WPDS++ User Manual *

October 5, 2005

1 Introduction

WPDS++ is an open-source library for implementing Weighted Pushdown Systems (WPDSs). The library is designed as a set of templated C++ [3] classes. The template parameter is a user-defined implementation of a bounded idempotent semiring. Though WPDS++ is a general implementation of Weighted Pushdown Systems, it was designed with the primary purpose of performing interprocedural dataflow analysis of computer programs. It is assumed that the reader is familiar with both Weighted Pushdown Systems and interprocedural dataflow analysis. This user manual does not describe the theoretical underpinnings of Weighted Pushdown Systems or interprocedural dataflow analysis. For more details on the subject, see [1, 2].

2 Implementing a Weight Domain

As described earlier, WPDS++ is a collection of templated C++ class files. The template parameter is a user-defined implementation of a semiring (whose elements are typically a family of dataflow transformers). The semiring must implement a certain collection of methods for a WPDS++ application to compile. The following subsections describe each of these methods. The `T` listed in the method signatures refers to the classname that defines the semiring. A *semiring element* is an instance of the semiring implementation class, and is referred to by `T*`.

2.1 One - $\bar{1}$

```
T* T::one() const;
```

`one` returns a pointer to the $\bar{1}$ semiring element.

2.2 Zero - $\bar{0}$

```
T* T::zero() const;
```

`zero` returns a pointer to the $\bar{0}$ element of the Semiring.

2.3 Combine - $\oplus$

```
T* T::combine( const T* t ) const;
```

`combine` takes a semiring element parameter and returns a new semiring element that is the combination of `this` and the parameter `t` (`*this` $\oplus$ `*t`). Combine should not overwrite `this` or input parameter `t`.

*This material is based upon work supported by the National Science Foundation under Grant No. 9986308. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

2.4 Extend - $\otimes$

```
T* T::extend( const T* t ) const;
```

`extend` takes a semiring element parameter and returns a new semiring element that is equal to `this` extended by the parameter `t` (`this` $\otimes$ `t`). `extend` is typically related to functional composition; i.e., `this` $\otimes$ `t` is functionally equivalent to `t` $\circ$ `this`. Extend should not overwrite `this` or input parameter `t`.

2.5 Equal

```
bool T::equal( T* t ) const;
```

`equal` returns true if two semiring elements are equal and false otherwise. There is currently no method specifically designed to deal with partial orders. However, for any two semiring elements α and β, $\alpha \sqsubseteq \beta$ $\Leftrightarrow \alpha = (\alpha \oplus \beta)$.

2.6 Print

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

`print` writes a semiring element to the `std::ostream` parameter, i.e. `o`. It should return the same `std::ostream` when finished.

2.7 Reference Counting

```
ref_ptr<T>::count_t count;
```

WPDS++ provides reference counting for user-created semiring elements (instances of `T*`). To do this, the semiring element must have a publicly accessible field named `count`. The class `ref_ptr<T>` defines the type `count_t` for the type of the `count` field (it is currently an `unsigned int` but may change in the future). For reference counting to work properly, `count` should be initialized to `0`. To turn off reference counting, initialize `count` to a positive number, e.g. `60`.

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

```
#include "ref_ptr.h"

class Reach
{

    bool isreached;
    public:

        ref_ptr< Reach >::count_t count;

        Reach( bool b ) : isreached(b),count(0) {}

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

        Reach* zero() const { return new Reach(false); }
```

```
        // zero is the annihilator for extend
        Reach* extend( Reach* rhs ) const {
            if( isreached && rhs->isreached )
                return one();
            else // this or rhs is zero()
                return zero();
        }

        // zero is neutral for combine
        Reach* combine( Reach* rhs ) const {
            if( isreached || rhs->isreached )
                return one();
            else
                return zero();
        }

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

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

3 Creating the Weighted Pushdown System

The transformation from program to WPDS is straightforward. There are three types of rules in the WPDS corresponding to three types of edges in the program's callgraph. The type of a rule is distinguished by the number of stack symbols on the right-hand side of the rule. More precisely, rules with one, two, or zero right-hand side stack symbol(s) correspond to intraprocedural, interprocedural call, and interprocedural return edges, respectively. To be clear, Figure 2 is a translation of the pseudo code in Figure 1 into a WPDS using the `Reach` semiring.

The C++ program in Figure 2 creates a WPDS and prints it to `std::out`. Some new classes and types are used in the example program. One class is named `wpds::Semiring`. The `wpds::Semiring` class is used by the `wpds::WPDS` class to call the user-defined semiring methods (in this case, the methods of class `Reach`). It is instantiated with a semiring element (it is customary to use $\bar{1}$ for instantiation). A second type used in Figure 2 is the `wpds::wpds_key_t` object. The WPDS++ library only knows about keys. A key is a way of identifying a state or stack symbol of the WPDS. Each key has a unique `wpds::key_source` object associated with it. Some common sources have been defined like `wpds::string_src` and `wpds::int_src`. User's can define their own key source by subclassing the `wpds::key_source` class (see `key_source.h`). The function `str2key` is simply a helpful wrapper for creating a `wpds::string_src` and returns the `wpds::wpds_key_t` associated with that object. A similar function `int2key` exists for working with `wpds::int_src`. Once all the keys have been

```
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 $>
```

Figure 1: Pseudo code.

defined, the rules are added to the myWpds object.

4 Queries in WPDS++

4.1 Prestar and Poststar

WPDS++ allows for two types of queries, *prestar* and *poststar*. A query takes as input a WPDS, a Configuration Automaton, and a Semiring. A query outputs a new annotated Configuration Automaton. Configuration Automata are represented by the class `wpds::CA`. The constructor takes a `wpds::Semiring` as its one input. Transitions are added to the CA class using the `wpds::CA::add` method. The following sample code creates a Configuration Automaton `query`, and performs a *poststar* reachability query with respect to WPDS `myWpds` (created in Figure 2). A *prestar* query is performed similarly.

```
wpds::CA< Reach > query( s );
query.add( p, n[0], accept, reachOne );
query.print( std::cout << "BEFORE\n" ) << std::endl;
wpds::CA< Reach > answer = wpds::poststar< Reach >(myWpds,query,s);
answer.print( std::cout << "\nAFTER\n" ) << std::endl;
```

4.2 Path Summary

A path-summary query is performed on a `wpds::CA`. It annotates the states of the automaton with a semiring element that represents the sum ($\oplus$) over all paths from that state to the accepting state of the automaton. If there is no accepting state, then every state will be annotated with $\bar{0}$. The following code illustrates a path-summary query (again assuming we have the same objects created in Figure 2 and subsection 4.1). After the path-summary method completes, the code retrieves the weight annotation for state `p` and writes it to `std::cout`.

```
answer.path_summary();
ref_ptr< Reach > pWeight = answer.state_weight( p );
std::cout << "Weight on state \"p\": ";
```

```
#include "WPDS.h"
#include "Reach.h"
#include <string>
#include <sstream>

int main()
{
    Reach* reachOne = new Reach(true);
    wpds::Semiring< Reach > s( reachOne );
    wpds::WPDS< Reach > myWpds(s);
    wpds::wpds_key_t p = str2key("p");
    wpds::wpds_key_t accept = str2key("accept");
    wpds::wpds_key_t n[10];
    for( int i=0 ; i < 10 ; i++ ) {
        std::stringstream ss;
        ss << "n" << i;
        n[i] = str2key( ss.str() );
    }

    // 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);

    // f call 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);

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

    return 0;
}
```

Figure 2: WPDS++ encoding of the pseudo code in Figure 1.

```
pWeight->print( std::cout ) << std::endl;
```

WPDS++ returns weights as values of type `ref_ptr<T>` (i.e., an instance of the `ref_ptr` class instantiated with the user's semiring). In most cases, a `ref_ptr` acts like an ordinary C++ pointer. Two additional methods that might be of interest are `ref_ptr<T>::is_valid` and `ref_ptr<T>::get_ptr`. `ref_ptr<T>::is_valid` is a null pointer check, and `ref_ptr<T>::get_ptr` returns the underlying object pointer. There is no guarantee that a pointer retrieved from a `ref_ptr` will remain valid if the `ref_ptr` object goes out of scope. For more information see `ref_ptr.h`.

4.3 Reglang Query

Reglang queries are queries over an automaton with respect to a regular language.[1] For example, after a *poststar* (*prestar*) query, the user can ask for the weight that is the sum over all paths that end (begin) in a configuration in the regular language defined by some automaton (i.e., an object of class `wpds::CA`). In terms of interprocedural dataflow analysis, this allows the user to perform stack-qualified queries on a supplied program. In the current implementation, the regular language is supplied as an object of class `wpds::CA`. The weights on the `wpds::CA` encoding the regular language are silently ignored. The following code performs a `reglang_query` on our running example (Figure 2 and subsection 4.1). It asks for the weight of function `g`'s exit node with calling context `f`. Calling contexts are expressed by the return points of function call sites. The set of function return points is the set consisting of the rightmost stack symbols of WPDS rules with two right-hand side stack symbols.

```
wpds::CA< Reach > reglang( s );
Reach *ignored = new Reach( true );
// manually add transitions of the regular expression
reglang.add( str2key("t1") , n[9] , str2key("t2") , ignored);
reglang.add( str2key("t2") , n[5] , str2key("t3") , ignored);
reglang.add_initial_state( str2key("t1") );
reglang.add_final_state( str2key("t3") );
// answer is the same CA from the earlier poststar example
ref_ptr< Reach > reglangWeight = answer.reglang_query( reglang )
std::cout << "Result of reglang_query: ";
reglangWeight->print( std::cout ) << std::endl;
```

References

[1] Thomas Reps, Stefan Schwoon, Somesh Jha, and Dave Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 2005. to appear. 1

[2] Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002. 1

[3] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000. 1

[1] `wpds::CA<T>::reglang_query` is not currently supported with Visual Studio 6.0