WALi User Manual

Nicholas Kidd
kidd@cs.wisc.edu

February 11, 2008

1 Introduction

WALi is an open source library implementation of a Weighted Pushdown System (WPDS). WPDSs have been shown to be a powerful formalism for performing interprocedural dataflow analysis [3, 1, 2]. (For a listing of relevant papers see `http://www.cs.wisc.edu/wpis/wpds`.)

2 Interprocedural Control-Flow Graph Encoding

For performing dataflow analysis, the standard practice is to encode the program's interprocedural control flow graph as a single state PDS (see Fig. 1). The weights that annotate the rules of the PDS are dataflow transformers that encode the effect of (abstractly) executing a program statement associated with the rule.

Rule	Control flow modeled
$\langle p, n_1 \rangle \hookrightarrow \langle p, n_2 \rangle$	Intraprocedural edge $n_1 \rightarrow n_2$
$\langle p, n_c \rangle \hookrightarrow \langle p, e_f\ r_c \rangle$	Call to f, with entry e_f, from n_c that returns to r_c
$\langle p, x_f \rangle \hookrightarrow \langle p, \epsilon \rangle$	Return from f at exit x_f

Figure 1: The encoding of an ICFG's edges as PDS rules.

3 Implementing a Weight Domain

A WALi user defines a weight domain D that encodes the desired abstract domain. D must be a subclass of the provided abstract class `wali::SemElem`. Inside of WALi, all instances of SemElem are reference counted. The reference counting implementation is defined by the C++ template class `wali::ref_ptr<T>`. For easier notation, WALi provides the type definition `typedef wali::sem_elem_t wali::ref_ptr<wali::SemElem>`. We next describe each of the methods that must be overridden by the class D.

3.1 One - $\bar{1}$

```
sem_elem_t one() const;
```

`one` returns an instance of the $\bar{1}$ element.

3.2 Zero - $\bar{0}$

```
sem_elem_t zero() const;
```

`zero` returns an instance of the $\bar{0}$ element.

3.3 Combine - $\oplus$

```
sem_elem_t combine( SemElem * se );
```

`combine` returns a new weight that is the combination of `this` and the parameter `se`.

3.4 Extend - $\otimes$

```
sem_elem_t extend( SemElem * se );
```

`extend` returns a new weight that is equal to `this` extended by the parameter `se` (`this` $\otimes$ `se`). `extend` is typically related to functional composition. Is this regard, `this` $\otimes$ `t` is functionally equivalent to `t` $\circ$ `this`.

3.5 Equal

```
bool equal( SemElem * se ) const;
```

`equal` returns true if two weights are equal and false if not. There is currently no method specifically designed to deal with partial orders. However, for any two semiring elements α and β, $\alpha \subseteq \beta \Leftrightarrow \alpha = (\alpha \oplus \beta)$.

3.6 Print

```
std::ostream & print( std::ostream & o ) const;
```

`print` writes a semiring element to the passed in std::ostream parameter. It should return the same std::ostream when finished.

4 Examples

4.1 Reachability Weight Domain

The following weight domain implements simple reachability. The weight is $\bar{1}$ if it is reachable by the WPDS and $\bar{0}$ otherwise. The C++ header and source files are distributed with WALi under the Examples directory.

Code Listing 4.1 (Weight domain implementing reachability.).

```
#include "wali/SemElem.hpp"

using wali::SemElem;
using wali::sem_elem_t;

class Reach : public wali::SemElem
{

    public:

        Reach( bool b ) : isreached(b) {}

        virtual ~Reach() {}

        sem_elem_t one() const { return new Reach(true); }

        sem_elem_t zero() const { return new Reach(false); }

        // zero is the annihilator for extend
        sem_elem_t extend( SemElem* rhs ) {
          Reach* r = static_cast<Reach*>(rhs);
          return new Reach( isreached && r->isreached );
        }

        // zero is neutral for combine
        sem_elem_t combine( SemElem* rhs ) {
          Reach* r = static_cast<Reach*>(rhs);
          return new Reach( isreached || r->isreached );
        }

        bool equal( SemElem* rhs ) const {
          Reach* r = static_cast<Reach*>(rhs);
          return isreached == r->isreached;
        }

        std::ostream & print( std::ostream & o ) const {
          return (isreached) ? o << "ONE" : o << "ZERO";
        }

    protected:
```

```
        bool isreached;

};
```

Using this weight domain is equivalent to using a Pushdown System without weights. All user created weights are $\bar{1}$ and unreachable configurations (abstractly) have weight $\bar{0}$.

5 Creating a Weighted Pushdown System

In this section, we show how to translate pseudo code following pseudo code is translated into a WPDS using the `Reach` semiring.

Code Listing 5.1 (Pseudo Code.)**.**

```
// Pseudo Code //
x = 0
y = 0

fun f()
    n0: <$ f enter node $>
    n1: if( x = 0 )
    n2:     then y := 1
    n3:     else y := 2
    n4: g()
    n5: <$ f exit node $>

fun g()
    n6: <$ g enter node $>
    n7: y := 4
    n8: x := 60
    n9: <$ g exit node $>
```

Code Listing 5.2 (WALi header files)**.**

```
#include "wali/Common.hpp"
#include "wali/wpds/WPDS.hpp"
#include "wali/wfa/WFA.hpp"
#include "Reach.hpp"
```

First, a WPDS `myWpds` is created.

Code Listing 5.3 (Define the WPDS object myWpds.)**.**

```
sem_elem_t reachOne(new Reach(true));
wali::wpds::WPDS myWpds;
```

Then the "keys" for the program locations are defined.

Code Listing 5.4 (Create Keys for program nodes.).
```
wali::Key p = wali::getKey("p");
wali::Key accept = wali::getKey("accept");
wali::Key n[10];
for( int i=0 ; i < 10 ; i++ ) {
     std::stringstream ss;
     ss << "n" << i;
     n[i] = wali::getKey( ss.str() );
}
```

The state and stack symbols of a WPDS rule have a type `wali::Key`. A key is a way of identifying a state or stack symbol of the WPDS. Each key has a unique `wali::KeySource` object associated with it. Some common sources have been defined like `wali::StringSource` and `wali::IntSource`. User's can define their own key source by subclassing the `wali::KeySource` class. The function `wali::getKey` is simply a helpful wrapper for creating a keys from C++ types `std::string` and `int`.

Once all the keys have been defined, the rules are added to the myWpds object.

Code Listing 5.5 (Add intraprocedural edges for f and g.).
```
// f intraprocedural
myWpds.add_rule( p, n[0], p, n[1], reachOne);
myWpds.add_rule( p, n[1], p, n[2], reachOne);
myWpds.add_rule( p, n[1], p, n[3], reachOne);
myWpds.add_rule( p, n[2], p, n[4], reachOne);
myWpds.add_rule( p, n[3], p, n[4], reachOne);

// g intraprocedural
myWpds.add_rule( p, n[6], p, n[7], reachOne);
myWpds.add_rule( p, n[7], p, n[8], reachOne);
myWpds.add_rule( p, n[8], p, n[9], reachOne);
```

Code Listing 5.6 (Add interprocedural edges for f and g.).
```
// f calls g
myWpds.add_rule( p, n[4], p, n[6], n[5], reachOne);

// f return
myWpds.add_rule( p, n[5] , p , reachOne);

// g return
myWpds.add_rule( p, n[9] , p , reachOne);
```

Then the initialized WPDS is printed to the standard output channel and marshalled as XML.

Code Listing 5.7 (Generic output methods.).

```
// Print the WPDS
myWpds.print( std::cout ) << std::endl;

// Marhasll the WPDS as an XML file
std::ofstream fxml( "myWpds.xml" );
myWpds.marshall( fxml );
fxml.close();
```

6 Querying the WPDS

WALi allows for two types of queries, *prestar* and *poststar*. A query takes as input a WPDS and a weighted finite automaton (WFA). A query outputs a new annotated WFA. WFAs are represented by the class `wali::wfa::WFA`. Transitions are added to the WFA class using the `wali::wfa::WFA::addTrans` method. The following sample code computes a *prestar* and *poststar* reachability query for the pseudo code (assuming the same objects are created as in the above C++ program).

Code Listing 6.1 (Prestar query.).

```
    wali::wfa::WFA prequery;
    prequery.addTrans( p, n[4], accept, reachOne );
    query.add_initial_state( p );
    query.add_final_state( accept );
    wali::wfa::WFA answer = myWpds.prestar(prequery);
    answer.print( std::cout );
```

Code Listing 6.2 (Poststar query.).

```
    wali::wfa::WFA postquery;
    postquery.addTrans( p, n[0], accept, reachOne );
    query.add_initial_state( p );
    query.add_final_state( accept );
    wali::wfa::WFA answer;
    myWpds.poststar(query,answer);
    answer.print( std::cout );
```

References

[1] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 2005.

[2] Thomas Reps, Akash Lal, and Nicholas Kidd. Program analysis using weighted pushdown systems. *FSTTCS*, 2007. Invited Paper.

[3] Thomas W. Reps, Stefan Schwoon, and Somesh Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SAS*, 2003.