Documento de Cambios - Proyecto 3 PLE

Equipo:

- Juan Felipe Ochoa - 202320053
- Juan Felipe Hortúa - 202320918
- Juan José Cocomá - 202322061

---

1. Introducción

Este documento describe las modificaciones realizadas al lenguaje ALU para cumplir con los requisitos del Proyecto 3, específicamente la integración de TypePal para el sistema de tipos y análisis estático. El proyecto parte de una implementación base (proyecto 2) y evoluciona hacia una versión mejorada (proyecto3) que incorpora anotaciones de tipo y verificación estática.

2. Cambios Estructurales

2.1 Arquitectura General

versión anterior:

- Sintaxis simple sin anotaciones de tipo
- AST genérico sin información de tipos
- Sin análisis estático
- Módulos: Syntax.rsc, AST.rsc, Parser.rsc, Main.rsc, Plugin.rsc

versión mejorada:

- Sintaxis extendida con anotaciones de tipo
- AST con definiciones de tipo explícitas
- Integración completa de TypePal
- Nuevos módulos: Collect.rsc, TypeChecker.rsc, Implode.rsc, Generator1.rsc

3. Modificadores en la Gramática (Syntax.rsc)

3.1 Incorporación de Tipos Primitivos

Se añadió la sintaxis TypeName para soportar anotaciones de tipo:

$= "Int" \mid "Bool" \mid "Char" \mid "String" \mid Identifier;$

Este cambio permite que los usuarios especifiquen tipos tanto primitivos (*Int, Bool, Char, String)* como definidos por el usuario (mediante *Identifier*).

3.2 Palabras Reservadas Extendidas

Se agregaron las palabras clave de tipos al conjunto de palabras reservadas:

$"cond" \mid "do" \mid "data" \mid "end" \mid "for" \mid "from" \mid "then"$

$\mid "function" \mid "else" \mid "elseif" \mid "if" \mid "in" \mid "iterator"$

| "sequence" | "struct" | "to" | "tuple" | "type" | "with"

| "yielding" | "and" | "or" | "neg" | "true" | "false"

| "Int" | "Bool" | "Char" | "String" // ← NUEVOS

;

Esto previene que los usuarios utilicen nombres de tipos como identificadores, evitando ambigüedades.

3.3 Sintaxis para Identificadores Tipados

Aunque la gramática original no lo soportaba explícitamente, en la versión actualizada, se preparó el terreno para declaraciones tipadas. Por ejemplo, en la declaración de datos:

// Permitir campos con tipos explícitos

syntax TypedIdentifier = Identifier ":" TypeName | Identifier;

// Esto facilita expresiones como:

rep = struct(real: Int, img: Int)

4. Ampliación del AST

4.1 Definición de Tipos en el AST

Se introdujo un nuevo tipo de dato *Type* que representa los tipos del lenguaje:

data Type

= intType()

| boolType()

| charType()

| stringType()

| userType(str name)

| unknownType()

;

Esta estructura permite que cada expresión y declaración pueda llevar consigo información sobre su tipo, fundamental para el análisis estático.

4.2 Estructuras de Programa Simplificadas

El AST se simplificó para facilitar el análisis de TypePal:

data Program = program(list[Module] modules);

data Module

= funMod(FunctionDecl f)

| dataMod(DataDecl d) ;

En la primera entrega, los módulos incluían también sentencias globales (*stmt(Stmt statement))*, lo cual implicaba el análisis. En la nueva versión, se enfocaba en módulos funcionales y de datos.

4.3 Declaraciones de Función y Datos

Funciones:

$data\ FunctionDec\ =\ function(\ str\ name, list[str]\ params, list[Statement]\ body);$

Datos:

$data\ DataDecl\ =\ dataDecl(\ str\ name, list[str]\ fields);$

Estas estructuras son más directas y permiten a TypePal rastrear definiciones y usos de manera eficiente.

5. Corrección de Problemas de versión Anterior

5.1 Problemas Identificados

1. Ambigüedad en la gramática: Algunos tokens no estaban bien delimitados
2. Falta de manejo de ámbitos: Variables globales y locales podían colisionar
3. Sin validación de identificadores: No se verificaba si las funciones o variables existían antes de usarlas

5.2 Soluciones implementadas

Delimitación de tokens:

$lexical\ Identifier\ =\ [a-z]\ [a-z0-9\_]*\ !>>\ [a-z0-9\_]\ \backslash\ Reserved;$

El "!>>" previene coincidencias parciales.

Manejo de ámbitos:

$c.enterScope(current);$

$//\ldots definiciones\ locales\ \ldots$

$c.leaveScope(current);$

Cada función y bloque de datos tiene su propio ámbito. Todos los usos de identificadores se verifican contra definiciones previas, garantizando que no se usen variables o funciones no declaradas.

6. Implementación de TypePal

6.1 Módulo *Collect.rsc*

El módulo *Collect.rsc* constituye el núcleo de la integración con TypePal, implementando la lógica que permite identificar y catalogar todas las definiciones y usos de identificadores en el programa. Para esto, se definieron tres roles fundamentales de identificadores mediante el tipo de dato *IdRole: variableId()* para variables, *functionId()* para funciones y *dataTypeId()* para tipos de datos definidos por el usuario. Esta distinción es esencial porque TypePal necesita diferenciar el contexto en el que cada identificador es válido, evitando así conflictos entre nombres que podrían tener diferentes significados según su rol.

La recolección comienza a nivel de módulo, donde se establece un ámbito (scope) principal que contendrá todas las definiciones del programa. El proceso funciona mediante un patrón de visitante que recorre el árbol de sintaxis y, al encontrar cada tipo de elemento (funciones, datos, variables), invoca su método de recolección específico. Esta entrada y salida de ámbitos mediante enterScope y leaveScope es crucial porque permite a TypePal mantener un registro jerárquico de qué identificadores son visibles en cada punto del código, respetando las reglas de alcance léxico del lenguaje.

Para las abstracciones de datos, el proceso de recolección primero identifica el nombre del tipo de dato (por ejemplo, complex en complex = data...) y lo registra con el rol dataTypeId. Luego, crea un nuevo ámbito específico para ese tipo de dato, dentro del cual se recolectan todos sus componentes: constructores como struct y funciones asociadas como create o add. Esta estrategia de ámbitos anidados permite que las funciones dentro de un tipo de dato puedan acceder a sus constructores, mientras que estos no contaminen el espacio de nombres global.

La recolección de funciones sigue un patrón similar pero con atención especial a los parámetros. Cuando se encuentra una definición de función, primero se registra el nombre de la función con el rol functionId, luego se abre un nuevo ámbito para su cuerpo, y dentro de este ámbito se definen todos los parámetros como variableId. Esto garantiza que los parámetros solo sean visibles dentro de la función y no interfieran con otras definiciones. El cuerpo de la función se procesa recursivamente, identificando todas las expresiones y sentencias que contiene.

Finalmente, cuando se procesan expresiones, cada identificador encontrado se marca como un "uso" mediante c.use, indicando a TypePal que debe verificar que existe una definición correspondiente en algún ámbito accesible. TypePal puede entonces detectar errores como el uso de variables no declaradas o funciones inexistentes, validando que cada referencia corresponda efectivamente a una definición previa válida en el contexto apropiado.

Ejemplo ilustrativo:

$$sqrt\ =\ function(x, g)\ do$$

$$if\ good - enough(x, g)$$

$$then\ g$$

$$else\ sqrt(x, improve(x, g))$$

$$end$$

$$end\ sqrt$$

En este código, sqrt se registra como functionId en el ámbito global, x y g como variableId en el ámbito de la función, y los usos de good-enough e improve se marcan como referencias que TypePal verificará contra las definiciones existentes.

6.2 Módulo TypeChecker.rsc

El módulo TypeChecker.rsc actúa como coordinador del proceso completo de verificación de tipos, integrando las capacidades de recolección con el motor de resolución de TypePal. La función principal typeCheckTree recibe un árbol de sintaxis y ejecuta tres fases secuenciales: primero instancia un recolector configurado para el lenguaje ALU, luego ejecuta la fase de recolección que puebla el recolector con todas las definiciones y usos encontrados, y finalmente invoca el solver de TypePal que analiza esta información para detectar inconsistencias.

El resultado de este proceso es un TModel, una estructura de datos que contiene tanto las definiciones resueltas como cualquier mensaje de error detectado. La función typeCheckFile utiliza este modelo para proporcionar retroalimentación al usuario, mostrando mensajes de éxito cuando no se encuentran errores o listando detalladamente cada problema detectado. Este diseño modular permite que el análisis de tipos se ejecute tanto desde el terminal de Rascal para pruebas manuales como desde el plugin del IDE para proporcionar diagnósticos en tiempo real mientras el usuario edita el código.

6.3 Integración en *Plugin.rsc*

La integración final con el IDE se realiza mediante la actualización del módulo Plugin.rsc, que ahora incluye un summarizer capaz de proporcionar diagnósticos en tiempo real. La función TModelFromTree

encapsula el proceso completo de análisis de tipos, creando un recolector, ejecutando la fase de recolección y resolviendo todas las referencias mediante el solver de TypePal. El aluSummarizer utiliza esta función para generar un resumen (Summary) cada vez que se modifica un archivo .alu, extrayendo tanto los mensajes de error como las definiciones resueltas del modelo de tipos resultante. Este resumen se integra con los servicios de lenguaje del IDE, permitiendo que los errores se muestren como subrayados rojos en el código, que la información de hover revele detalles sobre identificadores, y que el autocompletado sugiera definiciones válidas. El manejo de excepciones mediante try-catch asegura que errores durante el análisis no bloqueen el IDE, retornando un resumen vacío en caso de fallo. Esta arquitectura proporciona una experiencia de desarrollo fluida donde los errores semánticos y de tipos se detectan instantáneamente durante la edición, sin necesidad de compilar o ejecutar el programa.

7. ¿Por qué funciona esta implementación?

La efectividad de esta implementación radica en una arquitectura modular bien definida donde cada componente tiene responsabilidades claras: Syntax.rsc maneja la gramática y validación sintáctica, Collect.rsc implementa la semántica mediante roles y ámbitos de identificadores, TypeChecker.rsc coordina el proceso de verificación, y Plugin.rsc integra todo con el IDE para diagnósticos en tiempo real. Esta separación permite que cada módulo sea independiente, testeable y fácil de mantener. El aprovechamiento de TypePal como framework subyacente resulta fundamental, ya que proporciona Collectors, Solvers y TModels optimizados en lugar de requerir una implementación desde cero, garantizando un comportamiento robusto y eficiente.

El sistema implementa análisis incremental donde cada modificación de archivo activa automáticamente el aluSummarizer, que parsea solo el archivo editado, genera un TModel actualizado y muestra errores instantáneamente en el IDE. TypePal garantiza la verificación de consistencia asegurando que cada uso tenga una definición válida, previniendo redefiniciones en el mismo ámbito y verificando que las reglas de alcance léxico se respeten correctamente. Por ejemplo, en el código sqrt = function(x, g) do result = x + y end sqrt, TypePal detecta inmediatamente que y no está definido y genera un error con su ubicación exacta, demostrando cómo el sistema previene errores comunes antes de la ejecución.