A Hybrid Approach to Software Repository Retrieval: Blending Faceted Classification and Type Signatures

D. Eichmann

Dept. of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
email: eichmann@a.cs.wvu.wvnet.edu

Abstract

We present a user interface for a software reuse repository that relies both on the informal semantics of faceted classification and the formal semantics of type signatures for abstract data types. The result is an interface providing both structural and qualitative feedback to a software reuser.

1. Introduction

The importance of software reusability as a subdiscipline of software engineering is easily demonstrated by recent publications in the area, including a substantial two–volume collection edited by Biggerstaff and Perlis [2,3], as well as the earlier collection edited by Tracz [18].

Our current research focuses on composition–based reuse rather than generation–based reuse [4], since we feel that this is an area that promises the best short term results. As repositories increase in size and the components contained within them increase in complexity, increasing demands are placed upon the reuser, and thereby the retrieval mechanism, to discriminate between large numbers of candidate components. This paper discusses one such retrieval mechanism.

2. Background

This work draws from three areas of previous work: faceted classification, algebraic specification, and type inference.

2.1. Faceted Classification

Faceted classification was first proposed as a retrieval mechanism by Prieto–Diaz [14], and subsequently used in at least two repository efforts [1, 10]. The technique is founded upon the notion of literary warrant.

2.1.1. Literary Warrant

Literary warrant is a technique used in library science for the classification of texts [19]. Representative samples of works generate a set of descriptive terms subsequently organized for use in clustering the set of works as a whole.

2.1.2. Conceptual Closeness

The vocabulary of terms built up through literary warrant typically contains a great deal of semantic overlap, words whose meanings are the same – or at least similar. For instance, two components, one implementing a stack and the other a queue might both be characterized with the word insert corresponding to push and enqueue, respectively.

Synonym ambiguity is commonly resolved through the construction of a restricted vocabulary, tightly controlled by the repository administrators. Repository users must learn this restricted vocabulary, or rely upon the assistance of consultants already familiar with it. It is rarely the case, however, that the choice is between two synonyms. More typically it is between words which have similar, but distinct meanings.

The words in the two pairs (insert, push) and (insert, enqueue) are conceptually close, that is, they both are plausible characterizations of one of the operations for each of their respective components, and yet they have distinct definitions in normal English usage. This further leads to the notion that the word pair (push, enqueue) should similarly be conceptually close, if only transitively through the common word insert.

Attaching a weight from the interval (0,1) supports a closeness metric for word pairs, and additionally supports transitive weights as the product of the weights involved. For example, we might associate a weight of .8 to the pair (insert, push) and .9 to the pair (insert, enqueue), and thus a weight of .8 * .9 = .72 to the pair (push, enqueue).

Note that transitive closeness of conceptually close pairs results in a conceptually close pair, and transitive closeness of distant pairs results in an even more distant pair. Thus, the choice of the weights is critically impor-

Third International Conference on Software Engineering and Knowledge Engineering, Skokie, IL, June 27-29, 1991, pages 236-240.

tant to the success and utility of a user interface incorporating conceptual closeness.

2.1.3. Lattice-Based Faceted Classification

Eichmann and Atkins [6] described an approach to faceted classification that focused upon a structural framework (type lattices) as an alternative to explicit closeness weights. Each component possessed one or more tuples characterizing it, each comprised of a non-empty set of facet values. Users posed queries as tuples, and reuse candidates were retrieved based upon their conformance to the query tuple.

2.2. Type Signatures

An algebraic specification contains both a syntactic characterization of a component (the signature) and a semantic characterization of a component (the axioms). Algebraic specifications therefore are aptly suited as formal descriptions of software components.

Traditional efforts in reuse concentrated on the structural interfaces between components [1, 2], and hence solely on the signature portion of the specification. This proved less than adequate for component discrimination, in the face of numerous candidate components, all with the same interface, and directly prompted the work in faceted classification described above.

2.3. Type Inference

Recent research in programming language has resulted in a number of languages that are strongly typed, and yet, are flexible and remarkable expressive, (e.g., ML [13]). Such languages rely heavily on inferential mechanisms to ensure safe computation [5, 12]. The concept of conformance is particularly relevant to software repository query mechanisms [11]. Conformance allows one type instance to be treated as if it were an instance of another type, and can hold for arbitrary types, regardless of the type ordering scheme (e.g., inheritance).

Type inference notation organizes around a set of inference rules, comprised of sets of premises and conclusions, separated by a horizontal line. The symbol A represents an existing set of assumptions. A always contains the type information generated by the database schema implementing the repository. A.x denotes the set of assumptions extended with some fact x. $A \vdash x$ states that given a set of assumptions A, and the currently defined set of inference rules, x can be inferred. An expression is well-typed if a type for the expression can be deduced using the available inference rules, otherwise it is ill-typed.

3. A Hybrid Approach

The approach advocated here combines the semantic flexibility of faceted classification with the structural formality of type signatures. We accomplish this through the incorporation of function and abstract data type (ADT) definitions into the type lattice of [6].

3.1. The Type Lattice

As shown in figure 1, there are four principle sublattices comprising the complete type lattice, corresponding to the types generated by facet sets, tuples, functions and ADTs. In addition, the universal type, $\top$, and the void type, $\bot$, ensure that a least upper bound and a greatest lower bound, respectively, exist for any two types in the lattice. The usual built-in types (e.g., integers, strings, etc.) are not shown, in order to simplify the presentation. In principle, they can be specified as ADTs if needed.

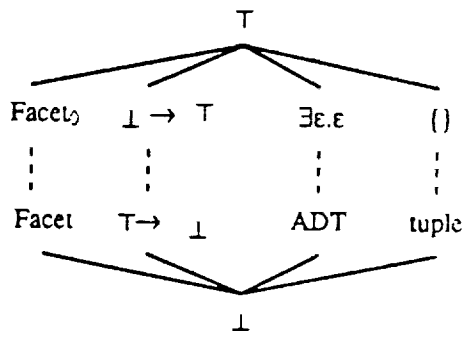

Figure 1.

$Facet_0$ characterizes the empty generic facet type; it contains no values, but is still a facet. Likewise, Facet characterizes the set of all possible facet values. The dotted line indicates an arbitrary number of intermediate types.

The tuple sublattice has a similar structure. At the top is the empty tuple type, {}, characterizing a type with no components. At the bottom is Tuple, the tuple type with all possible components.

Function types are bounded above by $\bot \rightarrow \top$, the function type with a void domain and universal range, and are bounded below by $\top \rightarrow \bot$, the function type with a universal domain and void range.

ADT types are bounded above by $\exists\varepsilon.\varepsilon$, the abstract type denoting a a hidden type, ε, with no information or operations available, and are bounded below by ADT, the type denoting all possible types with all possible operations.

3.2. Inference Rules

3.3. Facets

As in [6], we characterize facets as the inverse of our usual notion of interval subtypes; a facet subtype denotes a larger collection of facet values than does its supertype. Inference rule (1) formalizes this for a complete facet.

$$\frac{A \vdash m \in t \quad A \vdash n \in t \quad A \vdash m \le n}{A \vdash t \lesssim t_{\langle m \ldots n \rangle}} \tag{1}$$

Inference rule (2) does likewise for two singleton intervals, and inference rule (3) for two arbitrary collections of intervals.

$$\frac{A \vdash m \in t \quad A \vdash m' \in t \quad A \vdash n \in t \quad A \vdash n' \in t \quad A \vdash m' \le m \le n \le n'}{A \vdash t_{\langle m' \ldots n' \rangle} \lesssim t_{\langle m \ldots n \rangle}} \tag{2}$$

$$\frac{A \vdash t_{\langle m_1 \ldots n_1 \rangle} \lesssim t_{\langle m_1' \ldots n_1' \rangle} \quad \vdots \quad A \vdash t_{\langle m_i \ldots n_i \rangle} \lesssim t_{\langle m_i' \ldots n_i' \rangle}}{A \vdash t_{\langle m_1 \ldots n_1, \ldots, m_i \ldots n_i \rangle} \lesssim t_{\langle m_1' \ldots n_1', \ldots, m_i' \ldots n_i' \rangle}} \tag{3}$$

A number of inference rules no presented here address the reduction and manipulation of intervals [6].

3.3.1. Tuples

We view a tuple r to be of type record, $\{a_1 : t_1, \ldots, a_n : t_n\}$, where attribute a_i is of type t_i. We assume that t_i is some facet, function, or ADT type. Since attributes are labeled, components may appear in any order, and two types are assumed to be equivalent if they only differ in the order of their respective attributes.

Inference rule (4) characterizes subtyping for tuples. Informally, one tuple type is a subtype of another if it has all of the attributes of the other (and possible more), and for those common attributes, the type of a given attribute in the tuple subtype must be a subtype of that attribute's type in the tuple supertype.

$$\frac{A \vdash l \le m \le n \quad A \vdash t'_1 \lesssim t_1 \quad \vdots \quad A \vdash t'_m \lesssim t_m}{A \vdash \{i_1 : t'_1, \ldots, i_m : t'_m, \ldots, i_n : t_n\} \lesssim \{i : t_1, \ldots, i_m : t_m\}} \tag{4}$$

Inference rules (5) and (6) support definition of tuple constants and extraction of an attribute value, respectively.

$$\frac{A \vdash e_1 = t_1 \quad \vdots \quad A \vdash e_n = t_n}{A.(r = \{i_1 = e_1, \ldots, i_n = e_n\}) \vdash r : \{i_1 : t_1, \ldots, i_n : t_n\}} \tag{5}$$

$$\frac{A \vdash r : \{i_1 : t_1 \ldots i_n : t_n\} \quad A \vdash 1 \le j \le n}{A \vdash r.i_j : t_j} \tag{6}$$

3.3.2. Functions

Function types are useful both for characterizing programs and for characterizing the operations of ADTs. Inference rule (7) characterizes the usual notion of lambda abstraction, where x is the parameter, t' the parameter's type, e is the body of the function, and t the type of the function's result.

$$\frac{A, x : t' \vdash e : t}{A \vdash \lambda(x : t')\ e\ : (t' \to t)} \tag{7}$$

One function type, $s \to t$, is a subtype of another, $s' \to t'$, if the subtype function accepts the entire domain of the function supertype (i.e., $s' \lesssim s$), and produces a range contained in the supertype range (i.e., $t \lesssim t'$), as shown in inference rule (8).

$$\frac{A \vdash s' \lesssim s \quad A \vdash t \lesssim t'}{A \vdash s \to t \lesssim s' \to t'} \tag{8}$$

Function subtyping seems a little strange at first, but a simple example helps. Assume that f is a function type (1..4) → true and g is a function type (2..3) → (true..false). Function type f is a subtype of g. Any instance of f can always replace an instance of g in an expression without effecting the type–safety of the expression. The instance of f handles at least the values the supertype function does, and produces no more values than does the supertype function.

Inference rule (9) characterizes the type of the result of a function application; if the expression supplied as an ar-

gument is of the proper type, then the result of the function applied to that expression will be well-typed.

$$\frac{A \vdash e : (t' \rightarrow t) \quad A \vdash e' : t'}{A \vdash e(e') : t} \qquad (9)$$

3.3.3. ADTs

Inference rules (10) and (11) define type inference for existential types [4]. An existential type consists of a type variable a, representing the type, packaged with some number ($j_1 \ldots j_n$) of instances of the type and/or operations over the type.

$$\frac{\begin{array}{c} A \vdash e_1 : s_1|_{t/a} \\ \vdots \\ A \vdash e_n : s_n|_{t/a} \end{array}}{\begin{array}{l} A \vdash \text{pack } (a = t \text{ in } (j_1 : s_1, \ldots, j_n : s_n)) \\ \qquad (e_1, \ldots, e_n) : \exists a.(j_1 : s_1, \ldots, j_n : s_n) \end{array}} \qquad (10)$$

$$\frac{A \vdash e : \exists b.(j_1 : s_1, \ldots, j_n : s_n) \quad A.(x : (j_1 : s_1, \ldots, j_n : s_n))|_{a/b} \vdash e' : t}{A \vdash \text{open } e \text{ as } x[a] \text{ in } e' : t} \qquad (11)$$

A given expression e_i is of type s_i when t is substituted for a in s_i, and serves as the implementation of the value or operation labeled j_i in the abstract type. This substitution results in a concrete type (i.e., one with no type variables in it) for the expression. The substitution type t serves as the representation of the abstract type, denoted externally by the existential variable a. The actual representation and the implementations of the operations are not visible externally.

The pack operation constructs an instance of an abstract type, and encapsulates its representation. The open operation performs the converse, binding an abstract type variable to a concrete type, and evaluating some expression in the context of the (now concrete) abstract type.

Subtyping of ADTs derives from subtyping of the type parameters for the abstract type. Inference rule (12) characterizes subtyping of two instances of abstract types.

$$\frac{A.(t_1 \leq t_2) \vdash (t \leq t')}{A \vdash (\exists(t_1 \leq t_2).t) \leq (\exists(t_1 \leq t_2).t')} \qquad (12)$$

Note that in addition to providing subtyping of two ADTs, rule (12) also supports subtyping of two instances of the same ADT.

For an example of the former, $\exists T' \exists(T \leq T').T''$ denotes an existential type T'' generated by a type parameter T, which must be a subtype of the existential type T'. Since instances of abstract types are cross products of instances and operations, T would be a subtype of T' through additional operations. An example of this appeared in [17], showing stacks and dequeues as subtypes of queues.

For an example of the latter, stack of $\text{integer}_{(1..10)}$ is a subtype of stack of integer.

4. The User Interface

A query is a boolean expression containing predicates and the operators *and*, *or*, and *not*. A predicate is simply a constant of type tuple. When a user issues a query, the query evaluator first treats all of the facet values in the query as synonyms and replaces them with actual facet values from a value/synonym relation. For example, *database*, *databases*, *data base*, and *data bases* might all be replaced with database.

The evaluator then locates all of the relations in the database whose type conforms to some predicate of the query by testing the type of each relation in turn, using the inference rules previously described. The query lattice space for a given predicate is bounded above by the predicate type itself, and bounded below by the partition tuples that conform to it. For each user-specified predicate, the evaluator forms the disjunction of conforming relation tuples (with variables in each position) and then substitutes the conjunction of the disjunction and the new predicate in place of the original, user-specified predicate. The result of evaluating this query is then a set of component references for display and optionally, retrieval from the text storage area.

Note that since tuples of more than a single type may be displayed to the user, the query language is polymorphic in one of the manners discussed in [7].

5. Discussion

The work described here is another in a series of experimental user interfaces for software reuse repositories. Our initial efforts concentrated specifically on providing substructure for faceted classification [9]. This approach relied only upon the expertise of the classifier in populating the repository, and as such, suffered from what we refer to as the *vocabulary problem*.

The interface described here ameliorates the situation by supporting as part of the query tuple the specification of a formal interface structure to which the components of interest must conform.

A parallel effort exploring the role that algebraic specification can play in repository retrieval appears in [8]. This work is concerned particularly with retrieval over type signatures and behavioral axioms.

6. References

[1] J. Atkins, private communication, 1989.

[2] T. J. Biggerstaff and A. J. Perlis, *Software Reusability, vol. 1 – Concepts and Models*, Addison–Wesley, New York, NY, 1989.

[3] T. J. Biggerstaff and A. J. Perlis, *Software Reusability, vol. 2 – Applications and Experience*, Addison–Wesley, New York, NY, 1989.

[4] T. J. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, vol. 4, no. 2, pages 41–49, March, 1987.

[5] L. Cardelli, "Basic Polymorphic Typechecking," *Science of Computer Programming*, vol. 8, pages 147–172, 1987.

[6] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pages 471–522, December 1985.

[7] D. Eichmann, *Polymorphic Extensions to the Relational Model*, Ph.D. dissertation, Dept. of Computer Science, The University of Iowa, Iowa City, IA, August 1989.

[8] D. Eichmann, "Selecting Reusable Components Using Algebraic Specifications," *Second International Conference on Algebraic Methodology and Software Technology (AMAST)*, Iowa City, IA, May 22–25, 1991.

[9] D. Eichmann and J. Atkins, "Design of a Lattice–Based Faceted Classification System," *Second International Conference on Software Engineering and Knowledge Engineering*, Skokie, IL, pages 90–97, June 21–23, 1990.

[10] E. Guerrieri, "On Classification Schemes and Reusability Measurements for Reusable Software Components," SofTech Technical Report IP–256, SofTech, Inc, Waltham, MA 1987.

[11] C. Horn, "Conformance, Genericity, Inheritance and Enhancement," *ECOOP–87 – Proc. European Conference on Object–Oriented Programming*, Paris, France, pages 223–233, June 15–17, 1987.

[12] R. Milner, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences*, vol. 17, pages 348–375, 1978.

[13] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1990.

[14] R. Prieto–Diaz, *A Software Classification Scheme*, Ph.D. dissertation, Dept. of Information and Computer Science, University of California, Irvine, CA, 1985.

[15] R. Prieto–Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, vol. 4, no. 1, pages 6–16, January, 1987.

[16] J. V. Guttag and J. J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, vol. 10, pages 27–52, 1978.

[17] A. Snyder, "Inheritance in the Development of Encapsulated Software Components," *Research Directions in Object–Oriented Programming*, B. Shriver and P. Wegner, eds., MIT Press, Cambridge, MA, pages 165–188, 1987.

[18] W. Tracz, ed., *Tutorial, Software Reuse: Emerging Technology*, IEEE Computer Society Press, Los Angeles, CA, 1988.

[19] B. C. Vickery, *Faceted Classification: A Guide to Construction and Use of Special Schemes*, Aslib, London, 1960.