Snowlake Documentation

Release 0.0.1

William Li

Sep 10, 2020

Contents

Snowlake is both a declarative language of regular rules of inference and propositional logic for defining static type inference rules of programming languages, as well as a compiler-compiler that can synthesize such inference rule definitions into code used for static type checking, typically used for semantic analysis in language compilers.

The goals of Snowlake are to:

- Provide a flexible declarative language that is able to define the static type inference rules of most programming languages.
- Facilitate language developers in defining, documenting and sharing the set of type inference rules of any particular language.
- Alleviate the burden on language developers from implementing type checking logic that are usually extremely complex, tedious and error-prone.

CHAPTER 1

Contents:

1.1 Tutorial

Snowlake joins Flex and Bison (as well as their siblings Lex and Yacc) as a new member in the family of compiler-compilers in empowering programming language designers, authors, and engineers. As Flex and Bison focus on enabling the authoring of language lexers and parsers, Snowlake focuses on the next major step in language compiler construction: *static type checking as part of semantic analysis*.

Snowlake is a declarative language for expressing the static type inference rules expressed in programming language semantics. It is designed to be language agnostic, and its semantics and language constructs express rules of inference in propositional calculus, such as hypothetical syllogism. Therefore, prerequisite understanding of propositional logic is needed in mastering the Snowlake language, but prior experience in other imperative programming languages is not required.

Here we'd like to demonstrate the features, syntax and semantics of the Snowlake language by going through a brief example of defining the semantics and static type inference rules of a trivial reference language. We chose to intentionally omit the introduction of the syntax of this reference language, and solely focus on its semantics and static type inference rules, because the language syntax is irrelevant in this context. The various features of the Snowlake language are illustrated in detail as we progress in defining and expressing the inference rules of our reference language.

1.1.1 Inference groups

The top-level abstraction in the Snowlake language is **inference group**, which enables logical grouping of inference rules. Inference group definitions start with the keyword *group* followed by the name of the group. Each file can contain multiple inference group definitions, but their names must be unique.

Let us define one inference group to encapsulate all inference rules used for this exercise, and name it *SampleProject*. The definition will look like:

```
group SampleProject {
    ...
}
```

Each inference group definition translates directly into a corresponding C++ class with the same name. With the group definition above, the synthesized C++ class definition will resemble the following form:

```
class SampleProject {
    ...
};
```

Environment definitions

Each inference group definition contains a set of attributes that affect all the inference rules within the group. These attributes are denoted as key-value pairs and are specified within the group definition. Some keys are required, while others are optional. Below are descriptions of all supported environment definitions.

ClassName

The **ClassName** field specifies the name of the synthesized C++ class, if it needs to be different from the name of the encapsulating inference group definition.

The syntax of this field is:

ClassName : <value>;

TypeClass

The **TypeClass** field is a required attribute that specifies the class type in the synthesized C++ code for representing types in the target language.

It is critical to be aware that in Snowlake, all types in the target language are universally represented by a single type class in C++. One additional requirement for this type class is that it needs to be default constructible.

The syntax of this field is:

TypeClass : <value>;

ProofMethod

The **ProofMethod** field specifies the name of the C++ member function that can be used to infer types on identifiable entities. This is a user supplied function that needs to be a member of the synthesized C++ class.

The requirement on the signature of such function is that the return value is of type specified by the *TypeClass* field, and the parameters can be a single arbitrary type that fits of the context of the synthesized code.

The syntax of this field is:

ProofMethod : <value>;

TypeCmpMethod

Similar to the *ProofMethod* field described above, the **TypeCmpMethod** field specifies the name of the C++ member function that can be used to compare and evaluate equality among type instances. This is also a user supplied function that needs to be a member of the synthesized C++ class.

The requirement on the signature of such function is that the return value is of type *bool*, and the parameters are two instances of the type specified by the *TypeClass* field above, and an overloaded comparator functor in the *std* namespace. See *Equality premise* below for the supported comparison functor types.

The syntax of this field is:

TypeCmpMethod : <value>;

TypeAnnotationSetupMethod

The **TypeAnnotationSetupMethod** field is an optional attribute that specifies the name of the C++ member function that can be used to perform temporary type registration setup.

The signature of such function is that the return type be *void*, with a single parameter of type class specified by the *TypeClass* field above.

This field is used in conjunction with the *TypeAnnotationTeardownMethod* field to perform setup and teardown.

For more details on type registration setup and teardown, refer to *While-clause* section below.

The syntax of this field is:

TypeAnnotationSetupMethod : <value>;

TypeAnnotationTeardownMethod

The **TypeAnnotationTeardownMethod** field is an optional attribute that specifies the name of the C++ member function that can be used to perform temporary type registration teardown.

The signature of such function is that the return type be *void*, with a single parameter of type class specified by the *TypeClass* field above.

This field is used in conjunction with the *TypeAnnotationSetupMethod* field to perform type registration setup and teardown.

For more details on type registration setup and teardown, refer to *While-clause* section below.

The syntax of this field is:

TypeAnnotationTeardownMethod : <value>;

With the environment definitions described, let us specify the required field for our inference group definition.

Since we want to have the synthesized C++ class be named *SampleProjectTypeChecker*, and have the code live under *SampleProjectTypeChecker.h* and *SampleProjectTypeChecker.cpp*, let us define the following:

```
ClassName : SampleProjectTypeChecker;
```

We also know that we are going to be using a C++ type class called *TypeCls* for working with all type instances through our type inference logic, so we can define the following:

```
TypeClass : TypeCls;
```

Let's further assume that we are going to supply our own implementation of the type proof and type comparison member functions, and they are named *proveType* and *cmpType* respectively, we can then specify the following two key-value pairs:

```
ProofMethod   : proveType;
TypeCmpMethod : cmpType;
```

With that, our inference group definition now will look like the following:

```
group SampleProject {
    ClassName     : SampleProjectTypeChecker;
    TypeClass     : TypeCls;
    ProofMethod   : proveType;
    TypeCmpMethod : cmpType;
}
```

1.1.2 Inference rule definitions

Inference rule definitions are at the heart of the Snowlake language. Each inference rule definition uniquely captures the static type inference logic associated with one language construct. The Snowlake compiler synthesizes each inference definition into a corresponding C++ function, which is a member of the C++ class that is synthesized from the corresponding parent inference group.

Each inference rule definition is made up of four components: **global definitions**, **parameters**, **premises**, and **proposition**, as well as two entities that make up premise and proposition definitions: **identifiables** and **deduced targets**. Global definitions and parameters are input that the inference rules use for deriving type inferences. Premises are the logical rules that make up the assumptions of a particular inference. Finally, each inference definition consists one proposition definition that makes up the final inferred type of the rule.

Inference rule definitions start with the keyword *inference* followed by the name of the inference rule. For the purpose of this exercise, let us define a single inference rule used for inferring the return type of a static method dispatch in our reference language.

Let us call the inference rule *StaticMethodStaticDispatch*. Our inference rule definition will then look like the following:

```
inference StaticMethodStaticDispatch {
    ...
}
```

1.1.3 Global definitions

Global definitions refer to named entities in the synthesized C++ code that reference objects or other constructs at the global level. Global definitions are simply declared names that tell the Snowlake compiler that such definitions can be used throughout the inference rules in a semantically correct manner.

Global definitions are specified with the key *globals* and are a list of named constants, separated by commas.

Let's assume that in our reference language, there exists a constant that is used to represent the *self* class type in any given context, and this constant is called *SELF_TYPE*. In order for us to interact and make use of this constant in our inference rules later on, we have to declare it as a global constant inside our inference rule definition:

```
inference StaticMethodStaticDispatch {

    globals: [
        SELF_TYPE
    ]

}
```

1.1.4 Identifiables

Identifiables in Snowlake are identifiers that simply refer to entities or attributes of entities in the synthesized C++ code. Identifiables can be chained with the dot (i.e. '.') character to represent members on existing identifiables.

For example, we can have an identifiable named *StaticMethodCallStmt* that refer to a variable named *StaticMethodCallStmt* in C++, and *StaticMethodCallStmt.return_type* that refer to the return type of the expression.

1.1.5 Deduced targets

Deduced targets are declarations of the deduced types within an inference rule. Deduced targets are synthesized into C++ variable declarations and definitions, and thus can be used in subsequent premise definitions.

There are three form of deduced targets: **singular form**, **array form** (with and without size literal), and **computed form**.

Singular form

Deduced targets in singular form represent individual named types deduced in the inference rule.

Deduced targets in singular form are represented as individual names.

For example, we can use the following premise definition to denote the type inference for a static method dispatch's return type:

```
StaticMethodCallStmt.return_type : returnType;
```

Array form

Deduced targets in array form represent a collection of types deduced in the inference rule, and are synthesized into array/vector types in C++ depending on if a fixed size literal is used.

For example, we can use the following premise definition to denote the inferred types of a static method dispatch's argument list:

```
StaticMethodCallStmt.argument_types : ArgumentsTypes[];
```

Computed form

Deduced targets in computed form represent types deduced through calling a function. This form of deduced targets are used when the type deduction result is not bound at compile time, but rather at run time. This is important for many language constructs, such as class inheritance.

For example, we can use the following premise definition to denote the type inference for a static method dispatch's caller type:

```
StaticMethodCallStmt.caller_type : getBaseType();
```

1.1.6 Parameters

As mentioned above, each inference rule definition is synthesized into a corresponding C++ member function, thus it is a required step to define the parameters that get passed to the function, which in turn make up the missing part of

the function signature. Each parameter is made up of a name and its type, much like in C++. However, the difference lie in the syntax for expressing parameters in Snowlake.

Parameters are defined under the *arguments* key within an inference rule definition. Each parameter is defined with its name, followed by colon (i.e. *:*), and followed by its type in the final C++ code. Note that just like in C++, parameters in each inference rule definition must not contain duplicate names.

Back to the implemantation of our inference rule definition for static method dispatch. The synthesized C++ code needs to take an instance of an object type that represents the static method dispatch at a code level (i.e. an *ASTExpr* class). We can then incorporate the parameter list inside the inference rule definition as follows:

```
inference StaticMethodStaticDispatch {

    globals: [
        SELF_TYPE
    ]

    arguments: [
        StaticMethodCallStmt : ASTExpr
    ]

}
```

1.1.7 Premises

Premises are the building block of inference rule definitions that capture the logic of the inference, and are translated to actual C++ code within the body of the corresponding synthesized C++ function. Premises are categorized into two types: **inference premises** and **equality premises**.

Inference premise

Inference premises are logical rules that establish the assumption that an identifiable entity can be proven to be a particular type. This type of premise is essential and are used in the majority of inference rules. Inference premises have the following syntax:

<identifiable> : <deduced target>

Back to our example, we can use the following inference premise to denote the inferred type of a static method dispatch's return type:

```
StaticMethodCallStmt.return_type : returnType;
```

While-clause

Within the semantics of many programming languages, it is necessary to make temporary assumptions on the types of certain entities as part of other inferences. While-clauses are extensions to inference premise definitions that make expressing such assumptions possible. All premises specified under the body of a while-clause are translated as usual, and the premise definition that starts the while-clause becomes the assumption that gets temporarily set up and teared down before and after the inferences in the while-clause body.

To specify a while-clause, use the *while { ... }* following an inference premise definition.

For example, we can specify the following while-clause to operate under the assumption that the type of *StaticMethodCallStmt.caller_type* will infer to the global definition *CLS_TYPE*:

```
StaticMethodCallStmt.caller_type : CLS_TYPE while {
    ...
};
```

Equality premise

Equality premises are logical rules that establish the expected equality relations between inferred types. They are binary expressions that evaluate on two deduced types, along with an equality operator that denotes the equality relation. There are four types of equality relations:

Equality relation	Operator	Synthesized C++ comparison functor
Equal	=	std::equal_to<>
Not equal	!=	std::not_equal_to<>
Less than	<	std::less<>
Less or equal	<=	std::less_equal<>

Equality premise definitions have the following syntax:

<deduced target> <operator> <deduced target>;

For example, we can check that the static method dispatch's first argument is not equal to the self type of the method definition, with the following equality premise definition:

```
ArgumentsTypes[0] != SELF_TYPE;
```

Range-clause

Range-clause is an extension to equality premise definitions which enables comparison of set of type instances between deduced targets in array form.

To specify range-clause, use the *inrange* keyword after an equality premise definition, followed by the starting index used for the array form deduced targets on the left-hand-side and right-hand-side of the equality check respectively, and ends with the deduced target instance that forms the upper bound of the array check. All three values are separated by ....

For example, we can apply range-clause to check and make sure that the static method dispatch's argument types match the parameters of the function definition:

```
ArgumentsTypes[] <= ParameterTypes[] inrange 1..1..ParameterTypes[];
```

We can now incorporate all the necessary premise definitions into our inference rule definition to build up the inference logic required for static method dispatch type checking:

```
inference StaticMethodStaticDispatch {

    ...

    premises: [
        StaticMethodCallStmt.argument_types            : ArgumentsTypes[];
        StaticMethodCallStmt.callee.parameter_types    : ParameterTypes[];
```

(continues on next page)

(continued from previous page)

```
        ArgumentsTypes[] <= ParameterTypes[] inrange 0..1..ParameterTypes[];
        ArgumentsTypes[0] != SELF_TYPE;

        StaticMethodCallStmt.caller_type : CLS_TYPE while {
            ArgumentsTypes[] <= ParameterTypes[] inrange 1..1..ParameterTypes[];
        };

        StaticMethodCallStmt.caller_type                : getBaseType();
        StaticMethodCallStmt.return_type                : returnType;
    ]

}
```

1.1.8 Proposition

Each inference rule definition ends with a proposition definition that declares the inferred type of the inference. The syntax of propositions is as:

proposition: <*deduced target*>*;*

For example, we can specify the following proposition definition to denote the inferred type of a static method dispatch's return type:

```
proposition : baseType(returnType);
```

1.1.9 Error handling

The synthesized C++ code makes use of *std::error_code* and *std::error_category* constructs to handle errors throughout the inference deduction process. Therefore, the Snowlake compiler will also synthesize an extra *InferenceErrorDefn.h* and *InferenceErrorDefn.cpp* that contain the error definitions.

InferenceErrorDefn.h:

```
/**
 * Auto-generated by Snowlake compiler (version 0.1.1).
 */
#pragma once

enum InferenceError
{
    InferenceErrorInferredTypeMismatch = 0x01,
    InferenceErrorTypeComparisonFailed,
};

class InferenceErrorCategory;
extern const InferenceErrorCategory inference_error_category;
```

InferenceErrorDefn.cpp:

```
/**
 * Auto-generated by Snowlake compiler (version 0.1.1).
 */
#include "InferenceErrorDefn.h"
#include <string>
```

(continues on next page)

(continued from previous page)

```
#include <system_error>

class InferenceErrorCategory : public std::error_category
{
    virtual const char* name() const except override {
        return "Inference error";
    }

    virtual std::string message(int condition) const override {
        switch (condition) {
            case InferenceErrorInferredTypeMismatch:
                return "Inferred type does not match with expected.";
            case InferenceErrorTypeComparisonFailed:
                return "Type comparison failed.";
            default:
                return "Inference failed (unknown error).";
        }
    }
};

const InferenceErrorCategory inference_error_category {};
```

1.1.10 Put it all together

We can now put all the pieces together to form the entire inference definition under our inference group:

```
group SampleProject {

    ClassName     : SampleProjectTypeChecker;
    TypeClass     : TypeCls;
    ProofMethod   : proveType;
    TypeCmpMethod : cmpType;

    inference StaticMethodStaticDispatch {

        globals: [
            SELF_TYPE
        ]

        arguments: [
            StaticMethodCallStmt : ASTExpr
        ]

        premises: [
            StaticMethodCallStmt.argument_types            : ArgumentsTypes[];
            StaticMethodCallStmt.callee.parameter_types    : ParameterTypes[];

            ArgumentsTypes[] <= ParameterTypes[] inrange 0..1..ParameterTypes[];
            ArgumentsTypes[0] != SELF_TYPE;

            StaticMethodCallStmt.caller_type : CLS_TYPE while {
                ArgumentsTypes[] <= ParameterTypes[] inrange 1..1..ParameterTypes[];
            };

            StaticMethodCallStmt.caller_type               : getBaseType();
```

(continues on next page)

(continued from previous page)

```
            StaticMethodCallStmt.return_type                : returnType;
        ]

        proposition : baseType(returnType);
    }
}
```

Assume we save the entire inference definition group in a file called *SampleProject.sl*, we can compile the definition by invoking the following command on the Snowlake compiler:

```
snowlakec --errors --verbose --output ./ SampleProject.sl
```

This will synthesize the C++ output into *SampleProjectTypeChecker.h* and *SampleProjectTypeChecker.cpp*. For the curious bunch, below is the synthesized C++ output.

SampleProjectTypeChecker.h:

```
/**
 * Auto-generated by Snowlake compiler (version 0.1.1).
 */

#pragma once

#include <cstdlib>
#include <cstddef>
#include <vector>
#include <system_error>

class SampleProjectTypeChecker
{
public:
    TypeCls MethodStaticDispatch(const ASTExpr& StaticMethodCallStmt, std::error_
↪code*);
};
```

SampleProjectTypeChecker.cpp:

```
/**
 * Auto-generated by Snowlake compiler (version 0.1.1).
 */
#include "SampleProjectTypeChecker.h"
#include "InferenceErrorDefn.h"

TypeCls
SampleProjectTypeChecker::MethodStaticDispatch(const ASTExpr& StaticMethodCallStmt,␣
↪std::error_code* err)
{
    std::vector<TypeCls> ArgumentsTypes = proveType(StaticMethodCallStmt.argument_
↪types);
    std::vector<TypeCls> ParameterTypes = proveType(StaticMethodCallStmt.callee.
↪parameter_types);
    for (size_t i = 0, size_t j = 1; i < ParameterTypes.size(); ++i, ++j) {
        if (!cmpType(ArgumentsTypes[i], ParameterTypes[j], std::less_equal<TypeCls>
↪())) {
            *err = std::error_code(0, inference_error_category);
            return TypeCls();
        }
```

(continues on next page)

(continued from previous page)

```
    }

    if (!cmpType(ArgumentsTypes, SELF_TYPE, std::not_equal_to<TypeCls>())) {
        *err = std::error_code(0, inference_error_category);
        return TypeCls();
    }

    // Type annotation setup.
    typeAnnotationSetup(StaticMethodCallStmt.caller_type, CLS_TYPE);

    for (size_t i = 1, size_t j = 1; i < ParameterTypes.size(); ++i, ++j) {
        if (!cmpType(ArgumentsTypes[i], ParameterTypes[j], std::less_equal<TypeCls>
↪())) {
            *err = std::error_code(0, inference_error_category);
            return TypeCls();
        }
    }

    // Type annotation teardown.
    typeAnnotationTeardown(StaticMethodCallStmt.caller_type, CLS_TYPE);

    TypeCls var0 = getBaseType();
    TypeCls var1 = proveType(StaticMethodCallStmt.caller_type);
    if (!cmpType(var0, var1, std::equal_to<>())) {
        *err = std::error_code(0, inference_error_category);
        return TypeCls();
    }

    TypeCls returnType = proveType(StaticMethodCallStmt.return_type);
    return baseType(returnType);
}
```

1.1.11 Invoking Snowlake compiler

Once the Snowlake project is built, invoking the Snowlake compiler, namely *snowlakec*, is fairly trivial. Below is the command-line interface:

```
snowlakec (version 0.1.1)

Snowlake compiler.

Snowlake is both a declarative language of regular rules of inference
and propositional logic for defining static type inference rules of
programming languages, as well as a compiler-compiler that can
synthesize such inference rule definitions into code used for
static type checking, typically used for semantic analysis
in language compilers.

Usage: snowlakec [OPTION]... INPUT


OPTIONS:

-v, --verbose
      Verbose mode.
```

(continues on next page)

(continued from previous page)

```
      Optional. Default value: 0
-d, --debug
      Debug mode.
      Optional. Default value: 0
-s, --silent
      Silent mode.
      Optional. Default value: 0
-b, --bail
      Bail on first error.
      Optional. Default value: 0
-e, --errors
      Treat warnings as errors.
      Optional. Default value: 0
-o, --output <value>
      Output path.
```

All options are fairly self-explanatory. The argument to *–output* needs to be a directory path in which multiple .h and .cpp files can be saved at.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search