FuncLangArith: Enhancing Abstract Interpretation with Concrete Arithmetic

Cade Breeding - *breeding@iastate.edu*
Daniel Duerr - *djduerr@iastate.edu*
Jesse Slater - *jcslater@iastate.edu*
Mason Wichman - *mwichman@iastate.edu*

Introduction:

This project aims to enhance funcLang by introducing abstract variable support, allowing developers to define variables in an abstract manner. Unlike traditional abstract interpretation, we selectively apply it to expressions involving abstract values, providing nuanced abstraction without constraining the entire codebase. This document serves as a guide, exploring syntax, detailing modifications to arithmetic operations, and unveiling the nuances associated with selective interpretation. Developers will gain a clearer understanding of design decisions and implementation choices, offering a roadmap for utilizing funcLang with this extended feature.

Overview of 'funcLang' Language:

Funclang is a programming language that integrates arithmetic operations with logical constructs. It supports the definition of variables and functions, providing a robust foundation for data manipulation. Notably, Funclang includes boolean operations, allowing the creation of conditional structures like if statements with equality, less than, and greater than comparisons. The language is designed with a focus on clarity and conciseness in its syntax, aiming to facilitate efficient and readable code. Funclang is suited for applications requiring a balance between mathematical computations and logical reasoning

In addition to its feature set, Funclang employs an Abstract Syntax Tree (AST) parsing mechanism and follows a visit pattern. This approach involves handling each type of expression through its dedicated visit function. The use of AST parsing provides a structured representation of the program's syntax, allowing for systematic analysis and interpretation. The visit pattern further enhances the language's extensibility, as each expression type can be processed independently within its designated visit function. This design choice in Funclang contributes to a modular and maintainable codebase, enabling developers to navigate and manipulate the language efficiently.

Extension: Introducing Abstract Variables:

To incorporate abstract variables into Funclang, a new class, AbstractValue, was created that implements the Value interface. It contains an Enum of Vals related to possible elements inside the abstract domain. Furthermore, AbstractValue contains a HashSet of Val to represent the different elements of the domain that are present in the AbstractValue. Additionally, the AbstractValue class provides a multitude of methods, constructors, and utility methods to perform different Funclang expressions on AbstractValue's HashSet of Vals. Changes to the Value.java file were made to include AbstractValues.

If a developer using Funclang wants to specify variables and their corresponding elements of the abstract domain, they need to make modifications to their .scm file. For example, adding "(abstract (x NumPos NumNeg))" will specify that there are abstract variables to be defined. Then the variable 'x' will become associated with the domain elements 'NumPos' and 'NumNeg'. This allows developers to then use the variable 'x' in their programs logic, and it will be represented as an AbstractVal with a HashSet containing 'NumPos' and 'NumNeg'. Changes to the Reader.java file were made to handle the parsing of abstract variables.

Supported Domain Elements Include:

- TypeError
- UnsupportedFunctionError
- UnsupportedTypeError
- RuntimeError
- NumPos
- NumZero
- NumNeg
- BTrue
- BFalse

Arithmetic Operations and Abstract Values:

The extension of arithmetic operations in Funclang involves a selective interpretation of expressions containing abstract values. Not all expressions are subjected to abstract interpretation; rather, the extension focuses on specific cases where abstract values are present. The introduction of the AbstractValue class is pivotal for handling accommodations for abstract values in arithmetic expressions. Changes have been made to the Evaluator.java file's visit functions to extend arithmetic support for AbstractValues. Furthermore, the language distinguishes between concrete and abstract values by checking the instance type of Values when handling expressions.

Selective Abstract Interpretation:

In Funclang, the abstract interpretation is selective, meaning that it is applied only under specific conditions and is not universally applied to all expressions. The selective nature of this abstract interpreter allows developers to target and apply abstraction where it is most relevant, providing a nuanced approach to handling abstract values. The following clarifications outline the conditions under which abstract interpretation is triggered.

1. **Presence of Abstract Values:**
   Abstract interpretation is selectively activated when expressions involve variables of values associated with the abstract domain. This includes instances where variables are declared as abstract in the .scm file, specifying their association with elements from the predefined abstract domain. Expressions that contain abstract values, either operands or results, are candidates for abstract interpretation. This includes arithmetic operations (‘AddExp’, ‘SubExp’, ‘MultExp’, ‘DivExp’) and comparison operations (‘LessExp’, ‘EqualExp’, ‘GreaterExp’).
2. **Visit Function Modifications:**
   The selective nature of abstract interpretation is implemented through modifications to the visit functions within the ‘Evaluator’ class. Specifically, these modifications include checks for the presence of abstract values within expressions. If any operands, variable, or result within an expression is an instance of ‘AbstractVal’, the visit function incorporates the logic for abstract interpretation, otherwise the visit function executes normally.
3. **Parsing of Abstract Variables:**
   The parser in the ‘Reader.java’ file identifies and processes abstract variables declared in the .scm file. This parsing step contributes to the conditions under which abstract interpretation is selectively applied. Abstract variables, when utilized in subsequent expressions, may lead to the activation of abstract interpretation.

Syntax Examples with Abstract Value:

The following are some example .scm files to provide a simple understanding of how abstract interpretation can be initiated.

1. (abstract (x NumPos NumNeg))

   (if (= x -1) (+ x -1) (/ x 0))

2. (abstract (x NumNeg) (y NumPos))

```
(+ (/ 3 (+ x y)))
```

3. (abstract (x NumNeg) (y NumPos) (z BTrue))

```
(+ 2 z)
```

Impact on Existing Code:

The language remains fully functional as before, with the recent changes extending its capabilities to include selective abstract interpretation. Existing code will continue to work seamlessly unless it explicitly utilizes the new abstract features.The primary change involves the introduction of selective abstract interpretation triggered by the presence of (abstract) in the first line of the .scm file. Existing code that doesn't declare variables as abstract or involve abstract values will experience no impact. In essence, developers can choose to leverage the new abstract features, allowing them to enhance their code with selective abstract interpretation only where needed.

Error Handling and Messaging:

Errors and messaging are all handled through the elements of the abstract domain. Meaning that the result of abstract interpretation will yield a set that may contain errors such as the following:

1. TypeError:
    - Description: This error indicates a type mismatch or unexpected data type during abstract interpretation.
    - Context: It may occur when performing operations that involve abstract values, and the interpreter encounters an unexpected data type.
2. UnsupportedFunctionError:
    - Description: Signifies that an abstract operation or function is not supported or defined within the abstract domain.
    - Context: This error occurs when trying to perform an operation that goes beyond the capabilities of the abstract domain or when an unsupported function is applied.
3. UnsupportedTypeError:
    - Description: Indicates that an operation involving abstract values encounters an
4. RuntimeError:

- Description: Represents a generic runtime error during abstract interpretation.
- Context: This error is a catch-all for unexpected runtime issues that may arise during abstract interpretation. Is most likely a result of the potential possibility to divide by zero.