Generics in Jif

Matthew Loring
Advisor: Andrew Myers
Cornell University

1 Problem and Motivation

Generics and type parameterization are key features in most programming languages. They allow programmers to decouple the data structures and algorithms from the types they operate on and enable powerful software design patterns. Languages such as Java support generic programming using type parameters explicitly declared in a class or method declaration that may be instantiated on concrete types at runtime.

Security typed languages such as Flowcaml [2] and Jif (Java + information flow [1]) enforce information flow and access control policies. It is not immediately obvious how to compose type parameterization with these languages while preserving the benefits of generic programming. For instance, in Jif, a security typed programming language that extends Java, it is difficult to leverage traditional design patterns while maintaining their expressive power and flexibility.

We provide a methodology and implementation for generics in a language that incorporates security policies into its type system. We evaluate this implementation by constructing many common data structures using Jif.

2 Background and Related Work

In Jif, every type consists of a traditional type component and a dependent label component. Labels describe the integrity and confidentiality policies associated with values of that type as specified by the Decentralized Label Model [3]. For example, the variable declaration `int{o→r} x = 2;` defines a value with type-component `int` and the reader policy {o→r}. This policy states that `o` is the owner of the policy and any principal `q` may read the information in `x` if `q` acts for either `o` or `r`. A lattice can be formed from these labels which is displayed in Figure 1.

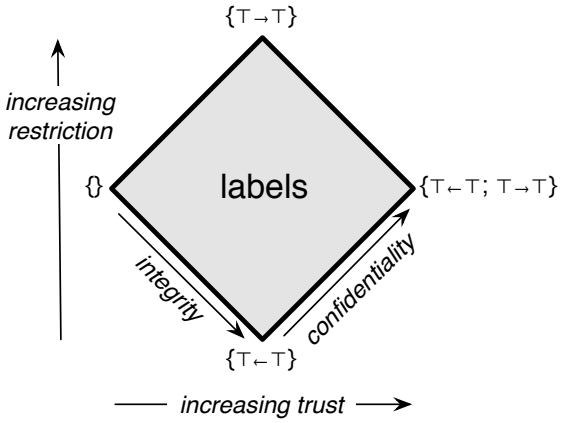

Figure 1: The Jif label lattice

Though Jif ensures the security of a program, it does not support the implementation of a type safe library of data structures. The current collections library stores all data elements as JifObjects (the Jif version of java.lang.Object). This forces programmers to rely on unsafe casts when removing elements of known type from any data structure. Our generics implementation improves the static type checking of this collections framework and removes the reliance on casting to store a group of similarly typed values.

This project also enables highly expressive generic programming in Jif. This requires supporting the concept design pattern. This design pattern provides a more extensible and expressive alternative to subtype constraint by the instantiation of a generic interface (`T extends Comparable<T>`). If the existence of the interface `Comparable<T>` is not known at the time when a class

is created, it is expensive to write adapters allowing that class to be treated as a `Comparable`. Additionally, it is difficult to determine statically if two objects are using the same notion of comparison. The concept design pattern abstracts properties of a data type into model objects which can be easily compared. The expressive power of this design pattern has made previously complex generic code both simple and extensible. By adding generics to Jif, we enable programing with this design pattern and facilitate the application of the concept design pattern to simplify the complexities introduced by information flow control in the Jif type system.

3 Approach

The design of generics for Jif was heavily influenced by the implementation of generics in Java as the two languages share many common constructs. Our primary deviation from Java's design is in not placing any type bound on type parameters by default. Java places an upper bound of Object on type parameters to ensure that they support equality and the computation of a hash code. However, not all types support the semantic notion of extensional equality or the ability to compute a hash code. Additionally, we are not restricted to the use of reference types as type parameters and could parameterize on primitive types; a feature not currently available in Java.

We explored two alternative approaches to handling generics in Jif. Type parameters could either be instantiated on labeled types or on unlabeled types which will be labeled when values of the generic type are declared. Flowcaml takes the first approach following the OCaml style of polymorphic types. We take the second approach in Jif by decoupling the type-component and label-component. Consider the example of a resizable array class with the following signature: `class ResizableArray[type T, label L]`. Such a class would be instantiated on an unlabeled type-component for `T` and a label for `L`. Since `T` is unlabeled, we cannot declare any values of type `T` directly. This can be solved by providing `T` with the label `L` which protects values stored in the `ResizableArray`.

More generally, any class parameterized on a type `T` will take a separate label parameter as the base policy protecting values of type `T`. This allows values with type component `T` to be instantiated on different labels. This is especially useful when values must be created in an area where the label on the program counter does not flow to `L` as `T` can be provided a more restrictive label. This is also helpful when flows relating to data structure access must be checked because `L` provides an upper bound on the labels of all values in the structure removing the need to iteratively check the label component of all values stored in the collection.

This separation of type and label components successfully simplifies the logic required to express the security concerns introduced by generic types. It removes the need to repeatedly retrieve the label component from the type parameter while allowing values of the generic type to be constructed by combining the type parameter with a corresponding label parameter.

4 Results and Contributions

We have implemented this form of generics as an extension to the Jif 3.0 compiler. With this extension we have built simple data structures including a list, a stack, and a queue and are porting the existing collections framework in Jif to a version using type parameters and the concept design pattern. In addition to the improved collections, generics will make Jif more appealing to programmers with an existing background in Java by enabling common Java abstractions that rely on type parameterization.

5 References

1. Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java + information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

2. Vincent Simonet. Flow Caml in a Nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152-165, March 2003.

3. Andrew C. Myers and Barbara Liskov. Protecting Privacy using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410442, October 2000.