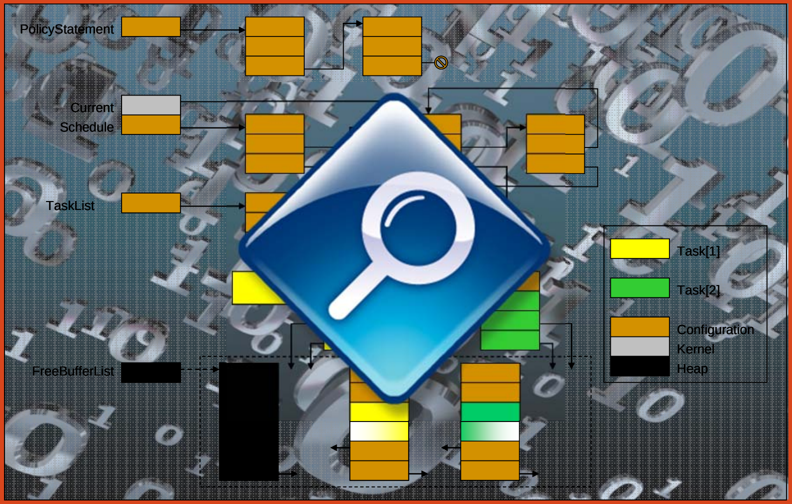
Data Flow Logic

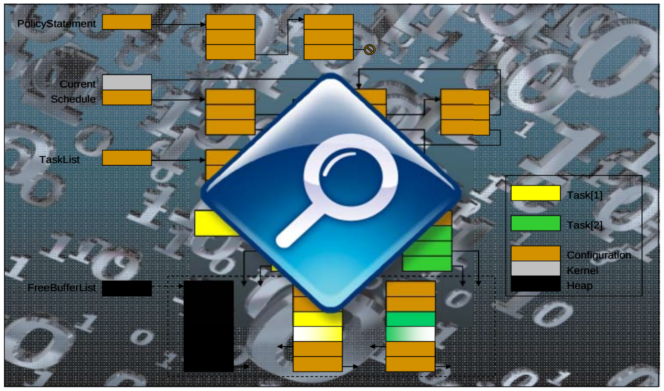
David Greve
05 December 2011



Resarch sponsored by Space and Naval Warfare Systems Command
Contract N65236-08-D-6805
Distribution Statement A: Approved for public release; distribution unlimited.

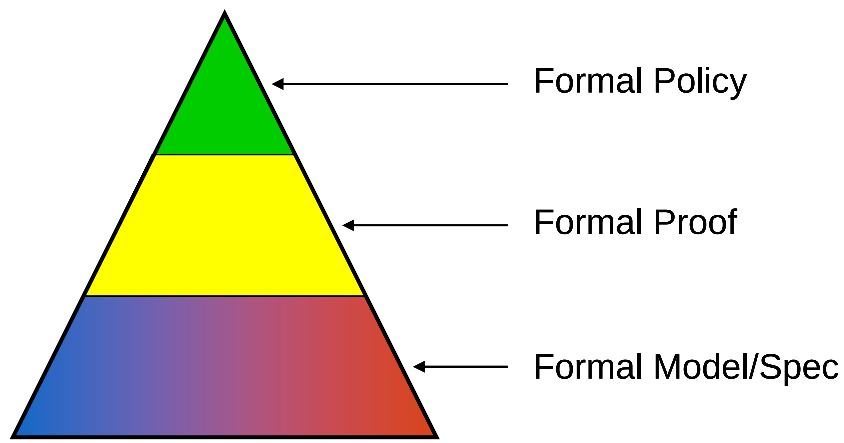
DFL: Data Flow Logic

- A Domain Specific Annotation Language
  - Supporting Information Flow Modeling & Analysis
- Informed By
  - AAMP7, GHS, Cybersecurity
  - JML, SPARK
- Targeting C Source Code
  - Leverages GCC __attribute__ syntax
  - And C Macros
- Minimize Verification Complexity
  - Static Checking (decidable) – 90%
  - Formal Proofs (undecidable) - 10%



Desired Enhancements

- Models
  - Reduce Maintenance
  - Improve Tracability
  - Reason Directly About Software
- Specifications
  - More Accessible
  - Leverage Type System
- Policies
  - More Intuitive
  - More Abstract
- Reasoning
  - Increase Automation
  - Leverage 90/10 Paradigm
  - Develop Static Checker



Usage

- Identify and Name Information Domains of Interest
  - DFL_DOMAIN TS,S,U;
- Classify System Variables w/to Domains
  - int key DFL_WITHIN((TS));
- Articulate Flow Contracts Between Domains
  - DFL_CONTRACT foo (...) ...
- Verify Contracts
  - Run dfl_analyzer

DFL Software Modeling Solution

- CIL – C Intermediate Language
  - Collection of C processing tools developed at Berkeley
  - Includes parser and a variety of transformation steps
  - Extended to emit an intermediate Abstract Syntax Tree (AST)

- C AST
  - Supports Most Basic Operations
  - Also Supports DFL Annotations

- AST Support in ACL2
  - Read, check, process, and write ASTs
    - ACL2 as a programming language!
  - Enables construction of software analysis tools
    - Static Information Flow Analysis
  - Possible to define/characterize execution semantics w/to AST
    - ACL2 is a theorem prover!
    - Prove Static Analysis Correct

Specification Style

- DFL Specifications
  - Look a lot like C source programs
  - Leverage C declarations, preprocessor

- Supports Integrated Specifications
  - Specification is part of the source code
  - Compiler Compatibility

```
DFL_DOMAIN TS;

int key DFL_WITHIN((TS));
```

- Supports Federated Specifications
  - Specification is expressed outside of source code
  - Maintain Consistency

Domains

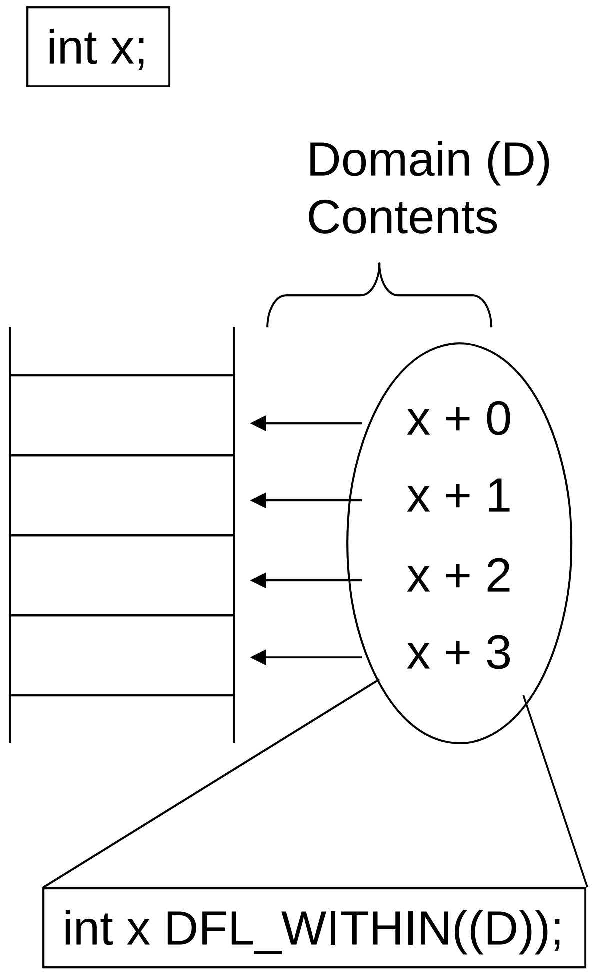
- Security Policies are often Expressed in terms of “Security Domains”
  - No flow from Classified to Unclassified
  - Partition A may communicate only with Partition B

- DFL Domains enable collections of program variables and heap allocated data structures to be gathered together and treated collectively under a single name.
    - These are the variables that define the state of partition A
    - These are the variables that define the state of partition B

- Giving such collections names allows us to articulate concise communication policies.
    - Partition A may get information from partition B

DFL Domain Contents

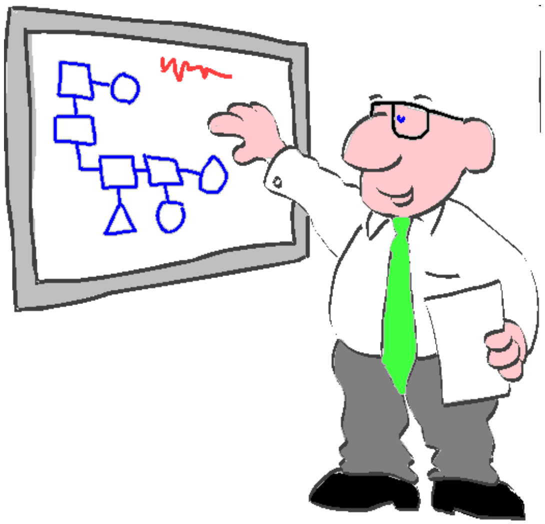
- Populated by Primitive Addresses
  - char * pointers
- Cover Span of Data Type
  - sizeof(type)
- Used to Express Policies
  - Use Domains rather than variables



Classification

- Classification is the process of determining which variable to assign to which domains under what conditions.
  - Most of the Work

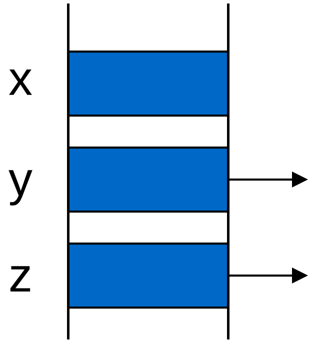
- The end result of the classification process is a classification procedure.
  - Captures an Understanding of the System
  - Provides a context within which information flow policies may be accurately expressed and verified



Global Classifications

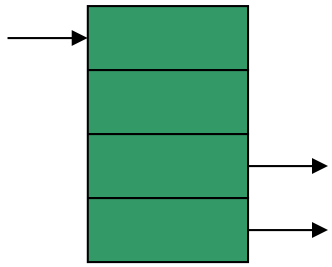
- A global classification is used to classify global variables.
  - The Root Set
- The body of a global classification consists of a sequence of global variable declarations and associated attributes
- Every declaration appearing in the body of the procedure must match an existing global declaration (modulo DFL attributes)
  - They extend the original declaration

```
int x;

DFL_GLOBAL_CLASSIFICATION GClass() {
  int x DFL_WITHIN((D));
};
```



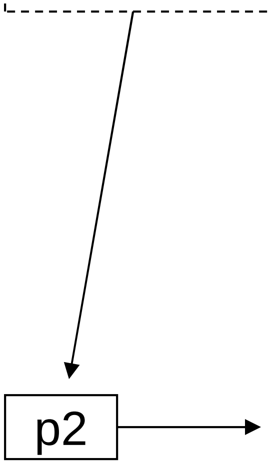
Heap Classifications

- A heap classification procedure is used to classify pointers and the heap objects they identify.
- The first argument to a heap classification procedure is a void pointer.
- The body of the heap classification procedure contains a type (re)declaration of the void pointer.
- All such type declarations must match an existing type declaration (modulo DFL attributes)
  - They extend the original declaration
- Heap Classifications are Initiated by Crawling (following) pointers



Interpreting Classifications

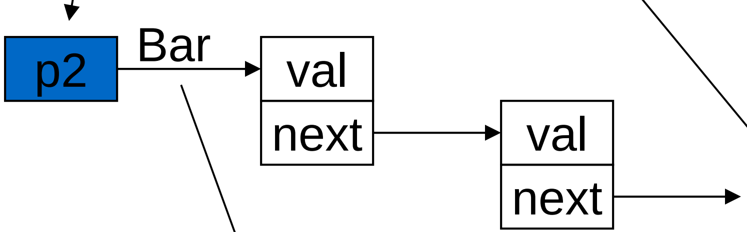
```
typedef struct list {
  int   val;
  list *next;
} list;

list *p2;
```




```
DFL_DOMAIN TS,S,C,U;

DFL_HEAP_CLASSIFICATION Foo (void *x) {

  struct list {
    int   val  DFL_WITHIN((U));
    list *next DFL_WITHIN((U))
               DFL_CRAWL((Foo(next)));
  } *x;

}

DFL_HEAP_CLASSIFICATION Bar (void *x) {

  struct list {
    int   val  DFL_WITHIN((TS));
    list *next DFL_WITHIN((TS))
               DFL_CRAWL((Foo(next)));
  } *x;

}

DFL_GLOBAL_CLASSIFICATION Global () {

  list * p2 DFL_WITHIN((S))
            DFL_CRAWL((Bar(p2)))
            DFL_WHERE((p2 && p2->next));

}
```

Interpreting Classifications

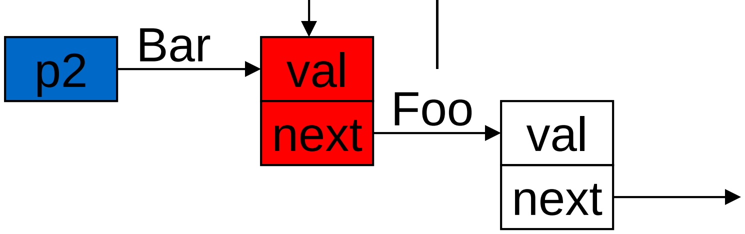
```
typedef struct list {
  int   val;
  list *next;
} list;

list *p2;
```




```
DFL_DOMAIN TS, S,C,U;

DFL_HEAP_CLASSIFICATION Foo (void *x) {

  struct list {
    int   val  DFL_WITHIN((U));
    list *next DFL_WITHIN((U))
               DFL_CRAWL((Foo(next)));
  } *x;

}

DFL_HEAP_CLASSIFICATION Bar (void *x) {

  struct list {
    int   val  DFL_WITHIN((TS));
    list *next DFL_WITHIN((TS))
               DFL_CRAWL((Foo(next)));
  } *x;

}

DFL_GLOBAL_CLASSIFICATION Global () {

  list * p2 DFL_WITHIN((S))
            DFL_CRAWL((Bar(p2)))
            DFL_WHERE((p2 && p2->next));

}
```

Interpreting Classifications

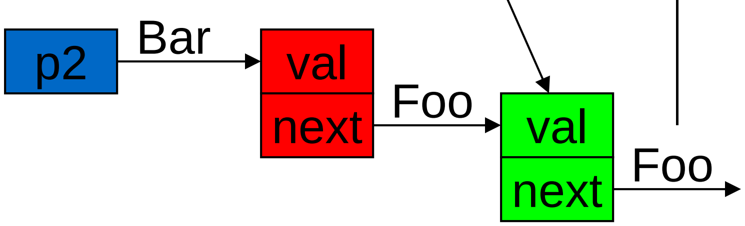
```
typedef struct list {
  int   val;
  list *next;
} list;

list *p2;
```




```
DFL_DOMAIN TS, S, C, U;

DFL_HEAP_CLASSIFICATION Foo (void *x) {

  struct list {
    int   val  DFL_WITHIN((U));
    list *next DFL_WITHIN((U))
               DFL_CRAWL((Foo(next)));
  } *x;

}

DFL_HEAP_CLASSIFICATION Bar (void *x) {

  struct list {
    int   val  DFL_WITHIN((TS));
    list *next DFL_WITHIN((TS))
               DFL_CRAWL((Foo(next)));
  } *x;

}

DFL_GLOBAL_CLASSIFICATION Global () {

  list * p2 DFL_WITHIN((S))
            DFL_CRAWL((Bar(p2)))
            DFL_WHERE((p2 && p2->next));

}
```

Interpreting Classifications

```
typedef struct list {
  int   val;
  list *next;
} list;

list *p2;
```




```
DFL_DOMAIN TS, S, C, U;

DFL_HEAP_CLASSIFICATION Foo (void *x) {

  struct list {
    int   val  DFL_WITHIN((U));
    list *next DFL_WITHIN((U))
               DFL_CRAWL((Foo(next)));
  } *x;

}

DFL_HEAP_CLASSIFICATION Bar (void *x) {

  struct list {
    int   val  DFL_WITHIN((TS));
    list *next DFL_WITHIN((TS))
               DFL_CRAWL((Foo(next)));
  } *x;

}

DFL_GLOBAL_CLASSIFICATION Global () {

  list * p2 DFL_WITHIN((S))
            DFL_CRAWL((Bar(p2)))
            DFL_WHERE((p2 && p2->next));

}
```

C2DFL

- Classification Process can be Daunting
  - Need Tools to Help Kickstart the Process

- C2DFL Takes a Target Source File
  - Identifies Interesting Types from Source
    - Generates a Generic Heap Classification
  - Identifies Global Variables from Source
    - Generates a Generic Global Classification
  - Resulting Spec
    - Useful as an Initial Specification Template
    - Ideally: constitutes a foundation for a valid policy

Requires: Pre Conditions

- Pre Conditions
  - Conditions required for correct operation
  - Conditions necessary to satisfy post conditions

```
DFL_CONDITION
void MyFun_pre(p1,p2)
  list *p1 DFL_CRAWL((Foo(p1)))
           DFL_WITHIN((C))
           DFL_WHERE((p1)));
  list *p2 DFL_CRAWL((Goo(p2)))
           DFL_WITHIN((S));
{
 DFL_ASSERT(p2 && p2->next);
 GlobalClassification();
 return;
}
```

Classification Procedures

Provides: Post Conditions

- Post Conditions
  - Obligation on Procedure Behavior

```
DFL_CONDITION
int MyFun_post (p1,p2)
  list *p1;
  list *p2;
{
  DFL_DEPENDS((TS) ,(TS,U));
  DFL_DEPENDS((U)  ,(U));
  return;
}
```

Information Flow Graph

Contracts

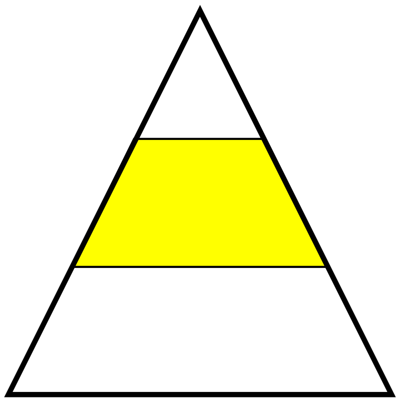
- Procedure Declarations
  - With REQUIRES/PROVIDES/INSTANCE annotations
- Signature of Contract must match
  - Signature of Instance (Target) Procedure
  - Signature of Post Condition
- If procedure body exists
  - Analysis will be attempted
- If only signature exists
  - Becomes an obligation on Future Implementation
- Used in Analysis of Calling Procedures

```
DFL_CONTRACT int MyFun_contract(list *p1,list *p2)
  DFL_REQUIRES(MyFun_pre (p1,p2))
  DFL_PROVIDES(MyFun_post(p1,p2))
  DFL_INSTANCE(MyFun(p1,p2));
```

Checker

- Reasoning
  - Increase Automation
  - Leverage 90/10 Paradigm
  - Develop Static Checker

- Static Analysis
  - For properties simple enough to be checked by programs that, themselves, adhere to a set of logical rules of inference.
  - Property may be built in to the tool, so it is never explicitly expressed
  - Checker itself can be verified

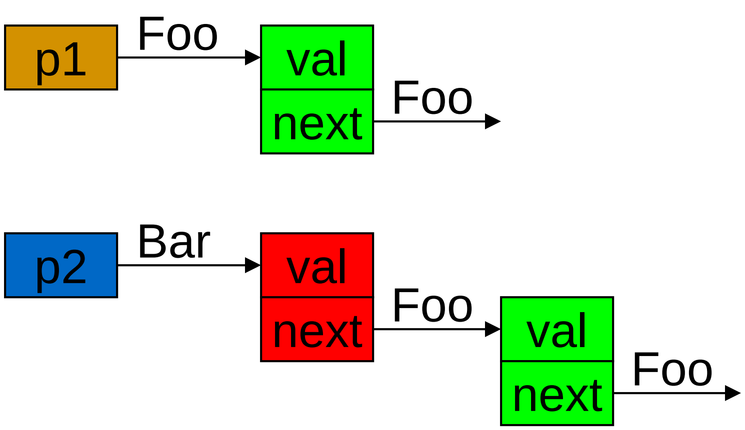


- Static Checking Tools
  - Small, Regular Gap
    - Type Checking
  - Tend to be very fast

- Undecided Properties
  - Verification Conditions
  - ACL2/Gryphon/SAT

Reasoning Example

```
typedef struct list {
  int   val;
  list *next;
} list;

void BadBoy (list * p1, p2) {
  p1->val = p2->next->val;
}
```

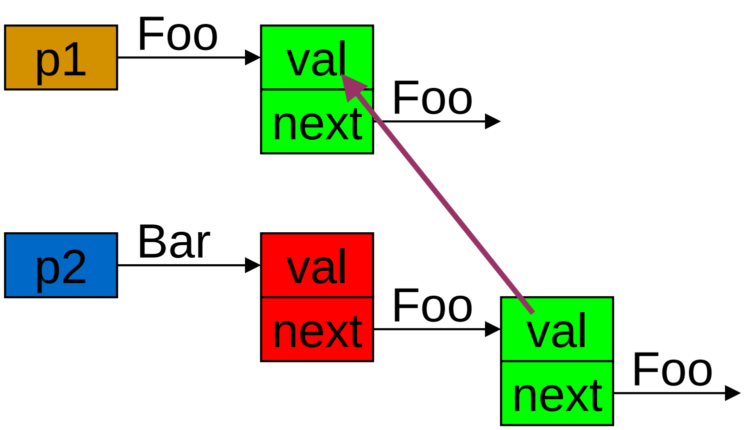



```
DFL_DOMAIN TS, S, C, U;

DFL_HEAP_CLASSIFICATION Foo (void *x) {
  struct list {
    int   val  DFL_WITHIN((U));
    list *next DFL_WITHIN((U))
               DFL_CRAWL((Foo(next)));
  } * x;};

DFL_HEAP_CLASSIFICATION Bar (void *x) {
  struct list {
    int   val  DFL_WITHIN((TS));
    list *next DFL_WITHIN((TS))
               DFL_CRAWL((Foo(next)));
  } *x ;};

DFL_CONDITION void BadBoy_requires(p1,p2)
  list * p1 DFL_CRAWL(Foo(p1))
            DFL_FROM((C))
            DFL_WHERE(p1);
  list * p2 DFL_CRAWL(Bar(p2))
            DFL_FROM((S));
{
  DFL_ASSERT(p2 && p2->next);
  return;
}
```

Reasoning Example

```
typedef struct list {
  int   val;
  list *next;
} list;

void BadBoy (list * p1, p2) {
  p1->val = p2->next->val;
}
```

U <- U



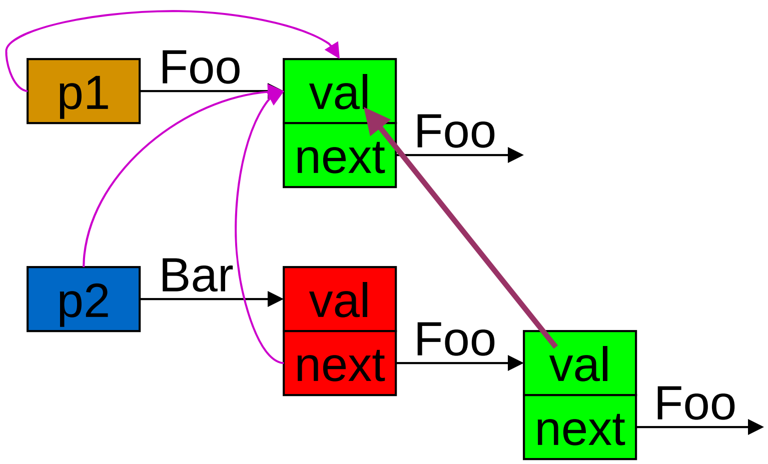

```
DFL_DOMAIN TS, S, C, U;

DFL_CONDITION BadBoy_requires(p1,p2)
  list * p1 DFL_CRAWL(Foo(p1))
            DFL_FROM((C))
            DFL_WHERE(p1);
  list * p2 DFL_CRAWL(Bar(p2))
            DFL_FROM((S));
{
  DFL_ASSERT(p2 && p2->next);
  return;
}

DFL_CONDITION BadBoy_provides(p1,p2)
  list * p1;
  list * p2;
{
  DFL_DEPENDS((TS),(TS,U));
  DFL_DEPENDS((U),(U));
  return;
}

DFL_CONTRACT
void BadBoy_desired(list * p1,list * p2)
  DFL_REQUIRES(BadBoy_requires(p1,p2))
  DFL_PROVIDES(BadBoy_provides(p1,p2))
  DFL_INSTANCE(BadBoy(p1,p2));
```

TS <- TS + U
U <- U

Reasoning Example

```
typedef struct list {
  int   val;
  list *next;
} list;

void BadBoy (list * p1, p2) {
  p1->val = p2->next->val;
}
```

```
U <- C + S + TS + U
```



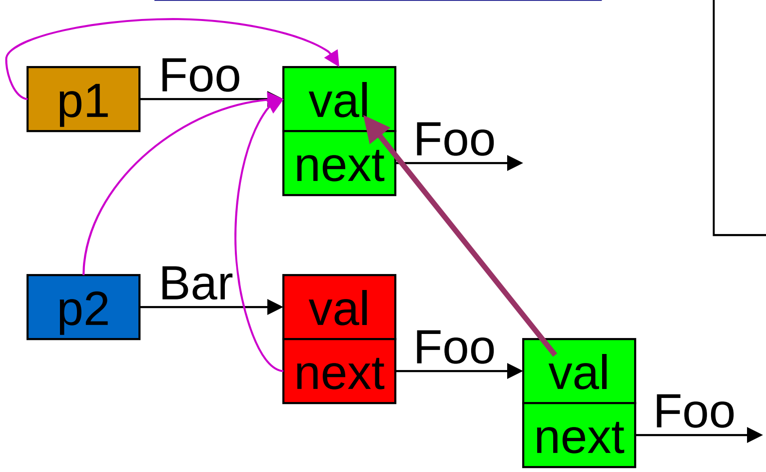

```
DFL_DOMAIN TS,S,C, U;

DFL_CONDITION BadBoy_requires(p1,p2)
  list * p1 DFL_CRAWL(Foo(p1))
            DFL_FROM((C))
            DFL_WHERE(p1);
  list * p2 DFL_CRAWL(Bar(p2))
            DFL_FROM((S));
{
  DFL_ASSERT(p2 && p2->next);
  return;
}

DFL_CONDITION BadBoy_provides(p1,p2)
  list * p1;
  list * p2;
{
  DFL_DEPENDS((TS),(TS,U));
  DFL_DEPENDS((U),(U));
  return;
}

DFL_CONTRACT
void BadBoy_desired(list * p1,list * p2)
  DFL_REQUIRES(BadBoy_requires(p1,p2))
  DFL_PROVIDES(BadBoy_provides(p1,p2))
  DFL_INSTANCE(BadBoy(p1,p2));
```

```
TS <- TS + U
U  <- U
```

Reasoning Example

```
typedef struct list {
  int   val;
  list *next;
} list;

void BadBoy (list * p1, p2) {
  p1->val = p2->next->val;
}
```

U <- C + S + TS + U




```
DFL_DOMAIN TS, S, C, U;

DFL_CONDITION void BadBoy_requires(p1,p2)
  list * p1 DFL_CRAWL(Foo(p1))
            DFL_FROM((C))
            DFL_WHERE(p1);
  list * p2 DFL_CRAWL(Bar(p2))
            DFL_FROM((S));
{
  DFL_ASSERT(p2 && p2->next);
  return;
}

DFL_CONDITION void BadBoy_provides2(p1,p2)
  list * p1;
  list * p2;
{
  DFL_DEPENDS((TS),(TS,U));
  DFL_DEPENDS((U),(TS, S,C, U));
  return;
}

DFL_CONTRACT
void BadBoy_actually(list * p1,list * p2)
  DFL_REQUIRES(BadBoy_requires(p1,p2))
  DFL_PROVIDES(BadBoy_provides2(p1,p2))
  DFL_INSTANCE(BadBoy(p1,p2));
```

Status

- Demonstrable Prototype
  - Regression Suite of Small Examples
    - Verifies Correct Contracts
    - Identifies Incorrect Contracts
  - Analyzed 5 Minix Procedures
    - Employed Compositional Reasoning
- Faster than Proof
  - Small examples in under a second
    - 23 Contracts / 2.3 Seconds
  - Largest DFL example
    - 54 Source Lines, 224 Logical Paths
    - 30 sec, .0025 s/sloc
  - Previous Proof Time
    - 36 source lines, 4 Logical Paths
    - 5 min, 1.25 s/sloc
- Features
  - 5/12 C constructs (Missing GOTO, Looping)
  - Under Continued Development