Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 49 (2007) 55–64
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

ATERMS for manipulation and exchange of structured data: It's all about sharing

Mark G.J. van den Brand [a], Paul Klint [b,c,*]

[a] Technical University Eindhoven, Department of Mathematics and Computer Science, The Netherlands
[b] Centrum voor Wiskunde en Informatica (CWI), Software Engineering Department, University of Amsterdam, Amsterdam, The Netherlands
[c] Informatics Institute, University of Amsterdam, Amsterdam, The Netherlands

Available online 10 October 2006

Abstract

Some data types are so simple that they tend to be reimplemented over and over again. This is certainly true for *terms*, tree-like data structures that can represent prefix formulae, syntax trees, intermediate code, and more. We first describe the motivation to introduce *Annotated Terms* (ATERMS): unifying several term formats, optimizing storage requirements by introducing maximal subterm sharing, and providing a language-neutral exchange format. Next, we present a brief overview of the ATERM technology itself and of its wide range of applications. A discussion of competing technologies and the future of ATERMS concludes the paper.
© 2006 Elsevier B.V. All rights reserved.

Keywords: ATERMS; Trees; Syntax trees; Abstract syntax trees; Maximal subterm sharing; Annotations; Parsing; Term rewriting; Intermediate data format; Abstract data type; Data exchange; Component-based software; Model checking; Toolbus; ASF+SDF meta-environment; APIGEN; Program generation

1. History and motivation

Some data types are so simple that they tend to be reimplemented over and over again. This is not only true for linked lists and symbol tables but also for *terms*, tree-like data structures that can represent prefix formulae, syntax trees, intermediate code, and more. The explanation is probably that every project needs slight variations of these simple data types and that existing parameterization techniques for software components cannot easily accommodate this variability.

Generic language technology is one of our research topics and related to this research we have developed an interactive development environment for writing language specifications, the ASF+SDF Meta-Environment [35,8]. Terms play an important role in this meta-environment: they are used to represent source code, parse tables, error messages, and so forth. When we made an inventory of term data types in our own software projects related to the ASF+SDF Meta-Environment, it turned out that we were using (and maintaining!) six different variants of a term data type and this provided a strong incentive to look for a single data type that could be used in all projects.

A first attempt at unification were the TOOLBUS terms that were introduced as part of the implementation of the TOOLBUS coordination architecture [3], our component interconnection technology. TOOLBUS terms introduced the simple *make-and-match* paradigm (explained below) for constructing and decomposing terms. A linear string representation was used to exchange terms between components. The C implementation supports automatic garbage collection.

Annotated Terms (or ATERMS as described in [11]) introduced several innovations over the original design: maximal subterm sharing, annotations, a compressed binary exchange format, and a two-level application programming interface (API) that enables both simple and efficient use of

* Corresponding author. Tel.: +31 20 592 4126.
E-mail address: Paul.Klint@cwi.nl (P. Klint).
URLs: www.win.tue.nl/~mvdbrand (M.G.J. van den Brand), www.cwi.nl/~paulk (P. Klint).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.08.009

ATerms. Mature implementations exist for C and Java and experimental implementations for, C#, ML, and Haskell.

Although ATerms were introduced to solve just our own local problem, the wide acceptance of ATerms in numerous projects suggests that this problem was not so local after all. The purpose of the present paper is to sketch the contexts and problem domains in which ATerms are useful and to compare them with competing technologies. The plan of this paper is as follows. In Section 2 we give a quick introduction to ATerms and discuss all technology that has been developed to seamlessly integrate ATerms in applications. Next, we give a survey of applications of ATerms in Section 3. We complete the paper with a comparison of ATerms with other technologies (Section 4) and we speculate about their future (Section 5).

2. The ATerm technology

2.1. A quick introduction to ATerms

The data type of ATerms is defined as follows (see [11] for full details):

- An integer or real constant is an ATerm.
- A function application is an ATerm, e.g., `f(a,b)`.
- A list of zero or more ATerms is an ATerm, e.g., [`f(a)`, `1`,“`abc`”].
- A placeholder term containing an ATerm that represents the type of the placeholder is an ATerm, e.g., `f(⟨int⟩)`.
- A binary large object (BLOB) containing arbitrary binary data is an ATerm.
- A list of (*label*,*annotation*) pairs may be associated with each ATerm. Label and annotation are both ATerms and can thus contain nested annotations.

ATerms are constructed under the constraint that all subterms of all ATerms in a given universe are *maximally shared*. ATerms thus represent directed acyclic graphs and should, in fact, have been called “ADags”. As a consequence, all operations on ATerms are applicative: an ATerm can be decomposed into its constituent parts, but those parts can never be replaced. Replacement can only be achieved by building a new ATerm that contains new values at the places to be modified.

The ATerm API is based on the *make-and-match* paradigm:

- *make* (compose) a new ATerm by providing a pattern for it and filling in the placeholders, in the pattern with given values.
- *match* (decompose) an existing ATerm by comparing it with a pattern and decompose it according to this pattern.

Functions for the input and output of ATerms (both in textual and in binary form) are provided. For efficiency reasons also direct access functions for the constituents of ATerms such as arguments of applications, elements of lists, and annotations are provided. As a first example, consider the following code fragment which shows how to *make* an ATerm (using the C version):

```
ATerm t1 = ATmake(“or(true,false)”);
ATerm t2 = ATmake(“and(true,⟨term⟩)”, t1);
```

First, the term `or(true,false)` is constructed and then assigned to variable `t1`. Next, a second term is constructed using the term pattern “`and(true,⟨term⟩)`”. The value of `t1` is substituted for the placeholder `⟨term⟩` and as a result the term `and(true,or(true,false))` is assigned to `t2`. Now let us try to *match* against this last term:

```
ATerm t3, t4;
if(ATmatch(t2, “and(⟨term⟩,⟨term⟩)”, &t3, &t4)) {
  ...
}
```

The pattern “`and(⟨term⟩,⟨term⟩)`” is matched against the current value of `t2`. The match succeeds and the subterms corresponding to the placeholders, respectively `true` and `or(true,false)` are assigned to the variables `t3` and `t4`. The same example can also be coded using direct access to the term representation. For instance,

```
t3 = ATgetArgument(t2,0);
t4 = ATgetArgument(t2,1);
```

achieves the same effect as the `ATmatch` condition.

As these examples show, the physical structure of the terms being manipulated by this code is explicit in the form of patterns and indices representing argument positions. This intimacy between an application and the ATerm representation it uses is shown in Fig. 1(a). The code would be broken by any change of the representation such as renaming function names (e.g., in a Dutch language version *and* might be replaced by *en*), swapping arguments, adding

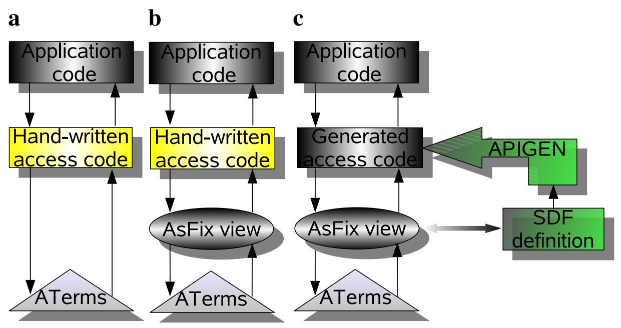

Fig. 1. Application code uses ATerms. (a) Application code uses hand-written code to manipulate ATerms directly; (b) application uses hand-written code to manipulate an AsFix view on ATerms; (c) the AsFix view is defined in an SDF grammar and the access code is generated by Apigen, the application uses this generated code to manipulate the AsFix view.