Generic Name Resolution with Metamodels

Yuriy Tymchuk*, Benjamin Arezki†, Anne Etien†, Nicolas Anquetil† and Stéphane Ducasse†
* REVEAL - Faculty of Informatics, University of Lugano (USI)
Via G. Buffi 13, CH-6904 Lugano, Switzerland
Email: yuriy.tymchuk@usi.ch
† Inria Lille-Nord Europe - University Lille 1, LIFL CNRS UMR 8022,
40 Avenue Halley, Park Plaza, 59650 Villeneuve d'Ascq, France
Email: *firstname.lastname*@inria.fr

Nowadays almost all modern IDE provides editor services such as reference resolving while writing code to identify where a given symbol has been defined, constraints checking to highlight duplicate definitions, use before definition, unresolved reference…or code completion by proposing valid identifier in a given context. In parallel, in maintenance, lot of metrics or program analysis approaches rely on the identification of dependencies between concepts. All of these services rely on name resolution also known as symbol resolution.

Providing such services for programs written in a language not currently supported requires as a first essential step to develop a name resolution algorithm for this language. Dealing with all the specific cases of the language may put a damper on the developer's enthusiast that will choose tools already providing name resolution algorithm for the given language and plug the new services to this tool. Such a solution has the major advantage to avoid writing the name resolution algorithm for the new language. Nevertheless, it has the major drawback for the new service to be dependent to another tool.

Name resolution amounts to linking a name (an identifier) in the source code to an entity of the program: in the expression `i++`, the symbol `i` refers to a variable of the program that must be incremented by one. The basic rule for name resolution in lexical scoping is to look for the entity in the current scope, e.g. a variable name will be first searched in the scope of the function within which it appears. The entity must first match the identifier, that is to say have the right name, second it must match the kind of entity (variable, function, method, class) if it is known, third in some languages the type of the name must also matched (for example in Java methods are matched on their signature). If a matching entity is not found in the current scope, one searches recursively in the containing scope.

But this generic algorithm has many variations according to the programming language. Here are the specificities of some languages:

- In C, there is a global scope for the entire program and a local scope for each function. Functions can only be defined at the global scope, variables and types can be defined globally or locally.
- Pascal has the same global and local scopes as C, but all scopes can include functions (or procedures), types, and variables.
  Pascal also has the `with` instruction that creates a temporary scope for a given structured type.
- In OO languages, on top of lexical inclusion of scopes (method scope included in the scope of its class), inheritance also defines an inclusion of scopes: the scope of a subclass is included in the scope of its superclass.
  OO languages also assume two implicit variables, `this` (or `self`) and `super`, that are never defined but accessible within the scope of a class.

One can assume that every individual programming language will have a set of specific constraints or rules that affects how name resolution works. It is clear from this list that a truly generic name resolution mechanism cannot be defined. There are too many rules depending on the language. Nevertheless, some characteristics are common to every languages. Our generic resolution algorithm exploits these common points while letting some parts to adapt. Before detailing the algorithm, we present the AST metamodel that enables a unified representation of programs written in various languages and serve as base to our approach. Such AST Metamodel is an effort similar to Famix [DAB^{+}11], the Dagsthul Middle Metamodel [LTP04] or ASTM [AST11] for structural source code representation but at the AST level.

ASTM defines a core set of modeling elements that are common to many programming languages; it is the union of concepts from almost all the languages. It considers object-oriented programming languages with concepts such as ClassType, ExceptionType or AccessKind. It has also concepts specific to procedural programming languages such as JumpStatement or Pointer. In total ASTM of the OMG defines 188 concepts. With so many concepts, the metamodel is hard to understand. Moreover, it is difficult to develop an algorithm adapted to all languages because of the conceptual difficulty of dealing with so many different concepts with definitions sometimes unclear. For example, although Java and Pharo, both have packages, classes and methods, the rules for scoping are different. This implies that having all the concepts in the metamodel does not prevent us from having to specialize them for each language.

Our metamodel, FAST, is defined as the intersection of all programming languages. By doing this, we have a metamodel with less than 20 concepts that can still accommodate the same large spectrum of programming languages while being much easier to apprehend and extend.

It starts with an abstract concept FASTEntity serving as the root class of the FAST metamodel. A FASTEntity may have a scope FASTScope or not. Four types of entities are distinguished:

- A FASTBehaviouralEntity is an abstract concept for all entities having a behavior like methods or functions.

Such entities may be named (in most cases) or not (*e.g.,* lambda-functions).

- A FASTStatement is also an abstract concept. We chose to be conservative and did not include any specific statement in the core FAST since they do not appear in all languages.
  Statements can be FASTStatementBlock, for example to represent the body of a function.
- A FASTExpression is an abstract concept that has a value. Again it would be difficult to try to be too specific here as even arithmetic expressions can be treated in different ways by different languages (*e.g.,* Pharo, Lisp). We believe some literals (FASTLiteral) are truly generic and included them.
- A FASTNamedEntity represents an identifier.

All these concepts are very generic and exist in any language. They don't capture the specificity of any language or even any paradigm (procedural, object, list...). Our name resolution algorithm relies only on these concepts making it generic.

The generic algorithm is decomposed in two parts: lookup and selection as illustrated in Figure 1.

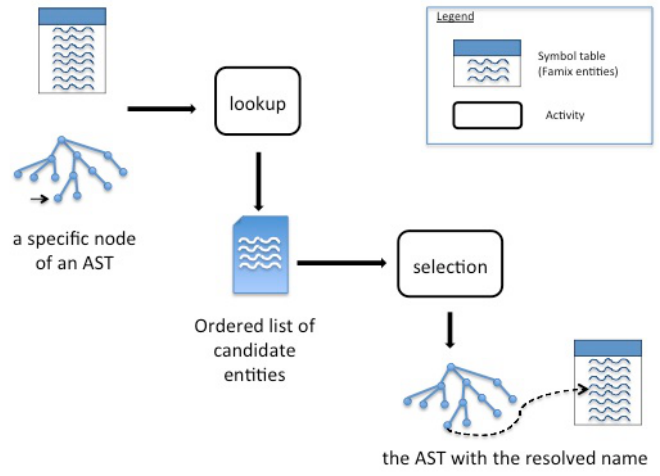
Legend
Symbol table (Famix entities)
Activity
lookup
a specific node of an AST
Ordered list of candidate entities
selection
the AST with the resolved name

Figure 1. Sketch of the algorithm

The input of the name resolution algorithm is the name of a given AST node. In details, the *lookup* searches entities with this name and specified in the current scope of this AST node. Then it recursively search candidate entities in the containing scopes of the node where the name appears. Candidates are returned in order of proximity; candidates in the immediate scope appear before candidates from a parent scope. This rule is one of the foundations of lexical scoping and is therefore generic. The output of this first part is an ordered list of candidate entities.

Candidates are matched on their name, and the kind of the entity (a function, a variable, etc.) Mapping of the name is very simple. Mapping of the entity kind is delegated to the AST node that contains the name, *i.e.,* a `FASTFunction` node for a Pascal program would only accept `FAMIXFunction` entities as matches. In this sense, this part of the algorithm is as generic as it can be, by delegating a part of the work to the AST nodes which are created by the parser for the language.

Given an ordered list of candidate entities, the *selection* algorithm will return the first candidate that matches the name resolution rules of the language.

This part is more dependent of the language but may be made generic by a kind of double-dispatching similar to what we did for the matching of the kind of entity in the lookup.

Each element of the input ordered list of candidate entities is successively studied. By construction, it matches the searched name and the expected kind. We now check if the (FAMIX) entity corresponding to the AST node input of the name resolution algorithm may access to this candidate entity or not. Each non accessible entity is eliminated from the list of candidates. The first entity satisfying this accessibility condition is the searched entity; it corresponds to the output of the name resolution algorithm.

To check the accessibility of the entities, several rules depending of the used language have been implemented. Mostly, these rules correspond to the visibility rules associated to the access modifiers. Thus, the access modifier of the studied candidate is analyzed and the corresponding accessibility rules are checked through the call of its implementation.

We have adapted FAST and the generic algorithm for name resolution to three different languages: Cobol, Pharo, and Java by following the following process:

- Specializing the FAST core metamodel by adding elements necessary to represent the concepts of this language;
- Writing a parser for the language and generating the AST;
- Deciding which FAST elements specific to the language have a scope/namespace, and;
- Choosing (eventually by implementing it) the identification strategy *i.e.,* the accessibility rules.

References

[AST11] *Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM)- Version 1.0.* Object Management Group, January 2011.

[DAB+11] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an interexchange format and source code model family. Technical report, RMod – INRIA Lille-Nord Europe, 2011.

[LTP04] Timothy Lethbridge, Sander Tichelaar, and Erhard Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. In *Electronic Notes in Theoretical Computer Science*, volume 94, pages 7–18, 2004.