiciones que serán usadas en el mismo como son los controles, la definición de nuevas funciones, y declaración de tipos enumerados. El alcance del formulario está delimitado por el bloque `form`.

- **Definiciones:** Dentro del formulario encontramos las definiciones de los controles, funciones y enumerados. Los controles serán aquellas definiciones que se verán reflejados en el formulario como los campos donde se ingresarán los datos (`input` y `select`) o donde se mostrarán datos para el caso de las etiquetas (`label`). La definición de funciones tiene como objetivo que el desarrollador pueda programar sus propias verificaciones, cálculos, y transformaciones.Todas las definiciones estarán acompañadas del tipo excepto los enumerados que constituyen un tipo nuevo.

- **Sentencias:** Éstas constituyen las restricciones que tendrán los controles definidos en el formulario. Cada control podrá contener desde ninguna sentencia que marque una restricción hasta la cantidad que se desee, de esta manera se pueden expresar de varias formas según la cantidad que el desarrollador quisiera imponer: bien podría no incluir ninguna sentencia por lo cual la definición del control concluiría luego del tipo, podría incluir solo una sin necesidad de abrir un bloque, o puede expresar una o más dentro del bloque `let...end`.

```
input nombre:string ;
```

Código 3.1: Control sin sentencias.

```
input nombre:string not_empty ;
```

Código 3.2: Control con una sentencia.

```
input "Fecha de Nacimiento" as f_nac:datetime let
   not_empty ;
   range (subs_years(now, 150))..now ;
end
```

Código 3.3: Control con más de una sentencia.

- **Expresiones:** Son los elementos más atómicos del lenguaje, ellas son las que forman los valores ya sea desde un literal como puede ser un número entero (124), una variable (`f_nac`), una operación aritmetica (2+2.1), etcétera. Es interesante mencionar además que las expresiones tienen un nivel de precedencia que indica qué expresiones se tienen que evaluar antes que otras; esto ayuda a que la sintáxis no se vuelva tan engorrosa por el uso de tanto paréntesis para delimitar qué se debe evaluar primero. A modo de ejemplo, las expresiones literales como los números enteros, *strings*, fechas, etc. tienen el nivel más alto ya que no necesitan evaluarse. También se puede brindar el ejemplo de la suma de productos donde el producto tiene mayor precedencia (se evalúa primero) que la suma. En la definición de la gramática podemos observar cómo las expresiones están definidas en este orden y que cada nivel de precedencia está denotado con un número.

```
ETrue.     Exp17    ::= "true" ;
EFalse.    Exp17    ::= "false" ;
EInt.      Exp17    ::= Integer ;
.
.
.
ETimes.    Exp14    ::= Exp14 "*" Exp15 ;
EDiv.      Exp14    ::= Exp14 "/" Exp15 ;
EMod.      Exp14    ::= Exp14 "%" Exp15 ;
EPlus.     Exp13    ::= Exp13 "+" Exp14 ;
EMinus.    Exp13    ::= Exp13 "-" Exp14 ;
```

Código 3.4: Expresiones con precedencia. Notación BNFC.

Luego de especificada la gramática, utilizaremos la herramienta *BNFC* [8] (*BNF Converter*) la cual, a partir de la gramática escrita en *BNF*, produce *Lexer*, *Parser* y la sintaxis abstracta en *Haskell*. De aquí en adelante y debido a que tenemos la sintaxis abstracta en este lenguaje, hemos decidido seguir utilizándo este último lenguaje para el resto del compilador.

3.2 *Type Checker*

Luego de la generación de la gramática debíamos realizar validaciones en cuanto al tipo de las expresiones y la corrección de la estructura del formulario. Esto se debe a que luego de haber pasado por el *lexer* y *parser*, de todos modos podían haber varios errores. Algunos de bajo nivel de abstracción, como puede ser una expresión que debe ser booleana y no lo es, y otros de alto nivel como una circularidad en dependencias entre controles. De la misma manera que procedimos en el curso de Lenguajes de Programación, nos dispusimos a analizar qué cosas necesitábamos para poder verificar los tipos de las expresiones. Para poder hacerlo necesitábamos de contextos con cierta información para verificar que cuando se hiciera referencia a funciones o al valor de otros controles los tipos sean los correctos. Concluimos que necesitaríamos por lo menos dos "pasadas" por el código, una para poder generar un contexto con la información de los tipos de las funciones, controles y tipos enumerados, y otra para ver que los tipos en todas las sentencias y expresiones fueran correctos.

3.2.1 Generación de contextos

El contexto necesario para poder verificar las expresiones y sentencias necesitaba de la siguiente información:

- **Contexto de controles:** Esta es una tabla con el nombre de los controles que se encuentran en el formulario y su tipo
- **Contexto de funciones:** Este es una tabla del nombre de las funciones con su firma (lista ordenada con los tipos de sus parámetros y de su resultado)
- **Contexto de enumerados:** Este es una tabla con los nombres de los enumerados y los elementos que dicho tipo contiene

Para poder generar estos contextos era necesario recorrer el código de manera superficial, sólo hasta el nivel de las definiciones, ya que toda la información necesaria para hacerlo estaba en ellas. A medida que íbamos

recorriendo el árbol de sintaxis abstracta, cuando nos encontramos con un control, lo agregamos al contexto correspondiente y de la misma manera hicimos con las funciones y los enumerados. Con la generación de estos contextos ya podíamos identificar algunos errores de bajo nivel, como el hecho de que no hubieran controles, funciones o enumerados con el mismo nombre. Estos errores son notificados antes de siquiera analizar las sentencias. Otra cosa para la cual necesitábamos hacer una pasada por el código era para ver cómo se formaban las relaciones entre los controles, o mejor dicho las dependencias que se generaban entre ellos. Por lo tanto agregamos antes de verificar los tipos de las sentencias una pasada para generar otro contexto de dependencias.

3.2.2 Generación del contexto de dependencias

Debido a que nuestro lenguaje permite dependencias entre los valores de los controles necesitábamos de alguna manera verificar que no hubieran circularidades entre dichas dependencias, ya que las circularidades podrían provocar que los controles no se pudieran calcular, no pudieran ser correctos bajo ninguna entrada o que no se pudieran habilitar. La manera que elegimos para poder identificar las circularidades es mediante la formación de grafos que representaran las dependencias que se generaban entre los mismos. Sin embargo, no todas las dependencias tenían el mismo significado para nosotros ni provocaban los mismos problemas. Por lo tanto generamos un grafo para cada sentencia que pudiera generar una dependencia:

- `calculate`
- `equity`
- `range`
- `enable`
- `check`

Primero en cada uno de los cinco grafos agregamos la lista de los controles como vértices (conveniente era el hecho de que ya teníamos la lista entera de los controles obtenida en la pasada anterior). Recorriendo las sentencias de un control cuando nos encontrábamos con una de esas sentencias, recorríamos las expresiones que las formaban buscando alguna referencia a un control. Cuando encontrábamos una referencia agregábamos una arista al grafo de dicha sentencia entre el control que estabamos recorriendo y el

control mencionado en la referencia.

Sin embargo, todos los grafos no son así de simples. El generado por la relación de `equity` necesita de los de `calculate`, ya que no sólo se pueden generar cíclos dentro de las relaciones de `equity`, sino que entre ése y los `calculate` también pueden haber problemas, por lo que en el grafo de la sentencia `equity` agregamos las relaciones de `calculate`. Una vez que tenemos todos los contextos de tipos y el contexto de dependencias podemos pasar al a última pasada para chequear los tipos.

3.2.3 Verificación de tipos en sentencias y expresiones.

El proceso para poder verificar los tipos fue el mismo que se utilizó en el curso de "Lenguajes de Programación". Con la información de los contextos generados anteriormente, recorrimos el árbol de sintaxis abstracta producida por el *lexer* y *parser*, agregando al mismo etiquetas de tipo a las expresiones y controlando que los tipos de las expresiones siguieran la estructura que nosotros habíamos ideado. Las etiquetas de tipo son útiles para una posterior etapa de generación de código. El código de esta etapa fue escrito en *Haskell*, ya que con el mismo lenguaje habíamos construido un verificador similar en el curso de Lenguajes. A continuación describiremos de manera abreviada el procedimiento del verificador, pero una lista de todas las reglas utilizadas para escribir posteriormente el código se encuentra en el **anexo B**.

Analizándolo con más detalle, partimos de un formulario que debe ser validado y un contexto con las características mencionadas en las secciones precedentes. Para validar el formulario necesitamos recorrer las definiciones que lo componen. Las definiciones pueden ser funciones, enumerados o controles (`select`, `input` o `label`). Las funciones no tienen validaciones de tipo ya que su cuerpo no está expresado en código InterfLan, sino en *JavaScript*. Por lo tanto tenemos una ventaja muy grande con esta restricción, ya que no se puede generar recursividad ni dependencias de ningún tipo entre funciones. El cuerpo de cada función es independiente de las demás ya que solo se puede hacer uso de los parámetros declarados en la firma de la misma[1]. Los enumerados tienen la única validación de que si se declaran por extensión (con una lista de los valores posibles), solo debíamos verificar que no hubieran valores repetidos. En cambio cuando estaban declarados con una sentencia, debíamos verificar que esa sentencia fuera una dependencia `domain`, con una

[1]Esto es una característica muy importante porque nos ahorró muchas validaciones, de las cuales algunas serían imposibles.

expresión de `case` dentro y que el tipo del `case` fuera un enumerado.

Para los controles, las validaciones eran más complejas. Antes de verificar los tipos de las sentencias, en los `select` debíamos controlar que no hubiera ninguna sentencia de `equity` ni `calculate` y que el tipo del control sea un enumerado. Para los `label`, se debía verificar que no hubieran sentencias `equity`, ya que no tiene sentido que un `label` los tenga porque no se puede alterar el valor del mismo manualmente. Finalmente para los `input` no debe haber más de un `calculate`, no más de un `equity` y si había alguno de los dos, no podría estar el otro, ya que no es correcto que un `calculate` y un `equity` convivan en el mismo control. Luego de dichas verificaciones es necesario recorrer las sentencias de los controles y ver que todas sean válidas. Pero con los contextos generados en la etapa anterior no basta, ya que en las expresiones se puede hacer mención del valor del control mediante la palabra reservada `self`, así que agregamos a los contextos (antes de ingresar a un control) el tipo que tiene `self` en esa sección de código.

Algunas de las sentencias no tienen validaciones complejas así como `not_empty` (la cual no tiene ninguna) y `regex` (que solo debe validar que el tipo de `self` sea `string`). Las validaciones para las otras sentencias son las siguientes:

- `check` y `enable`:
  - La expresión que las compone debe ser de tipo `bool`.
- `range`:
  - el tipo de ambas expresiones (el extremo inferior y el extremo superior) son del mismo tipo que el `self` y no son `enumerate`.
- `calculate`:
  - El tipo de la expresión que lo compone debe ser igual al del `self`
- `equity`:
  - El tipo de las expresiones que la componen deben ser del mismo tipo que `self`.
  - El control asociado en la sentencia debe existir y tener el mismo tipo que `self`.

Para culminar, el proceso para las expresiones es similar, sólo que, cuando inferimos el tipo de una expresión, agregamos una estructura al árbol con el

tipo de la expresión y la expresión en sí. Las únicas reglas interesantes para la inferencia de tipo son las operaciones aritméticas que están sobrecargadas para los `integer` y los `float`, y la suma que además de estar sobrecargada para los tipos anteriores, también lo está para los `strings` (permitiendo así expresar la concatenación de `string`).

3.2.4 Validación de los grafos de Dependencia

En la etapa final de la verificación del formulario, buscamos errores en cuanto a dependencias en los grafos generados en la segunda "pasada". Una de las tareas más díficiles fue determinar cuáles eran las condiciones que debían tener los grafos que habíamos generado para que fueran correctos o mejor dicho, qué condiciones no podían tener. Entonces empezamos a determinar en qué situaciones un formulario era correcto y cuándo no. En cuanto a las relaciones simples como `enable`, `check`, y `range`, era simple ver que lo único que debíamos evitar eran los ciclos, pero en las relaciones de `calculate` y de `equity`, los escenarios debían ser más complejos. Para las dependencias `calculate`, al igual que para las sentencias anteriores, no se podían producir ciclos en el grafo, pero un análisis más complejo era necesario cuando existieran relaciones de equidad junto con las de cálculo. Entonces decidimos ver qué características tenía que cumplir el grafo de `equity`. Para empezar, era obvio que los ciclos no eran un indicativo de que algo anduviera mal porque de hecho bastaba con que hubiera una única relación de `equity` para que hubiera un ciclo. Además, con las aristas generadas en las relaciones de `equity` no bastaba a este grafo necesitamos agregar las aristas del grafo de las dependencias `calculate` porque los problemas surgían de las relaciones entre ellos y no sólo de las aristas de `equity`. Luego de varios ejemplos correctos y erróneos, determinamos que la manera de verificar si este grafo era correcto tenía que ver con los grados de entrada y de salida de los vértices. En una relación de equidad (**Figura 3.2**) un nodo tiene una relación o arista que sale de él en dirección al otro vértice de la relación y uno que entra desde el otro vértice en el primero, pero pueden producirse más dependencias también. Por ejemplo si un vértice o valor A se calcula en base a un vértice B y un valor externo C, y luego el vértice B se calcula en base al A y el C también se generan las aristas que se ven en la **Figura 3.3**, ya que tanto A como B dependen del valor de C y esto podría ser un escenario válido.

Pero si el valor de C dependiera de alguno de los campos anteriores (como en la **Figura 3.3**), el formulario podría tener inconsistencias, ya que se genera un comportamiento circular no sólo en la relación de `equity`, sino junto con una de `calculate`. Lo particular de dicho grafo es que uno de los vértices

Figura 3.2: Grafo relación de `equity`

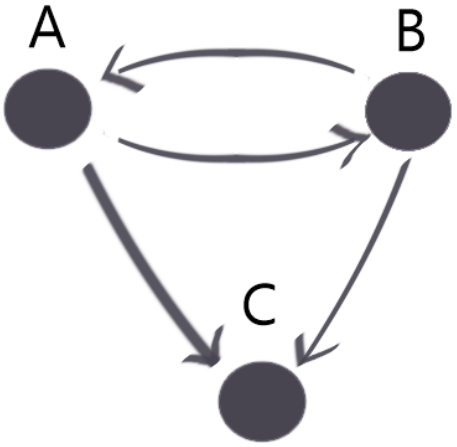

Figura 3.3: Grafo relación de `equity` con dato externo

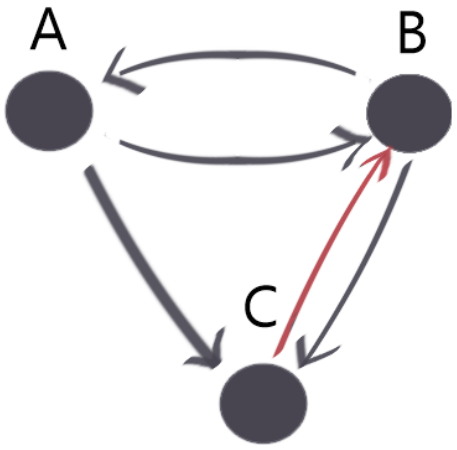

Figura 3.4: Grafo incorrecto relación de `equity` y `calculate`

(en ese caso B) tiene dos aristas que salen y dos que ingresan en él. Experimentando con otros escenarios similares llegamos a la conclusión de que ésa es la condición que determina que el grafo es incorrecto. Así que nuestra tarea consistió en identificar si en el grafo de `equity` y de `calculate` había algún vértice con grado de entrada y grado de salida mayor o igual a 2.

3.3 Generación de Código

La última etapa de este proceso es la generación de código que refleje la especificación escrita en código InterfLan. El hecho de que se valide la estructura, los tipos y las dependencias en el formulario es lo que importa más, pero sería conveniente que luego de ese proceso quien haga la especificación tenga una pieza de código con la cual empezar a trabajar como base. La etapa de generación constaría de tomar el árbol que devuelve el *type checker* con el formulario completo y los tipos de las expresiones para poder generar un formulario en alguno de los lenguajes y *frameworks* que estuvimos manejando como *KnockoutJS*, *AngularJS* o simplemente *JavaScript*. La traducción de esta estructura al código no es trivial, pero tampoco presenta mucha complejidad debido a que todo es declarativo excepto las funciones, las cuales ya deben estar expresadas en *JavaScript* (lenguaje admitido por todos los *frameworks* antes mencionados). Además, no sólo se podría generar código a uno de ellos sino a varios. La idea es que nuestra especificación se pueda usar como base para la posterior generación de código y no para que se acople a ningún *framework* en particular.

No planeamos hacer la generación de código en esta instancia ya que el tiempo de este trabajo no fue suficiente para alcanzar esta etapa. Sin embargo, si lo fuéramos a hacer, probablemente la traducción más directa sería a *KnockoutJS* ya que los tipos de relaciones que el *framework* permite son similares a las de nuestro lenguaje.

4 Resultado de Experimentación

En el trabajo de Degrandi [1] se plantea el lenguaje a raíz del problema de la programación de marcapasos cardíacos donde el riesgo a errar en la entrada de un dato es tan alto como el de perder una vida. Desde aquí es donde el caso de experimentación de este autor fue la especificación de la interfaz de programación del marcapasos para que se pudieran detectar errores de programación antes de grabar los cambios en el dispositivo del paciente.

A raíz de todo esto, nosotros hemos decidido que sería conveniente comparar nuestro trabajo con el de Degrandi ya que los demás lenguajes que hemos investigado con los cuales se pueden lograr prácticamente los mismos objetivos no tienen el mismo poder de expresividad como lo tiene éste. Para lograr esto implementaremos la misma interfaz que formuló Degrandi con InterfLan y a continuación compararemos si resulta más sencillo implementar este tipo de interfaces en nuestro lenguaje.

4.1 Interfaz de programación de marcapasos

Para poner en contexto al lector sobre el universo de datos que posee esta interfaz, citaremos las partes clave de la especificación tomadas del trabajo de Degrandi.

En la **Tabla 4.1** se puede observar la especificación de los datos de la interfaz de programación de un marcapasos bicameral.

Además, de los parámetros numéricos se menciona una serie de datos adicionales que el formulario deberá verificar dependiendo del ingreso de ciertos datos.

Parámetro	**Rango de Valores**
Modos	DDD, VDD, VVI, VVT, VOO, OVO, AAI, AAT, AOO, OAO, OOO
Frecuencia	32 a 120 ppm (46 valores)
Frecuencia Máxima	60 a 150 ppm (46 valores)
AV post estímulo	63 a 281 ms (15 valores)
AV post sensado	53 a 272 ms (15 valores)
Blanking	21 a 68 ms (4 valores)
Amplitud de pulso	2.5 V y 5.0 V
Ancho de pulso	122 a 1464 μs (12 valores)
Refractario	125 a 430 ms (16 valores)
Sensibilidad Auricular	0.5 a 4.0 mV (8 valores)
Sensibilidad Ventricular	1.0 a 8.0 mV (8 valores)
Porcentaje de histéresis	0% a 20%
Polaridad de estímulo	Unipolar/Bipolar
Polaridad de sensado	Unipolar/Bipolar

Tabla 4.1: Especificación de datos para programar un marcapasos bicameral

Por ejemplo, se conoce que los modos de funcionamiento de un marcapasos puede ser:

- Modos unicamerales ventriculares
- Modos unicamerales auriculares
- Modos bicamerales
- Modo apagado

Además, todos estos modos se clasifican en:

- Modo de sensado de la actividad cardíaca
- Modo de estimulación cardíaca
- Modo de sensado y estimulación cardíaca

Luego de seleccionado un modo, dependiendo de qué tipo de modo se seleccionó, ya sea de sensado, estimulación o ambos, se deben utilizar los siguientes campos:

Si se seleccionó un modo de sensado se deben especificar los campos:

- Refractario
- Sensibilidad
- Polaridad del sensado

Si se seleccionó un modo de estimulación se deben especificar los campos:

- Frecuencia y período
- Ancho del pulso
- Amplitud del pulso
- Polaridad del pulso

Aparte de estas restricciones en cuanto al modo de funcionamiento seleccionado existen otras con respecto a los valores ingresados en los demás parámetros. Algunas son propiedades que se deben cumplir entre dos campos como son el caso de la frecuencia y el período que una es inversa de la otra, y las demás son validaciones de contralor, como lo son las siguientes:

- Los refractarios deben ser menores que el período.
- Los refractarios deben ser menores que el período de *trigger*.
- El período de *trigger* debe ser menor que el período.
- El período de la frecuencia máxima debe ser menor que el período.
- La suma de cualquiera de los refractarios más el período AV debe ser menor que el período de la frecuencia máxima.
- El *blanking* debe ser menor que los períodos AV.

4.2 Implementación del formulario

En base a todas las restricciones mencionadas en la sección anterior y al código fuente de la especificación de la interfaz realizada por Degrandi, comenzamos a encontrar diferencias entre los lenguajes aunque esta evaluación la dejaremos para más adelante luego de terminada la implementación en InterfLan.

Sin más preámbulos comenzaremos por definir todos los campos que necesitaremos para este formulario. Lo único que no podremos definir inmediatamente es el tipo de control para cada uno de ellos ya que eso dependerá de su dominio o si es calculado o no.

Al comenzar con el primer campo (el modo de funcionamiento) debemos especificar un nuevo tipo enumerado para restringir los modos.

```
enum pm_mode [("VOO", VOO), ("OVO", OVO), ("VVI", VVI),
    ("VVT", VVT), ("AOO", AOO), ("OAO", OAO),
    ("AAI", AAI), ("AAT", AAT), ("VDD", VDD),
    ("DDD", DDD), ("OOO", OOO)] ;
```

Código 4.1: Enumerado para el modo de funcionamiento

Declarado ya el nuevo tipo ahora podemos definir el control para el campo del modo de funcionamiento y no tendrá más restricciones más allá de que es un campo obligatorio.

```
select combo mode:pm_mode not_empty;
```

Código 4.2: Definición del campo modo de funcionamiento

De la misma forma tendremos definidos otros campos de un nuevo enumerado como son los casos de: la polaridad del pulso (unipolar/bipolar) y la amplitud del pulso (2.5 V y 5.0 V). Estos campos se pueden observar en el **anexo C.3**, así como también la definición completa del formulario.

También podemos definir todas las restricciones de habilitación de campos. Tenemos algunos campos que se van a habilitar en cualquier modo salvo el modo apagado, como lo vemos en el **Código 4.3**, y otros campos que se habilitan con varios modos, como por ejemplos todos aquellos que se deben habilitar en modo de sensado (**Código 4.4**).

```
input period:integer let
  enable (mode != pm_mode.OOO);
end
```

Código 4.3: Campo habilitado en todos los modos salvo apagado

```
input atrium_sensitivity:float let
  enable (mode in [OAO, AAI, AAT, VDD, DDD]);
  ...
end
```

Código 4.4: Campo habilitado solo en modo de sensado

Otras restricciones importantes para este formulario son las que delimitan un rango para algún campo como puede ser el caso del AV post estímulo que tiene un rango entre 63 y 281 *ms*, el problema que se nos ha planteado aquí es que la especificación determina que en ese rango sólo son válidos 46 valores pero en ninguna parte del trabajo [1] se pudo hallar de qué manera se distribuían esos valores. Por este motivo hemos decidido, en primera instancia, dar por válido cualquier valor entre el rango indicado.

```
input pace_av:integer let
  enable (mode == pm_mode.DDD);
  range 63..281;
end
```

Código 4.5: Campo con restricción de rango

También se podría visualizar como un rango el porcentaje de histéresis, pero en este caso no es un valor que el programador deba ingresar sino que éste se calcula en base a otros campos y ese resultado debe estar entre 0% y 20%. Entonces esto nos motivó a que este campo sea un `label` que se calcule en base a la frecuencia base y la frecuencia de histéresis, y además cuyo resultado se encuentre entre 0 y 20.

```
label hyst_percentage:integer let
  calculate (100 * (rate - hyst_rate)/rate);
  check (self <= 20) "El porcentaje de histéresis debe ser menor o igual a
      20%";
  check (self >= 0) "El porcentaje de histéresis debe ser mayor o igual a
      0%";
end
```

Código 4.6: Porcentaje de histéresis con restricción

Por último veremos el caso de las verificaciones en cuanto a las restricciones adicionales que se enumeran en la especificación. Hay dos tipos de éstas: unas que relacionan dos campos entre sí, por ejemplo la relación inversa entre frecuencia y período, y las verificaciones de contralor como por ejemplo que un campo sea menor que otro. Para estas últimas, veremos el caso de "La suma de cualquiera de los refractarios más el período AV debe ser menor que el período de la frecuencia máxima" donde hemos decidido colocar esta restricción en el campo que se ingresa más al final del formulario ya que a partir de allí es donde se podrá "disparar" esta validación suponiendo que los demás campos que relaciona ya tienen datos. Además estas verificaciones tendrán un mensaje de error en caso que no se cumpla la condición para advertir al usuario.

```
input rate:integer let
  enable (mode != pm_mode.OOO);
  range 32..120;
  equity period (60000/self) (60000/period);
end

input period:integer let
  enable (mode != pm_mode.OOO);
end
```

Código 4.7: Campos con relaciones inversas

```
input ventricle_refractory:integer let
  enable (mode in [OVO, VVI, VVT, VDD, DDD]);
  range 195..430;
  check (self < period) "El refractario ventricular debe ser menor que el
      período";
  check (self < trigger_period) "El refractario ventricular debe ser menor
       que el período de trigger";
  check (self + pace_av < upper_period) "La suma del refractario
      ventricular más el período AV de estimulación debe ser menor que el per
      íodo correspondiente a la frecuencia máxima";
  check (self + sense_av < upper_period) "La suma del refractario
      ventricular más el período AV de sensado debe ser menor que el período
      correspondiente a la frecuencia máxima";
end
```

Código 4.8: Campo con verificaciones de contralor

4.3 Comparación de lenguajes

Luego de especificado el formulario en InterfLan veremos cuáles son las principales ventajas y desventajas de esta implementación con respecto a la hecha en el lenguaje implementado por Degrandi.

Ventajas

- Toda información en cuanto a las restricciones, validaciones, etc. están establecidas junto a la definición del campo, por lo cual si otro desarrollador quisiera leer el código puede obtener toda esa información en un mismo lugar. En el lenguaje de Degrandi, primero se especifican como

`struct` los campos que contiene el formulario y luego se especifican todas las relaciones por separado. Por consiguiente, creemos que estamos aportando un mejor nivel de expresividad y legibilidad al código producido en InterfLan.

- La sintaxis se muestra más natural que la presentada por Degrandi, nuestro lenguaje se puede leer casi como si fuera la especificación dada por el cliente, en cambio en el otro lenguaje hay bastante código que no es tan legible, sobre todo en funciones para obtener el dominio de un campo, recorrer valores, etc.

- En nuestro lenguaje es claro el tipo de control que se utiliza para cada campo, una cosa que no se nota clara en la definición de los campos en el lenguaje de Degrandi. No hay explícitamente una forma de declarar de qué tipo va a ser un campo para el usuario, esto también se podría explicar porque siempre se calculan los valores correctos y todo es seleccionable.

- Nuestro lenguaje es completamente declarativo, salvo la definición de funciones, en contraposición al otro lenguaje en donde se puede observar código procedural, por ejemplo, para calcular los valores de un dominio. Pensamos que orientar nuestro lenguaje para un lado más declarativo hace que el lenguaje quede de un nivel de abstracción superior y más cercano a la especificación escrita en lenguaje natural.

Desventajas

- En el lenguaje de Degrandi hay casos que se presentan los valores correctos para seleccionar, por lo tanto se está previniendo al error desde antes que el usuario seleccione el valor. Estos son los casos cuando se define una relación de actualización en un campo, por lo cual cuando cambian algunos campos, en el primero se calculan qué valores serán válidos y se pondrán a disposición del usuario para su selección. En nuestro lenguaje, los errores recién se establecen luego de especificado el valor, ya que todas las validaciones se realizan luego de que el campo contiene un valor.

- Notamos que hay un trabajo más profundo en el lenguaje de Degrandi en cuanto a los dominios, se están calculando cada vez que cambia un campo relacionado y además están casi todos especificados como un tipo de dato nuevo (igualmente no vimos en su trabajo cómo se definen éstos).

- Algo que notamos muy conveniente en el lenguaje de Degrandi es que se pueden abstraer ciertas secciones del formulario y así reutilizar un grupo de campos nuevamente en otro formualrio. Esto sería conveniente por ejemplo en el formulario de los marcapasos donde la programación de los valores de ambas cámaras es idéntica. En nuestro lenguaje hemos tenido que repetir los mismos campos para ambas cámaras, en cambio en la definición de Degrandi se puede observar cómo se define este grupo de campos en una nueva estructura y luego se utiliza dos veces para cada cámara.

5 Conclusiones y trabajo a futuro

De este trabajo podemos concluir que hemos logrado que la sintaxis de nuestro lenguaje sea bastante cercana al lenguaje natural, haciendo de esta manera más fácil de leer y entender, y que además no requiere de tanto conocimiento de programación para poder realizar un formulario. Además hemos comprobado que existen ciertas validaciones muy complejas por lo cual no hemos de asegurar al cien por ciento haber cubierto todos los escenarios posibles, aunque igual creemos que las validaciones contempladas en el lenguajes son satisfactorias. También podemos afirmar que el poder de expresión de InterfLan es lo suficientemente amplio como para poder abarcar todo tipo de formularios o interfaces, como lo hemos experimentado con el ejemplo de la interfaz de programación de marcapasos cardíacos y otros ejemplos interesantes.

A futuro pensamos que sería muy conveniente tener un generador de código que tome el árbol de sintaxis con tipos y generar el formulario con alguno de los *frameworks* de *JavaScript* que hemos analizado en este trabajo. Además podríamos evaluar e investigar más acerca sobre tipos dependientes ya que pensamos que varias restricciones que hoy en día manejamos a nivel de sentencia para cada control, podría estar implícita en el tipo de datos. Otro aspecto importante que sería muy conveniente a futuro, es el hecho de abstraer secciones de un formulario para poder reutilizarlas en el mismo como vimos en el resultado de la experimentación con el ejemplo de los marcapasos.

6 Referencias Bibliográficas

[1] S. Degrandi, "Herramientas de especificación y generación de módulos de entrada y validación de datos," PEDECIBA Informática - InCo, Facultad de Ingeniería - UdeLaR, Montevideo, Uruguay, 2005.

[2] J. Helms, *User Interface Markup Language (UIML) Version 4.0.* Oasis, 2008.

[3] Google, "AngularJS API Docs." [Online]. Available: https://docs.angularjs.org/api [Accessed: 12-Sep-2016].

[4] "Knockout: Introduction." [Online]. Available: http://knockoutjs.com/documentation/introduction.html [Accessed: 12-Sep-2016].

[5] "Ruby on Rails: Guides." [Online]. Available: http://guides.rubyonrails.org/ [Accessed: 12-Sep-2016].

[6] "Django Documentation." [Online]. Available: https://docs.djangoproject.com/en/1.10/ [Accessed: 12-Sep-2016].

[7] A. Ranta, *Implementing Programming Languages.* Londres, Reino Unido: College Publications, 2012.

[8] M. Forsberg and A. Ranta, "The Labelled BNF Grammar Formalism," Department of Cumputing Science - Chalmers University of Technology and the University of Gothenburg, Sweden, SE-412 96, February 2005.

Anexos

A Gramática

This section was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language.

The lexical structure of Interflan

Literals

String literals *String* have the form `"`x`"`, where x is any sequence of any characters except `"` unless preceded by \.

Integer literals *Integer* are nonempty sequences of digits.

Double-precision float literals *Double* have the structure indicated by the regular expression `digit+ '.' digit+ ('e' ('-')? digit+)?` i.e.\ two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.

Datetime literals are recognized by the regular expression ‘”” [”12”] digit digit digit ’-’ [”01”] digit ’-’ [”0123”] digit ’T’ [”012”] digit ’:’ [”012345”] digit ””’‘

Id literals are recognized by the regular expression ‘letter (letter | digit | ’_’)*‘

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in Interflan are the following:

`add_days`	`add_hours`	`add_minutes`	`add_months`
`add_years`	`as`	`bool`	`calculate`
`case`	`check`	`combo`	`datetime`
`domain`	`enable`	`end`	`enum`
`enumerate`	`equity`	`false`	`float`
`form`	`function`	`hide_unless`	`in`
`input`	`integer`	`label`	`let`
`not_empty`	`now`	`of`	`otherwise`
`radio`	`range`	`regex`	`select`
`self`	`string`	`subs_days`	`subs_hours`
`subs_minutes`	`subs_months`	`subs_years`	`true`

The symbols used in Interflan are the following:

:	;	(	)
[	]	,	..
.	!	*	/
%	+	-	<
>	<=	>=	==
!=	&&	||	->

Comments

Single-line comments begin with #.There are no multiple-line comments in the grammar.

The syntactic structure of Interflan

Non-terminals are enclosed between < and >. The symbols -> (production), | (union) and **eps** (empty rule) belong to the BNF notation. All other symbols are terminals.

Program	->	`form` *Id* `let` *[Def]* `end`
Def	->	`input` *Name* `:` *Type Body*
	|	`label` *Name* `:` *Type Body*
	|	`select` *SelectCtrl Name* `:` *Type Body*
	|	`function` *Id* `:` *Type FuncBody*
	|	`enum` *Id EnumBody*
[Def]	->	**eps**
	|	*Def [Def]*
Name	->	*Id*

| | *String* `as` *Id*
Type -> *PrimType*
| *Id*
Body -> `;`
| *Stm*
| `let` *[Stm]* `end`
FuncBody -> `let` *String* `end`
| `(` *[Param]* `)` `let` *String* `end`
EnumBody -> `[` *[EnumValue]* `]` `;`
| `let` *Stm* `end`
EnumValue -> `(` *String* `,` *Id* `)`
[EnumValue] -> *EnumValue*
| *EnumValue* `,` *[EnumValue]*
Stm -> `not_empty` `;`
| `regex` *String* `;`
| `range` *Exp* `..` *Exp* `;`
| `check` *Exp CheckBody*
| `enable` *EnableBody* `;`
| `calculate` `(` *Exp* `)` `;`
| `domain` `(` *Exp* `)` `;`
| `equity` *Id* `(` *Exp* `)` `(` *Exp* `)` `;`
[Stm] -> **eps**
| *Stm [Stm]*
EnableBody -> `(` *Exp* `)`
| `hide_unless` `(` *Exp* `)`
CheckBody -> `;`
| *String* `;`
Exp17 -> `true`
| `false`
| *Integer*
| *Double*
| *String*
| `now`
| *Datetime*
| `self`
| *Id*
| *Id* `.` *Id*
| `[` *[EnumValue]* `]`
| *Id* `in` `[` *[Id]* `]`
| `(` *Exp* `)`
Exp16 -> *Id* `(` *[Exp]* `)`

```
                |   Exp17
Exp15           ->  ! Exp15
                |   Exp16
Exp14           ->  Exp14 * Exp15
                |   Exp14 / Exp15
                |   Exp14 % Exp15
                |   Exp15
Exp13           ->  Exp13 + Exp14
                |   Exp13 - Exp14
                |   Exp14
Exp12           ->  DTOp ( Exp12 , Exp13 )
                |   Exp13
Exp11           ->  Exp11 < Exp12
                |   Exp11 > Exp12
                |   Exp11 <= Exp12
                |   Exp11 >= Exp12
                |   Exp12
Exp10           ->  Exp10 == Exp11
                |   Exp10 != Exp11
                |   Exp11
Exp9            ->  Exp9 && Exp10
                |   Exp10
Exp7            ->  Exp7 || Exp9
                |   Exp8
Exp6            ->  case Exp7 of [Branch] CaseEnd
                |   Exp7
CaseEnd         ->  end
                |   otherwise -> Exp end
Exp             ->  Exp1
Exp1            ->  Exp2
Exp2            ->  Exp3
Exp3            ->  Exp4
Exp4            ->  Exp5
Exp5            ->  Exp6
Exp8            ->  Exp9
[Exp]           ->  eps
                |   Exp
                |   Exp , [Exp]
Branch          ->  Exp -> Exp
[Branch]        ->  Branch
                |   Branch ; [Branch]
```

Param	->	*Id* : *Type*
[Param]	->	*Param*
	|	*Param* , *[Param]*
SelectCtrl	->	`combo`
	|	`radio`
PrimType	->	`bool`
	|	`integer`
	|	`float`
	|	`datetime`
	|	`string`
	|	`enumerate`
DTOp	->	`add_years`
	|	`add_months`
	|	`add_days`
	|	`add_hours`
	|	`add_minutes`
	|	`subs_years`
	|	`subs_months`
	|	`subs_days`
	|	`subs_hours`
	|	`subs_minutes`
[Id]	->	*Id*
	|	*Id* , *[Id]*

B *Type Checker*

Para la generación del *Type Checker* realizamos las reglas de validación de estructuras y las de inferencia de tipos para poder reflejar luego las mismas en el código de verificación de tipos. Las relgas tienen el siguiente formato:

$$\frac{P_1, ..., P_n}{\text{C}} \, S_1, ..., S_m$$

Donde $P_1, ..., P_n$ son las condiciones que se tienen que cumplir o premisas, C es la conclusión o la culminación exitosa de la validación y $S_1, ..., S_m$ las condiciónes adicionales que se tienen que cumplir.

Para poder realizar las validaciones de los tipos en las reglas precedentes fue necesario generar un *contexto* que en las mismas fue nombrado Γ. Este contexto está conformado por 4 grandes secciones.

- Una lista de los controles disponibles en el formulario junto con su tipo para poder verificar las variables dentro de las expresiones, las cuales llevan el nombre de los controles.
- Una lista de las funciones con sus respectivas firmas (tipo de devolución de la función y lista ordenada de los tipos de los parámetos).
- Una lista de los enumerados junto con los valores que lo componen.
- Finalmente un grupo de grafos con las dependencias generadas entre los distintos controles, un grafo por cada tipo de sentencia en la que se genera una dependencia, para validar la ausencia de circularidades e inconsistencias en las relaciones.

Para hacer uso de los elementos asociados a los nombres de cada estructura (controles, funciones y enumerados), podemos pedirle al contexto que busque por el nombre de la estructura de la forma $\Gamma(\{nombre\})$. El proceso de verificación de tipo y de validéz de las dependencias consta de las siguientes partes:

- Generación del contexto

- Generación del grafo de dependencias entre controles
- Verificación de los tipos de las expresiones
- Verificación de los grafos de dependencias (Busca de circularidades o inconsistencias)

Cada una de las partes anteriores excepto la verificación de los grafos, necesita de una *pasada* por el código, dejando en total 3 pasadas.

B.1 Reglas del lenguaje

A continuación se especifican las reglas de validación de los tipos en el código. Formulario:

$$\frac{\Gamma \vdash [d_1, d_2, ..., d_n] \text{ ; valid}}{\text{form n } [d_1, d_2, ..., d_n] \text{ ; valid}}$$

Definiciones:

$$\frac{\Gamma, self : t \vdash [s_1, s_2, ..., s_n] \text{ ; valid}}{\Gamma \vdash \text{input n t } [s_1, s_2, ..., s_n] \text{ ; valid}} \begin{array}{c} \#\text{Calculate} \leq 1 \\ \#\text{Equity} \leq 1 \\ \#\text{Calc} + \#\text{Equity} \leq 1 \end{array}$$

$$\frac{\Gamma, self : t \vdash [s_1, s_2, ..., s_n] \text{ ; valid}}{\Gamma \vdash \text{label n t } [s_1, s_2, ..., s_n] \text{ ; valid}} \begin{array}{c} \#\text{Calculate} \leq 1 \\ \#\text{Equity} = 0 \end{array}$$

$$\frac{\Gamma, self : t \vdash [s_1, s_2, ..., s_n] \text{ ; valid}}{\Gamma \vdash \text{select n t } [s_1, s_2, ..., s_n] \text{ ; valid}} \begin{array}{c} \text{t is enumerate} \\ \#\text{Calc} + \#\text{Equity} = 0 \end{array}$$

$$\frac{}{\Gamma \vdash \text{enum n } [v_1, v_2, ..., v_n] \text{ ; valid}} \begin{array}{c} [v_1, v_2, ..., v_n] \text{ no} \\ \text{tiene repetidos} \end{array}$$

$$\frac{\Gamma \vdash \text{case e } [e_1 \to v_1, ...e_n \to v_n] : \text{enumerate}}{\Gamma \vdash \text{enum n let domain (case e } [e_1 \to v_1, ...e_n \to v_n]); \text{ end valid}}$$

Sentencias:

$$\frac{}{\Gamma, self : t \vdash \text{not\_empty ; valid}}$$

$$\frac{}{\Gamma, self : t \vdash \text{regex s ; valid}} \text{ t} = \text{string}$$

$$\frac{\Gamma, self : t \vdash \text{e1} : t_1, \Gamma, self : t \vdash \text{e2} : t_2, t_1 = t_2}{\Gamma, self : t \vdash \text{range e1 e2 ; valid}} t_1, t_2 \neq \text{enumerate}$$

$$\frac{\Gamma, self : t \vdash \text{e} : \text{bool}}{\Gamma, self : t \vdash \text{check e ; valid}}$$

$$\frac{\Gamma, self : t \vdash \text{e} : \text{bool}}{\Gamma, self : t \vdash \text{enable e ; valid}}$$

$$\frac{\Gamma, self : t \vdash \text{e} : \text{t' , t'} = \text{t}}{\Gamma, self : t \vdash \text{calculate e ; valid}}$$

$$\frac{\Gamma, self : t \vdash e_1 : t_1, \Gamma, self : t \vdash e_2 : t_2, t_1 = t_2 = \text{t}}{\Gamma, self : t \vdash \text{equity n } e_1\ e_2 \text{ ; valid}}\ \Gamma(n) = t$$

Expresiones:

$$\frac{}{\Gamma, self : t \vdash b : \text{bool}}$$

$$\frac{}{\Gamma, self : t \vdash i : \text{integer}}$$

$$\frac{}{\Gamma, self : t \vdash d : \text{double}}$$

$$\frac{}{\Gamma, self : t \vdash s : \text{string}}$$

$$\frac{}{\Gamma, self : t \vdash \text{now} : \text{datetime}}$$

$$\frac{}{\Gamma, self : t \vdash \text{"YYYY-MM-DDTHHmm"} : \text{datetime}}$$

$$\frac{}{\Gamma, self : t \vdash self : \text{t}}$$

$$\frac{\Gamma(x) = t'}{\Gamma, self : t \vdash x : \text{t'}}$$

$$\frac{}{\Gamma, self : t \vdash en.val : \text{en}}\ \begin{matrix} en \in \Gamma \\ val \in \Gamma(en) \end{matrix}$$

$$\frac{}{\Gamma, self : t \vdash [en_1, ..., en_n] : \text{enumerate}}$$

$$\frac{\Gamma(f) = (t', [t'_1, ..., t'_n]), \Gamma, self : t \vdash [e_1, ..., e_n] : [t_1, ...t_n]}{\Gamma, self : t \vdash f(e_1, ..., e_n) : \text{t'}}\ [t_1, ..., t_n] = [t'_1, ..., t'_n]$$

$$\frac{\Gamma, self : t \vdash \text{e} : \text{bool}}{\Gamma, self : t \vdash \text{! e} : \text{bool}}$$

$$\frac{\Gamma, self : t \vdash e_1 : \text{t'}, \Gamma, self : t \vdash e_1 : \text{t'}}{\Gamma, self : t \vdash e_1 * e_2 : \text{t'}}\ \text{t'is integer or double}$$

$$\frac{\Gamma, self : t \vdash e_1 : \text{t'}, \Gamma, self : t \vdash e_1 : \text{t'}}{\Gamma, self : t \vdash e_1 / e_2 : \text{t'}} \text{ t'is integer or double}$$

$$\frac{\Gamma, self : t \vdash e_1 : \text{t'}, \Gamma, self : t \vdash e_1 : \text{t'}}{\Gamma, self : t \vdash e_1 - e_2 : \text{t'}} \text{ t'is integer or double}$$

$$\frac{\Gamma, self : t \vdash e_1 : \text{t'}, \Gamma, self : t \vdash e_1 : \text{t'}}{\Gamma, self : t \vdash e_1 \% e_2 : \text{t'}} \text{ t'is integer or double}$$

$$\frac{\Gamma, self : t \vdash e_1 : t_1, \Gamma, self : t \vdash e_1 : t_2 \text{ , } \max(t_1, t_2) = \text{t'}}{\Gamma, self : t \vdash e_1 + e_2 : \text{t'}} \text{ t'is integer or double or string}$$

$$\frac{\Gamma, self : t \vdash \text{e : datetime, } \Gamma, self : t \vdash \text{c : integer}}{\Gamma, self : t \vdash \text{add\_? e c : datetime}}$$

$$\frac{\Gamma, self : t \vdash \text{e : datetime, } \Gamma, self : t \vdash \text{c : integer}}{\Gamma, self : t \vdash \text{subs\_? e c : datetime}}$$

$$\frac{\Gamma, self : t \vdash e_1 : \text{t'}, \Gamma, self : t \vdash e_1 : \text{t'}}{\Gamma, self : t \vdash e_1 < e_2 : \text{bool}} \text{ t'is not enumerate}$$

$$\frac{\Gamma, self : t \vdash e_1 : \text{t'}, \Gamma, self : t \vdash e_1 : \text{t'}}{\Gamma, self : t \vdash e_1 > e_2 : \text{bool}} \text{ t'is not enumerate}$$

$$\frac{\Gamma, self : t \vdash e_1 : \text{t'}, \Gamma, self : t \vdash e_1 : \text{t'}}{\Gamma, self : t \vdash e_1 \leq e_2 : \text{bool}} \text{ t' is not enumerate}$$

$$\frac{\Gamma, self : t \vdash e_1 : \text{t'}, \Gamma, self : t \vdash e_1 : \text{t'}}{\Gamma, self : t \vdash e_1 \geq e_2 : \text{bool}} \text{ t' is not enumerate}$$

$$\frac{\Gamma, self : t \vdash e_1 : \text{bool}, \Gamma, self : t \vdash e_1 : \text{bool}}{\Gamma, self : t \vdash e_1 \&\& e_2 : \text{bool}}$$

$$\frac{\Gamma, self : t \vdash e_1 : \text{boo'}, \Gamma, self : t \vdash e_1 : \text{bool}}{\Gamma, self : t \vdash e_1 || e_2 : \text{bool}}$$

$$\frac{\Gamma, self : t \vdash \text{e : t''}, \Gamma, self : t \vdash [e_1, ..., e_n] : [\text{t''}, ..., \text{t''}], \Gamma, self : t \vdash [v_1, ..., v_n, v'] : [\text{t'}, ..., \text{t'}]}{\Gamma, self : t \vdash \text{case e } [e_1 \rightarrow v_1, ...e_n \rightarrow v_n] \text{ otherwise} \rightarrow \text{v': t'}}$$

Expliquemos un poco más esta regla. La estructura case se puede utilizar para poder elegir una expresión en base al valor de otra. En la estructura, e es la expresión que se evalúa y si su valor coincide con el de algunas de las expresiones e_t el resultado de la expresión será el valor v_t asociado al mismo. Para que una expresión case sea correcta, todas las expresiones internas a la misma deven ser válidas y además todas las expresiones a la izquierda de las "ramas" (e_t) y la expresión a evaluarse en el *case* tienen que ser del mismo tipo. Además las expresiones a la derecha de las "ramas" (v_t) deben ser todas del mismo tipo también. Estableciendo así el tipo del case entero que es el mismo que el de las expresiones a la derecha de las "ramas".

C Casos de Estudio

C.1 Primer Ejemplo - Línea de Crédito

```
form linea_de_credito let
    input "Nombre" as nombre:string let
        not_empty ;
    end

    input "Apellido" as apellido:string let
        not_empty ;
    end

    input "Dirección de correo electrónico" as email:string let
        not_empty ;
        regex "(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+
            $)" ;
    end

    function verificar_cedula:bool (cedula:integer) let
        "cuerpo de la función en JavaScript"
    end

    input "Cédula de identidad" as cedula:integer let
        not_empty ;
        check (verificar_cedula(self)) ;
    end

    input "Fecha de Nacimiento" as f_necimiento:datetime let
        not_empty ;
        range (subs_years now 150)..(subs_years now 18) ;
    end

    function calcular_edad:integer (fnac:datetime) let
        "cuerpo de la función en JavaScript"
```

```
end

label "Edad" as edad:integer let
    calculate (calcular_edad(f_nacimiento)) ;
end

 enum estado_civil [("Casado/a",CAS),("Soltero/a",SOL),
                ("Concubino/a",CON),("Viudo/a",VIU),
                ("Divorciado/a", DIV)] ;

select combo "Estado civil" as ecivil:estado_civil let
    not_empty ;
end

input "Fecha concubinato o matrimonio" as f_mat:datetime let
    enable (ecivil in [CAS,CON]) ;
    range (add_years f_nacimiento 14)..(now) ;
end

enum paises [("Uruguay",UY), ("Brasil", BR), ("Argentina", AR)] ;

select combo "País de nacimiento" as p_nacimiento:paises let
end

enum ciudades let
    domain (case p_nacimiento of
                paises.UY -> [("Montevideo", MVD), ("Rocha", ROC)
                    ,
                            ("Canelones", CAN)] ;
                paises.BR -> [("Brasilia", BRS),("Fortaleza", FRT),
                            ("Recife",RCF)] ;
                paises.AR -> [("Buenos Aires", BAS), ("Santa Fe",
                    STF),
                            ("Córdoba", CBA)]
            end) ;
end

select combo "Ciudad de nacimiento" as c_nacimiento:ciudades let
end

input "Mínimo ingreso nominal" as ingresos:integer let
    not_empty ;
    equity credito (self * 2) (credito / 2) ;
```

```
    end

    input "Máximo línea de crédito" as credito:integer let
        not_empty ;
    end
end
```

Código C.1: Formulario completo.

C.2 Segundo Ejemplo - Problemas con dependencias

```
form dependencias let
    input A:integer let
        calculate (B * 2);
    end

    input B:integer let
        calculate (A / 2);
    end

    input C:integer;
end
```

Código C.2: Circularidad con dependencias calculate

```
form dependencias let
    input A:integer let
        equity B (self / 2) (B * 2);
    end

    input B:integer;

    input C:integer;
end
```

Código C.3: Corrección del error del ejemplo anterior

```
form dependencias let
    input A:integer let
        equity B (self / 2) (B * 2);
    end
```

```
    input B:integer let
        calculate (C * 2);
    end

    input C:integer;
end
```

Código C.4: Error causado por equity y calculate

```
form dependencias let
    input A:integer let
        equity B (self / 2) (B * 2);
    end

    input B:integer;

    input C:integer let
        equity B (self * 23) (B * 23);
    end
end
```

Código C.5: Error causado por dos dependencias equity

C.3 Formulario de marcapasos

En este anexo se muestra el formulario completo de la especificación de la interfaz de programación de marcapasos formulada por Degrandi [1].

```
form pacemaker let

  enum pm_mode [("VOO", VOO),
    ("OVO", OVO),
    ("VVI", VVI),
    ("VVT", VVT),
    ("AOO", AOO),
    ("OAO", OAO),
    ("AAI", AAI),
    ("AAT", AAT),
    ("VDD", VDD),
    ("DDD", DDD),
    ("OOO", OOO)] ;

  enum polarity [("Unipolar", UNI), ("Bipolar", BIP)] ;
```

```
enum amplitude_range [("2.5 V", V2_5), ("5.0 V", V5_0)] ;

select combo mode:pm_mode not_empty;

input rate:integer let
  enable (mode != pm_mode.OOO);
  range 32..120;
  equity period (60000/self) (60000/period);
end

input period:integer let
  enable (mode != pm_mode.OOO);
end

input upper_rate:integer let
  enable (mode != pm_mode.OOO);
  range 60..150;
  equity upper_period (60000/self) (60000/upper_period);
end

input upper_period:integer let
  enable (mode != pm_mode.OOO);
  check (self < period) "El período correspondiente a la frecuencia má
      xima debe ser menor que el período" ;
end

input trigger_rate:integer let
  enable (mode in [VVT, AAT]);
  equity trigger_period (60000/self) (60000/trigger_period);
end

input trigger_period:integer let
  enable (mode in [VVT, AAT]);
end

input hyst_rate:integer;

label hyst_percentage:integer let
  calculate (100 * (rate - hyst_rate)/rate);
  check (self <= 20) "El porcentaje de histéresis debe ser menor o igual
       a 20%";
```

```
    check (self >= 0) "El porcentaje de histéresis debe ser mayor o igual
        a 0%";
  end

  input pace_av:integer let
    enable (mode == pm_mode.DDD);
    range 63..281;
  end

  input sense_av:integer let
    enable (mode in [VDD, DDD]);
    range 53..272;
  end

  input blanking:integer let
    enable (mode == pm_mode.DDD);
    range 21..68;
    check (self < pace_av) "El blanking debe ser menor que período AV de
        estimulación";
    check (self < sense_av) "El blanking debe ser menor que período AV
        de sensado";
  end

  input anti_pmt:bool not_empty;

# Cámara Aurícula
  select combo atrium_amplitude:amplitude_range let
    enable (mode in [AOO, AAI, AAT, DDD]);
  end

  input atrium_pulseWidth:integer let
    enable (mode in [AOO, AAI, AAT, DDD]);
    range 122..1464;
  end

  select radio atrium_pacePolarity:polarity let
    enable (mode in [AOO, AAI, AAT, DDD]);
  end

  input atrium_sensitivity:float let
    enable (mode in [OAO, AAI, AAT, VDD, DDD]);
    range 0.5..4.0;
  end
```

```
  select radio atrium_sensePolarity:polarity let
    enable (mode in [OAO, AAI, AAT, VDD, DDD]);
  end

  input atrium_refractory:integer let
    enable (mode in [OAO, AAI, AAT, VDD, DDD]);
    range 195..430;
    check (self < period) "El refractario auricular debe ser menor que el
        período";
    check (self < trigger_period) "El refractario auricular debe ser menor
         que el período de trigger";
    check (self + pace_av < upper_period) "La suma del refractario
        auricular más el período AV de estimulación debe ser menor que el per
        íodo correspondiente a la frecuencia máxima";
    check (self + sense_av < upper_period) "La suma del refractario
        auricular más el período AV de sensado debe ser menor que el período
        correspondiente a la frecuencia máxima";
  end


# Cámara Ventrícula
  select combo ventricle_amplitude:amplitude_range let
    enable (mode in [VOO, VVI, VVT, VDD, DDD]);
  end

  input ventricle_pulseWidth:integer let
    enable (mode in [VOO, VVI, VVT, VDD, DDD]);
    range 122..1464;
  end

  select radio ventricle_pacePolarity:polarity let
    enable (mode in [VOO, VVI, VVT, VDD, DDD]);
  end

  input ventricle_sensitivity:float let
    enable (mode in [OVO, VVI, VVT, VDD, DDD]);
    range 1.0..8.0;
  end

  select radio ventricle_sensePolarity:polarity let
    enable (mode in [OVO, VVI, VVT, VDD, DDD]);
  end
```

```
  input ventricle_refractory:integer let
    enable (mode in [OVO, VVI, VVT, VDD, DDD]);
    range 195..430;
    check (self < period) "El refractario ventricular debe ser menor que
        el período";
    check (self < trigger_period) "El refractario ventricular debe ser
        menor que el período de trigger";
    check (self + pace_av < upper_period) "La suma del refractario
        ventricular más el período AV de estimulación debe ser menor que el
        período correspondiente a la frecuencia máxima";
    check (self + sense_av < upper_period) "La suma del refractario
        ventricular más el período AV de sensado debe ser menor que el perí
        odo correspondiente a la frecuencia máxima";
  end
end
```