C ♭ Type Representation

Overview

The type checking phase of the compiler traverses the `AST` and annotates each node with a representation of the datatype of the value or structure represented by the subtree rooted at that node.

Cb type representations are all instances of the `CbType` class or its subclasses.

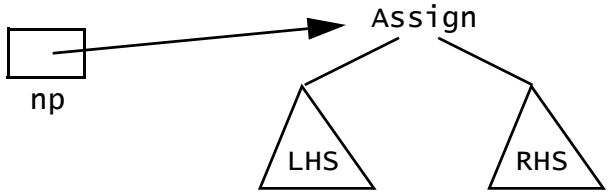
This document provides a brief description of the various type representations.

Basic Types

The `int`, `bool`, `string` and `void` types are all represented by instances of the `CbBasic` class. An *enum* field inside the instance distinguishes these basic types.

So that type checking is simplified, we want each of those four basic types to be represented by *unique* instances of the `CbBasic` class. Those unique instances are obtained by accessing the static class properties `CbType.Int`, `CbType.Bool`, `CbType.String` and `CbType.Void` respectively.

When unique type instances are used, typechecking is simplified. For example, a subtree for an assignment statement referenced by variable `np` has the structure



and, after the LHS and RHS subtrees have been traversed to label their nodes with type information, we can check the assignment by comparing two type references:

```
if (np[0].Type != np[1].Type)
    Console.WriteLine("Line {0}: Type mismatch!", np.LineNumber);
```

where `np[i]` obtains a reference to the child with inde number i.

Array Types

An array type, such as `int[]`, is represented by an instance of the `CbArray` class. As with basic types, we want these instances to be unique. For example, if two separate declarations in the CFlat program declare variables as having type `int[]`, then we do not want two instances of the `CbArray` class to be created. The static factory method `CbType.Array(`*element-type*`)` will return a unique instance of `CbArray` for each element type.

Struct Types

A *struct* type is represented by an instance of `CbStruct`. Note that these instances are certain to be unique for each struct type because it would be an error to declare two structs with the same name inside the single class which comprises the program. The class constructor creates a description of the

struct where the fields are missing. The `AddField` method can be used to add details of each field, one by one. Later, when typechcking, if `typ` is an instance of `CbStruct`, then code like the following can be used to look up details of a field whose name is held in `fnam`:

```
Field f = null;
if (typ.Fields.TryGetValue(fnam, out f)) {
    ... // use f
} else
    ... // field not found
```

Method Signatures

A method's signature is considered to be a datatype. A method's signature is represented by an instance of the `CbMethod` class. The class constructor creates an instance where the types of the arguments are missing. If `typ` is an instance of `CbMethod` then code like the following

```
typ.ArgType.Add(argt)
```

may be used to add `argt` as the datatype of the next method argument. (Names of the arguments do not form part of the signature). Later, during typechecking, the expressions `typ.ArgType.Count` and `typ.ArgType[i]` may be used to retrieve the number of arguments and the type of the *i*-th argument, respectively.

Handling null – The Null Type

In a CFlat program, `null` can be used to initialize any array variable. For example:

```
int[] list;
Point[] route;  // Point is a struct type
list = null;
route = null;
```

This illustrates the polymorphic nature of `null`. (It is even more polymorphic in C#.) It has to be handled as a special case. The datatype of `null` should be set to the special datatype `Null`, and the typechecking rules should allow `Null` to be assignable or comparable for equality with any array type.

Handling Errors – The Error Type

The type checking phase of the compiler should expect to find errors in the CFlat program. These errors include undeclared identifiers and operators applied to operands with incompatible types.

When an undeclared identifier is encountered, there should ideally be exactly one error message generated and type checking should continue as best it can. To help meet that goal, the recommended strategy is to add that identifier to the symbol table (so that further uses of the identifier will not generate more error messages) and to give it the special datatype `Error`. The typechecking rules should be extended so that whenever an operand has type `Error`, no type mismatch error messages are generated.

If an invalid expression such as `"abc"%99` is encountered, the error message is generated and then the result type of the expression should be set to either `Error` or to a basic type like `int`. (Choose `int` if you think that the operator '%', in this case, must always yield an `int` result, choose `Error` if

it is unclear what the datatype should be.) The goal is again to continue to typecheck the code as best you can, while not producing a cascade of additional error messages.

There are no perfect strategies. Your approach will not always be the best one for a particular situation.