DiGr: Directed Graph Processing Language

Ari Golub
Bryan Oemler
Dennis V. Perepelitsa

Columbia University
COMS W4115: Programming Languages and Translators

22 December 2010
Final Project Presentation
What can DiGr do?

- Represent trees, graphs, walks, (mirrors, knots, etc).
- Model everything from basic computer science constructs to network-based problems in engineering and industry.
- Store information in nodes and edges without overhead or hassle.
- Recursively or iteratively walk and modify directed graphs in user-specified ways.
What is the DiGr language / compiler like?

- Imperative.
- Compiled. Target language is C++, which is in turn compiled with g++ and linked against the DiGr backend.
- Statically (and locally) scoped.
- Specific graph-related objects (nodes, edges, walks) on top of a typed C-like base.
- Strongly typed.
Integers, floating point numbers and strings are primitive types.

: this is a comment:
str name = "Ari"
int age!
age = 22!
flt gpa = 4.0! : statements end with a ! :

Opts have no return types, but have in (not globally bound) and out (in-scope from the program that called them) variables.

opt times_two(in int n; out int doubled){
    doubled = n * 2!
}

Golub, Oemler, Perepelitsa

DiGr: Directed Graph Processing Language
- The high-level objects in DiGr are nodes and edges:

```plaintext
node n1!
node n2!
n1 -> n2! : n1 and n2 are now connected :
```

- Node and edge identifiers are handles. Edges are usually created anonymously:

```plaintext
edge e = n1.outedge(0)!
node target = e.innode!
```
Attributes are created as soon as they are referenced or assigned:

```plaintext
node city!
city.population = 60000!
print(city.population)! : prints 60000 :
print(city.area)! : defaults to 0 :
```

Connection contexts efficiently create graphs, and store the handles to the nodes in an array:

```plaintext
node binaryTree[7] = |3->(1->0,2),(5->4,6)| !
```
Crawls and rules

- A crawl is an opt run on a node, and can call a rule that tells it where to go next.

```plaintext
crawl markNode(in int marker) {
    current.mark = marker!
    call!  }
```

- Rules modify the queue of nodes to visit when called:

```plaintext
rule followLighterEdge{
    edge e1 = current.outedge(0)!
    edge e2 = current.outedge(1)!
    if (e1.weight < e2.weight)
        { add(e1.child(0))!  }
    else { add(e2.child(0))!  }
}
```
crawl printNode() { print(current.id)! call! } : print the node id and call the rule:

rule preOrder { addByFront(node.id,~,2)! } : add up to two children, smallest id first:

opt main() {
    node tree[5] = |3->(1->0,2),4| !
    tree[0].id = 0! tree[1].id = 1! tree[2].id = 2!
    tree[3].id = 3! tree[4].id = 4!
    printNode() from tree[3] with preOrder!
    : prints 3 1 0 2 4 :
}
Scanner turns DiGr program from standard input into tokens. Lexical correctness.

Parser creates initial AST (nested OCaml tree of typed tuples). Syntactical correctness.

Interpreter verifies AST. Semantic correctness.
The interpreter has several duties:

- Create global scope for `opt/rule/crawl` signatures.
- Create symbol table for all local scopes (inside functions and blocks).
- Check the scope of all identifiers.
- Check typing for all statements (recursively into expressions), including assignment, function calls, etc.

After the front-end stage, intermediate representation of a sensible program (instance of DiGr AST).
C++ AST: stripped-down, holds intermediate representation of C++ program. A few shortcuts, but largely extensible. C++ AST assures *syntactical* correctness of output.

Translator: converts DiGr AST to C++ AST. Does no semantic checking.

Compiler: crawls the C++ AST and outputs C++ code.

`g++`: turns compiled DiGr code into an executable.
rule myrule {
    int n = 0!
    while (n < current.outedges) {
        edge tmp_edge = current.outedge(n)!
        if (tmp_edge.mark == 1) {
            node destination = tmp_edge.innode!
            add(destination)!
        }
        n = n + 1!
    }
}

crawl thecrawl() {
    print (current.id)!
    call!
}
#include "digr.h"
#include <iostream>
void myrule(DiGrNode *current, deque<DiGrNode*> *returnQueue) {
    int n = 0;
    while(n < current->OutEdges())
    {
        DiGrEdge *tmp_edge = current->getOutEdge(n);
        if(tmp_edge->getAttribute("mark") == 1)
        {
            DiGrNode *destination = tmp_edge->inNode();
            returnQueue->push_back(destination);
        }
        else{}
        n=n + 1;
    }
}
void thecrawl(DiGrNode *current, void (*rule)(DiGrNode*, deque<DiGrNode*>*)) {
    deque<DiGrNode*> *queue = new deque<DiGrNode*>();
    queue->push_back(current);
    do {
        current=queue->front();
        queue->pop_front();
        std::cout << current->getAttribute("id") << std::endl;
        rule(current, queue);
    } while (queue->size() > 0);
For each test program, we have a *gold standard* that execution should output. Every build, we compile and execute all tests and compare output with the gold standard.

Test atomic DiGr elements from low-level (basic types, arithmetic, function calls, etc.) to high-level (graphs, attributes, connection contexts, etc.).

Test programs which integrate a wide cross-section of features.

Test errors at compilation (really, the interpret stage), and at run-time.