

μ CIL : A New Core Language for C

No Author Given

No Institute Given

Abstract. This paper proposes a new core language for C that is expression-oriented. The main focus of this core language is to support easier program analysis. We provide alternative constructs in the translated code to facilitate program analysis, where required. These constructs may exist side-by-side with the corresponding implementation code.

1 Introduction

- expression-oriented
- no goto/loops
- no static variables
- less distinction between stack and heap allocation, use heap allocation where necessary
- no address-of operator
- translates struct assignment to primitive copying
- array constructs are treated as pointers
- returns implemented by pass-by-ref

some issues : unsafe type casting warning on type errors; function-type pointers

2 Source and Target Languages

Source Language - CIL

```
P ::= tdecl* vdecl* meth*
tdecl ::= type v = t ;
vdecl ::= [STATIC] t v ;
meth ::= t mn((t v)*) {(t v)*; b}
prim ::= byte | int | float | bool | void | ..
t ::= prim | v | (t1|..|tn) | (t1,..,tn) | t* | t[k] | (t1,..,tn)->t
l ::= f | f.l
u ::= v | *e | u[e] | u.l
w ::= v | u[e] | u.l
x ::= w | k
b ::= (label:s)*
s ::= u := e | u:=mn(e*) | mn(e*) | malloc e | free e
      | s1 ; s2 | if e then b1 else b2 | switch e b | b | while e b
      | break | continue | return e | goto l
e ::= x | (t) e | &w
```

Target Language - μ CIL

```

P ::= tdecl* vdecl* meth*
tdecl ::= type v = t ;
vdecl ::= t v ;
meth ::= void mn((ref] t v)*) {(t v)*; s}
prim ::= byte | int | float | bool | void | ..
t ::= prim | v | (t1|..|tn) | (t1,..,tn) | t* | t[k] | (t1,..,tn)->t
l ::= f | f.l
u ::= v | *v
w ::= u | u.l
x ::= w | k
e ::= x | (t)x | new t | mn(x*) -- primitives incl v+x ptr arithmetic
s ::= w := e | mn(x*) | free(w) | s1 ; s2
    | if x then s1 else s2 | return

```

3 Translation

3.1 Decomposing Struct Copying

Struct copying shall be broken down into separate assignments of primitive values. This step greatly facilitates analysis. We recursively process as follows:

$$\frac{w_1:t \quad \text{IsStruct}(t) \quad \text{fields}(t) = \{f_1, \dots, f_n\} \quad w_1.f_i := w_2.f_i \implies c_i, i \in 1..n}{w_1 := w_2 \implies \{c_1, \dots, c_n\}}$$

3.2 Handling Pass-by-Value Parameters

For the pass-by-value parameters of each method, we copy each of its parameters to a new local variable. Any address-of operation shall be handled by the corresponding local variable. For pass-by-reference parameter, we do not use any local variables but shall instead allow its updated value to be passed back to its caller. As a language restriction, we disallow the address-of operator to be applied to **ref** parameters. Note that **ref** parameters are currently introduced by program analysis to handle (i) conversion of loop to tail-recursion, and (ii) removal of method's result. Such parameters are assumed to have the pass by reference semantics. For simplicity, we have assumed that these **ref** parameters are unaliased.

$$\frac{\text{fresh } w^*}{t \text{ mn}((t v)^*, (\mathbf{ref} t u)^*) \{s\} \implies t \text{ mn}((t v)^*, (\mathbf{ref} t u)^*) \{(t w)^*; (w := v)^*; [v \mapsto w]^*; s\}}$$

As an optimization, we need only localise parameters if they have been updated in method body. A parameter v has been updated if either v or $v.l$ has appeared on the LHS of an assignment statement. A parameter v may also be updated if either $\&v$ or $\&(v.l)$ has occurred. This can happen since such pointers may be dereferenced in the LHS of an assignment statement.

There may also be a need to handle parameters with **in** – **out** semantics. In this scenario, we allow the parameters to be aliased but will enforce a left-to-right ordering of updates to these parameters.

3.3 Removing Result from a Method via a **ref** Parameter

$$\frac{t \neq \text{void} \quad \rho = [(return\ e) \mapsto (r := e; return)]}{t\ mn(([\mathbf{ref}]\ t\ v)^*)\ \{s\} \implies void\ mn(([\mathbf{ref}]\ t\ v)^*, \mathbf{ref}\ t\ r)\ \{\rho\ s\}}$$

$$w := mn(e^*) \implies \{t\ r; mn(e^*, r); w := r\}$$

Under some circumstances, it is possible to perform the above without introducing intermediate variables:

$$w := mn(e^*) \implies mn(e^*, w)$$

3.4 Struct Values and their Pointers

In C, data structures built from **struct** types are actually values that can be freely copied via assignment or parameter passing. Furthermore, variables of struct type are actually stack-allocated. To build an object in the heap, we must use a pointer to a struct type whose struct value can then be allocated in the heap. From the view-point of analysis, there is a need to distinguish struct values from pointers to such values, as the latter causes aliasing while the former are allowed to be freely copied.

As an example, consider a struct type of the following form:

```
typedef struct Pair {int x; int y} Pair;
```

Let us declare two variables **Pair** **p**, ***x** where **p** denotes a struct value, while **x** is a pointer to such a value.

As variable **p** captures a struct value (in the stack), we can use the following formula for its initial state where $\perp_{\mathbf{t}}$ denotes an uninitialised value of type **t**. If needed, uninitialised values can be changed to some expected defaults depending on implementation.

```
p = Pair( $\perp_{\text{int}}$ ,  $\perp_{\text{int}}$ )
```

Correspondingly, since **x** is a pointer, we may capture its initial state as:

```
x =  $\perp_{\text{ptr}}$ 
```

Subsequently, a struct value may be allocated in the heap by the following assignment command.

```
x = new Pair(1, 2)
```

With this allocation, the object being built (in the heap) can now be modelled by the following formula.

```
x::ptr(Pair(1, 2))
```

This formula captures a pointer to a previously heap-allocated struct value. It is important as it allows us to track must-aliasing that can be used to support more precise program analysis. However, struct values may also be stack allocated with their pointers accessible via the address-of operator, denoted by **&**. This possible challenge and its solution shall be elaborated next.

3.5 Removing the Address-Of Operators

One of the peculiarities of C is that struct values can be allocated on the stack. Though such struct values are typically captured by variables, there is also the possibility of creating a pointer to such a struct value by the address-of operator $\&$. For example, it is legal to use the following assignment to point to a stack-allocated struct value of variable p itself.

```
x:=&p
```

The presence of such pointers to stack-allocated struct values induces aliasing to stack locations that must be taken into account by program analyses. Moreover, such references to stack may also lead to dangling pointers if these references were to escape from their corresponding method body. Such violation should also be detected by program analysis, if possible.

As a uniform solution to analysing pointers into stack-allocated values, we propose to heap-allocate each such value that may have a reference to it via the address-of operator. We refer to each variable (or parameter) that has this property as an *addressible variable*. Our approach covers values of any types, including primitive and struct types. As a simple example, consider the following code $\{\text{Pair } p; e\}$ where $\&p$ occurs inside e . To heap-allocate p , we propose to change its type to Pair^* and transform our earlier code to $\{\text{Pair}^* p = \text{new Pair}; \rho e; \text{free}(p)\}$ where $\rho = [\&p \mapsto p, p \mapsto *p]$. Note that we have created a heap object for p at the start of the code block, change all codes for $\&p$ and p to p and $*p$, respectively. At the end of code block, we also free the previously heap-allocated object, so that access to dangling references could be detected.

This uniform approach allows all pointers to be analysed as pointers to heap-allocated objects, but may cause some performance deficiency since heap-allocated objects are typically more costly to implement than stack-allocated ones. Nevertheless, we propose to use the above translation for program analysis purpose *only*, and do not recommend it for compilation use. To achieve this we may selectively provide two versions of some program codes using the notation $(e_1)\{e_2\}$ in which e_1 is to be used by analysis, while e_2 is to be used for implementation. For simplicity, we shall ignore issues related to implementation code but shall focus on only code for analysis purpose in the rest of this paper.

Formally, we can handle the translation to remove address-of operator for variables by the following rule.

$$\frac{\text{not}(\text{isArr}(t)) \quad \&v \in e \quad \rho = [\&v \mapsto v, v \mapsto *v]}{\{t \ v; e\} \Longrightarrow \{t \ v; v := \text{new } t; \rho \ e; \text{free } v\}}$$

For non-array variable $t \ v$, we heap allocate it if $\&v$ is ever found in the code block e . Handling of array type is elaborated in Sec 3.6.

The address-of operation may also be applied to the field of some struct values via expressions of the form $\&(e.f)$. Our solution uses a global analysis to identify all fields of each struct type in the entire program that may have the address-of operation taken. We refer to such fields as *addressible fields*. After such a global analysis, we will transform each struct type t whose field $f_i:t_i$ is addressible to a corresponding type $t[f_i:t_i \mapsto f_i:(t_i^*)]$ to allow each such pointer

field to be heap-allocated. This change is recursively applied for all fields of each struct type.

The global analysis to identify such addressible fields is simple, as we are only looking for syntactic occurrences of $\&(\mathbf{w.f}_i)$ so as to add $\mathbf{f}_i:\mathbf{t}_i$ to $\mathbf{F}(\mathbf{t})$ assuming $\mathbf{w}:\mathbf{t}$. Each collection $\mathbf{F}(\mathbf{t})$ is simply a set of addressible fields for each struct type \mathbf{t} . Once this global analysis is done, we shall change each type \mathbf{t} to $\mathbf{t}[\mathbf{f}_i:\mathbf{t}_i^*]^*$, whenever $\mathbf{F}(\mathbf{t}) = \{\mathbf{f}_i:\mathbf{t}_i\}^*$. Subsequently, each $\&(\mathbf{w.f}_i)$ and $(\mathbf{w.f}_i)$ shall be transformed to $\mathbf{w.f}_i$ and $*(\mathbf{w.f}_i)$, respectively, assuming that $\mathbf{f}_i \in \mathbf{F}(\mathbf{t})$. This translation is effected by the following two rules:

$$\frac{G(w)=t \quad f \in F(t)}{\&(w.f) \Longrightarrow w.f} \quad \frac{G(w)=t \quad f \in F(t)}{w.f \Longrightarrow *(w.f)}$$

Let $\mathbf{field}(\mathbf{t})$ returns all immediate fields of a struct type \mathbf{t} . Let $\mathbf{F}(\mathbf{t})$ returns all addressible fields based on a global analysis.

As struct types are nested, for each type \mathbf{t} , we shall also generate a set of addressible labels that would have to be simultaneously created (or freed) in conjunction with the object. This set of addressible labels is formally defined as:

$$\mathbf{L}(\mathbf{t}) = \mathbf{F}(\mathbf{t}) \cup \{\mathbf{f.l} \mid \mathbf{f}:\mathbf{t}_f \in (\mathbf{fields}(\mathbf{t}) - \mathbf{F}(\mathbf{t})), \mathbf{l} \in \mathbf{L}(\mathbf{t}_f)\}$$

As an example consider:

```
struct Rect {
    Pt fst;
    Pt snd;
} r;
struct Pt {
    int x;
    int y;
} *p;
```

The immediate fields of the above struct types are:

$$\begin{aligned} \mathbf{fields}(\mathbf{Pt}) &= \{\mathbf{x}, \mathbf{y}\} \\ \mathbf{fields}(\mathbf{Rect}) &= \{\mathbf{fst}, \mathbf{snd}\} \end{aligned}$$

Let us assume that $\&(\mathbf{r.fst})$ and $\&(*\mathbf{p.y})$ were found by global analysis. Under this scenario, the addressible fields would be:

$$\begin{aligned} \mathbf{F}(\mathbf{Pt}) &= \{\mathbf{y}\} \\ \mathbf{F}(\mathbf{Rect}) &= \{\mathbf{fst}\} \end{aligned}$$

From the above, we can now compute the addressible labels, as follows:

$$\begin{aligned} \mathbf{L}(\mathbf{Pt}) &= \{\mathbf{y}\} \\ \mathbf{L}(\mathbf{Rect}) &= \{\mathbf{fst}, \mathbf{snd.y}\} \end{aligned}$$

We are also required to change each **new** and each **free** construct to take into account the heap-allocated fields, as follows:

$$\frac{\mathbf{L}(t) = \{l_1 : t_1, \dots, l_n : t_n\} \quad \{w.l_1 := \mathbf{new} \ t_1; \dots; w.l_n := \mathbf{new} \ t_n\} \Longrightarrow e}{w := \mathbf{new} \ t \Longrightarrow \{w := \mathbf{new} \ t; e\}}$$

$$\frac{G(w)=t \quad L(t) = \{l_1 : t_1, \dots, l_n : t_n\} \quad \{free(w.l_1); \dots; free(w.l_n)\} \implies e}{free(w) \implies \{e; free(w)\}}$$

Furthermore, each stack-allocated variable shall be forced to have its addressible fields to be heap-allocated at the start of its code block, but shall be deallocated prior to exit of the block, as follows:

$$\frac{\begin{array}{l} L(t) = \{l_1 : t_1, \dots, l_n : t_n\} \\ \{v.l_1 := new\ t_1; \dots; v.l_n := new\ t_n\} \implies e_1 \\ \{free(v.l_1); \dots; free(v.l_n)\} \implies e_2 \end{array}}{\{t\ v; e\} \implies \{t\ v; e_1; e; e_2\}}$$

The deallocation of addressible fields for stack-allocated variables is to mimic the lost of struct values once we exit from the corresponding stack frame. To allow all addressible labels to persists for the entire method, we shall float all data declarations to the outermost scope, as follows:

$$\frac{fresh\ u}{\dots\{t\ v; e\}\dots \implies \{t\ u; \dots[v \mapsto u]e\dots\}}$$

3.6 Translating Array Access to Pointer Arithmetic

Local arrays in C are stack-allocated. To make it uniform for analysis, we propose to translate them to heap allocation, as follows:

$$\frac{\rho = [\&v \mapsto v, v[i] \mapsto *(v + i), \&v[i] \mapsto (v + i)]}{\{t[n]\ v; e\} \implies \{t* v; v := new\ t[n]; \rho\ e; free(v)\}}$$

Arrays may also appear as the last field of a struct. To make it easier for analysis, we shall heap-allocate such fields. This can be achieved by adding $\mathbf{f}_i : \mathbf{t}_i$ to $\mathbf{F}(\mathbf{t})$ whenever \mathbf{t}_i is of the array type.

As a result, we can convert each array access into its equivalent pointer arithmetic, as follows:

$$\frac{G(w.f) = t[n] \quad \rho = [\&w.f \mapsto w.f, w.f[i] \mapsto *(w.f + i), \&w.f[i] \mapsto (w.f + i)]}{e \implies \rho\ e}$$

Furthermore, we shall allow array pointer to be either $\mathbf{t}[n]$ or $\mathbf{t}*$. The former is useful for allocation (and deallocation), while the latter easily supports array accesses. Internally, an array pointer \mathbf{p} shall be captured as a pure expression to a heap-allocated object \mathbf{a} , in the following format:

$$\mathbf{p} = \mathbf{Aptr}(\mathbf{a}, i) \wedge \mathbf{a}::\mathbf{ptr}(\mathbf{Arr}(\mathbf{bnd}, \mathbf{v}^*))$$

Apart from an upper bound \mathbf{bnd} , it is also possible to capture some information on the contents in the array object which is currently left unspecified as \mathbf{v}^* .

For array allocation and deallocation, we shall perform the corresponding allocation and deallocation for the addressible fields of its components.

$$\frac{\text{for } i=0 \text{ to } c-1 \{*(w+i.l_1) := \text{new } t_1; \dots; *(w+i.l_n) := \text{new } t_n\} \implies e}{\begin{array}{c} L(t) = \{l_1 : t_1, \dots, l_n : t_n\} \\ w := \text{new } t[c] \implies \{w := \text{new } t[c]; e\} \\ G(w) = t[c] \quad L(t) = \{l_1 : t_1, \dots, l_n : t_n\} \\ \text{for } i=0 \text{ to } c-1 \{free(*(w+i.l_1)); \dots; free(*(w+i.l_n))\} \implies e \\ \hline free(w) \implies \{e; free(w)\} \end{array}}$$

Pointer arithmetic can be performed with the help of the following: primitives:

```
void add(t[] v, int i, ref t[] res)
  v=Aptr(a,j) *-> res'=Aptr(a,i+j)
```

In addition, array accesses are carried out via `*v.l`. This can be dealt with via Hoare-style reasoning rule. Depending on the type of $v : t^*$, we expect the following set of state transitions to be effected:

```
v:t^*      a::ptr<Arr(bnd)>& v=Aptr(a,i) & 0<=i<bnd
*-> a::ptr<Arr(bnd)> & res=\top
```

```
v:int^*    a::ptr<Arr(bnd,s,b)>& v=Aptr(a,i) & 0<=i<bnd
*-> a::ptr<Arr(bnd,s,b)> & s<=res<=b
```

Array update is carried out by assignment of the form `*v.l:=w`. We would expect the following state transitions to be used:

```
v:t^*      a::ptr<Arr(bnd)> & v=Aptr(a,i) & w=\top & 0<=i<bnd
*-> a::ptr<Arr(bnd)>
```

```
v:int^*    a::ptr<Arr(bnd,s,b)> & v=Aptr(a,i) & 0<=i<bnd
*-> a::ptr<Arr(bnd,min(s,w),max(b,w))
```

3.7 Translating malloc to new

```
malloc sizeof(t) => new t
malloc n*sizeof(t) => new t[n]
w:t & malloc sizeof(w) => new t
```

. "new" command shall be implemented by malloc. I suppose no argument is needed, except for array dimension

```
sizeof struct
```

The additional case that I encountered in the implementation:
`w:t* & w=(t*)malloc(kexp) => new t[l]`, where $l=kexp/sizeof(t)$

A special case is when the lhs of the assignment has type void*:
w:void* & w=(void*)malloc(kexp) => new byte[l], where l=kexp

3.8 Casting

Cast could only be made to scalar values, e.g. primitive or pointers. Redundant casting shall be removed:

```
G(w)=t
-----
(t)w => w
```

In case of pointer, we may lose mutable information:

```
G(w)=t* D |- w=\top * D1, F1
-----
{D} (t)w {D1,F1}
```

```
G(w)=t[] D |- w=Ptr<a,i> * a::ptr<Arr(bnd,\top^*) * D1, F1
-----
{D} (t)w {D1,F1}
```

```
isPrim(t)
-----
{D} (t)w {D & res=\top,\emp}
```

3.9 Other Issues

- . why is pass-by-ref needed? This is an in-out mechanism that is useful for extension to C# It can be used to handle result parameter and also while loop conversion.
- . how to handle union type?
- . compiler should rule out:
list = { int v, list next}
but allows:
list = { int v, list *next}
- . STATIC vars in methods persist and will be translated to global vars.

```
. pointer arithmetic property of struct location, esp
  of addressible fields should be captured
```

```
how to deal with initializer?
- use multiple assignments
// below is more for initializer
G(w)=t    t={f1:t1,..,fn:tn}
i in 1..n : G|- w.fi := e.fi ==> ci
-----
G |- w := e => {c1..cn}

G(w)=t    scalar(t)
-----
G |- w := e => w := e
```

```
. code for analysis + implementation
dual purpose code using e1[e2] where e1
is for analysis while e2 is for implementation
```

```
transform goto/label
  into loop
```

```
transform loop/break/return
  into tail-recursion
```

```
post-processing to C
```

```
string is "array of char"
```

4 Alias Analysis

Traditionally, alias analyses are based on a flow-insensitive may-alias analysis which can be rather imprecise. Furthermore, they are mostly based on the points-to graph. The use of points-to graph is related to the presence of the address-of operator for which either a variable, a field or a heap-allocated object may be the target of a points-to relation.

As we have endeavoured to remove the address-of operator in our code, we are no longer dependent on the points-to graph as the underlying representation of alias analysis. Instead, we shall use equality (e.g. $x=y$, $x=null$) and disequality (e.g. $x\neq y$, $x\neq null$) to directly capture aliasing and non-aliasing relationships, as well as the nullness property for pointers.

On its own, conjunctive formulae of the above form can be used to support must-alias analysis. In contrast, may-alias analysis can be supported by disjunctive formulae. For example $x=p \vee x=q$ indicates that x may alias with either p or q . For brevity, we shall use the set notation $x \in \{p, q\}$ to denote the same may-alias property.

To capture alias properties precisely, we shall also turn to separation logic notation to describe heap state in a concise and unambiguous manner. As an example, consider the following declarations:

```
int* x, y; struct Pair{int* f, int* s} * p;
```

If x and y points to distinct locations with an integer value each, we may represent this state by the following formula:

```
x::ptr(i) * y::ptr(j)
```

Alternatively, if x and y are aliased, we would capture it with a different heap-state instead:

```
x::ptr(i)  $\wedge$  x=y
```

As another example, we may have a pointer into a `Point` object (made of a struct value) where its fields are unaliased, as follows:

```
p::ptr(Point(a, b)) * a::ptr(i) * b::ptr(j)
```

Alternatively, if its two fields are pointers to the same integer location, we would model this must-aliasing of its fields, as follows:

```
p::ptr(Point(a, b)) * a::ptr(i)  $\wedge$  a=b
```

Disjunctive formula may also be used to capture alternative heap states which may arise from different branches of conditional constructs. For example, $\phi_1 \vee \phi_2$ indicates that the current heap state is either ϕ_1 (say from one branch) or ϕ_2 (say from another branch).

One of the nice things about separation logic is that it facilitates local reasoning. For each method, we can describe the expected heap state of its code by a precondition, and can also summarise the effects of the code with a postcondition. An example is the following where a special output parameter `res` is used to capture a freshly allocated heap node for the result of the method.

```
void foo(ref t res) true *-> res'::\ptr{t(..)} ;
{ res:=new t }
```

It is also possible to accurately model methods that never access any heap location. The following are two examples:

```
void foo( t v, ref t res) true *-> res'=v;
{ res=v }
void foo(ref t a, ref t b)
true *-> a'=b & b'=a
{t tmp; tmp=b; b=a; a=tmp }
```

Note that a', b' denote the values of the two `ref` parameters at the post-state, while a, b denote their initial values. The formula $a'=b \wedge b'=a$ captures a swapping of the two `ref` parameters.

Should accesses to heap location occur in a method, we must be explicit about the expected heap state in its precondition. An example is shown below where a pointer v has been dereferenced to access a value that is being passed to the output parameter.

```
void foo( t* v, ref t res)
  v::\sptr{a} *-> v::\sptr{a} {\wedge} res'=a ;
{ res=*v }
```

Take note that the postcondition captures an unchanged heap state in addition to the relation $\text{res}' = \text{a}$.

Furthermore, if a piece of code uses two locations of the same type, there is a possibility that the two locations are either aliased or unaliased. To capture both possibilities, we propose to use an intersection type of multiple pre/post conditions. An example is shown below:

```
void foo(t* a, t* b)
  a{::}\sptr{x}{\str}b{::}\sptr{y} *-> a{::}\sptr{y} * b{::}\sptr{x}
/\ a{::}\sptr{x} {\wedge} a{=} *-> a{::}\sptr{x}
{t tmp; tmp=*b; *b=*a; *a=tmp }
```

Note that $\text{a}::\text{ptr}\langle x \rangle * \text{b}::\text{ptr}\langle y \rangle$ indicates that a, b are unaliased, while formula $\text{a}::\text{ptr}\langle x \rangle \wedge \text{a}=\text{b}$ indicates that a, b are aliased

4.1 Forward Rules with Footprint Instantiation

The separation constraints are in disjunctive normal form. Each disjunct consists of a heap part κ , and a pure part. The pure part represents pointer equality-inequality γ and Presburger arithmetic ϕ .

$$\begin{aligned} \Sigma &::= \bigvee ([v^*]\kappa \rightsquigarrow \Phi)^* && \text{before normalization} \\ \sigma &::= ([v^*]\Phi \rightsquigarrow \Phi) && \text{after normalization} \\ \Phi &::= \bigvee (\exists v^*. \kappa \wedge \gamma)^* \\ \kappa &::= \text{emp} \mid \text{v}::\text{ptr}\langle v \rangle \mid \text{v}::\text{Arr}\langle v_1, v_2 \rangle \mid k_1 * k_2 \\ \gamma &::= v_1=v_2 \mid v_1 \neq v_2 \mid v=\text{null} \mid v \neq \text{null} \mid v=\text{Aptr}\langle v_1, v_2 \rangle \mid v=\text{sn}(v^*) \mid v=\text{top} \mid \gamma_1 \wedge \gamma_2 \end{aligned}$$

where

- $\text{v}::\text{Arr}\langle v_1, v_2 \rangle$ - v points to a heap-allocated array with the size v_1 and v_2 abstracting the array elements.
- $\text{v}=\text{Aptr}\langle v_1, v_2 \rangle$ - v is a pointer obtained via pointer arithmetic from the base pointer v_1 and the offset v_2 .
- $\text{v}=\text{sn}(v^*)$ - v is a struct value of the type defined by $(\text{struct sn}\{..\})$.

The judgement used by the forward rules:

$$\{\phi_1 \rightsquigarrow \phi_2\} e \{\phi'_1 \rightsquigarrow \phi'_2\}$$

The entailment as used by the forward rules:

$$\sigma_a \vdash \phi_c * \sigma_r$$

$\frac{}{\{\phi\} k \{\phi \wedge \mathbf{res}=k\}} \quad \text{[CONST]}$	$\frac{}{\{\phi\} v \{\phi \wedge \mathbf{res}=v'\}} \quad \text{[VAR]}$
$\frac{G(v)=t* \quad \phi \vdash v'::\mathbf{ptr}\langle v_1 \rangle * \phi_1}{\{\phi\} *v \{(v'::\mathbf{ptr}\langle v_1 \rangle * \phi_1) \wedge \mathbf{res}=v_1\}} \quad \text{[*VAR]}$	$\frac{}{\{\phi\} v.l \{\phi \wedge \mathbf{res}=v'.l\}} \quad \text{[FLD]}$
$\frac{G(v)=t* \quad \phi_1 = \mathit{init}(v', t*) \quad \phi \vdash \phi_1 * \phi_2}{\{\phi\} *v.l \{\phi_1 * \phi_2 \wedge \mathbf{res}=(v'.l)\}} \quad \text{[*FLD]}$	$\frac{G(w)=t* \quad \phi \vdash w::\mathbf{top} * \phi_1}{\{\phi\} (t)w \{w::\mathbf{top} * \phi_1\}} \quad \text{[CAST]}$
$\frac{\text{fresh } u, v^* \quad \phi_1 = \mathbf{res}::\mathbf{ptr}\langle u \rangle \wedge u=t(v^*) \wedge (v=\perp)^*}{\{\phi\} \mathbf{new } t \{\phi * \phi_1\}} \quad \text{[NEW]}$	$\frac{\text{fresh } a \quad \phi_1 = \mathbf{a}::\mathbf{Arr}\langle n, \perp_t \rangle \wedge \mathbf{res}=\mathbf{Aptr}\langle a, 0 \rangle}{\{\phi\} \mathbf{new } t[n] \{\phi * \phi_1\}} \quad \text{[NEW-ARR]}$

Fig. 1. Forward reasoning rules - Expressions

We intend for pre/post conditions for alias analysis to be automatically inferred. For automated program analysis, we could use Hoare-style triple of the form $\{\phi_1\} \mathbf{code} \{\phi_2\}$ to allow both forward and backward reasoning to be carried out with the help of separation logic. In the forward direction, when given an initial heap-state ϕ_1 , we expect to derive ϕ_2 as its post-state. To allow also backward reasoning of heap-state, we propose to use a recently introduced technique, based on footprint analysis [1], where variables from the original input state may be *instantiated*, where needed.

To identify the set of input variables that can be backward instantiated, we shall also maintain a set of footprint variables together with each program state. Intuitively, this set of variables is simply the original parameters together with those that arise from subsequent instantiations, but with the already instantiated variables removed. The set of rules is given in Fig. 1, 2 and 3.

To handle the presence of multiple returns in the method body, we will use a modified form of a Hoare triple: $\{\phi\} \mathbf{code} \{\phi_{\mathbf{norm}}, \phi_{\mathbf{ret}}\}$. The idea, borrowed from [2], is to use both a normal poststate $\phi_{\mathbf{norm}}$, as well as a return poststate $\phi_{\mathbf{ret}}$. For a return statement, the normal poststate is **false**. The return poststate in the [SEQ] and [IF] rules represent a disjunction from the two subexpressions. When needed, the standard Hoare triple form and can be extended to include a return poststate of **false**: $\{\phi_1\} \mathbf{code} \{\phi_2\} \equiv \{\phi_1\} \mathbf{code} \{\phi_2, \mathbf{false}\}$.

To process each method, we perform dual-mode processing on the body of the method using the initial parameters as footprint variables that can be instantiated. In the case of recursive methods, we also provide a fixpoint abstraction that is elaborated later in Sec ??.

$\frac{\boxed{\text{VAR-AS}} \quad \{\phi\} e \{ \phi_1 \} \text{ fresh } u}{\{\phi\} v := e \{ \exists \text{res}, u \cdot \phi_1[v'/u] \wedge v' = \text{res} \}}$	$\frac{\boxed{* \text{VAR-AS}} \quad G(v) = t* \quad \{\phi\} e \{ \phi_1 \} \quad \phi_1 \vdash v' :: \text{ptr}(v_1) * \phi_2}{\{\phi\} *v := e \{ \exists \text{res} \cdot (v' :: \text{ptr}(\text{res}) * \phi_2) \}}$
$\frac{\boxed{\text{FLD-AS}} \quad \{\phi\} e \{ \phi_1 \}}{\{\phi\} v.l := e \{ \exists \text{res} \cdot \phi_1 \wedge v'.l := \text{res} \}}$	$\frac{\boxed{* \text{FLD-AS}} \quad G(v) = t* \quad \{\phi\} e \{ \phi_1 \} \quad \phi_2 = \text{init}(v', t*) \quad \phi_1 \vdash \phi_2 * \phi_3}{\{\phi\} *v.l := e \{ \exists \text{res} \cdot \phi_2 * \phi_3 \wedge (*v').l := \text{res} \}}$

Fig. 2. Forward reasoning rules - Assignment

$\frac{\boxed{\text{SEQ}} \quad \frac{\{\phi\} s_1 \{ \phi_1, \phi_{r1} \} \quad \{\phi_1\} s_2 \{ \phi_2, \phi_{r2} \}}{\{\phi\} s_1; s_2 \{ \phi_2, \phi_{r1} \vee \phi_{r2} \}}}{\{\phi\} s_1; s_2 \{ \phi_2, \phi_{r1} \vee \phi_{r2} \}}$	$\frac{\boxed{\text{CALL}} \quad \text{void } mn(([\text{ref}] t v)^*) \phi_{pr} * \mapsto \phi_{po} \{e\} \quad \rho = [(v \mapsto \phi(w))^*] \quad \phi \vdash \rho \phi_{pr} * \phi_1}{\{\phi\} mn(w^*) \{ \phi_1 * \rho \phi_{po}, \text{false} \}}$
$\frac{\boxed{\text{IF}} \quad \frac{\{\phi \wedge v'\} s_1 \{ \phi_1, \phi_{r1} \} \quad \{\phi \wedge \neg v'\} s_2 \{ \phi_2, \phi_{r2} \}}{\{\phi\} \text{if } x \text{ then } s_1 \text{ else } s_2 \{ \phi_1 \vee \phi_2, \phi_{r1} \vee \phi_{r2} \}}}{\{\phi\} \text{if } x \text{ then } s_1 \text{ else } s_2 \{ \phi_1 \vee \phi_2, \phi_{r1} \vee \phi_{r2} \}}$	$\frac{\boxed{\text{FREE}} \quad G(w) = t* \quad \phi \vdash w :: \text{ptr}(v_1) * \phi_1}{\{\phi\} \text{free}(w) \{ \phi_1, \text{false} \}}$
$\frac{\boxed{\text{RETURN}}}{\{\phi\} \text{return} \{ \text{false}, \phi \}}$	$\frac{\boxed{\text{LOCAL}} \quad \{\phi \wedge v' = \perp_t\} s \{ \phi_1, \phi_{r1} \}}{\{\phi\} t v; s \{ \exists v' \cdot \phi_1, \exists v' \cdot \phi_{r1} \}}$

Fig. 3. Forward reasoning rules - Other statements

$$\frac{(\phi, F) = \text{initParams}((t, v)^*) \quad \{\phi\} (t l;)^* s \{ \phi_1, \phi_r \} \quad (\phi_{pr}, \phi_{po}) = \text{fixpt}(\phi_r)}{\text{void } mn(([\text{ref}] t v)^*) \{ (t l;)^* s \} \implies \text{void } mn(([\text{ref}] t v)^*) \phi_{pr} * \mapsto \phi_{po} \{ (t l;)^* s \}}$$

At the start of each analysis, there is a need to generate the default prestate of each method. This generation is carried out by the `initParams` procedure in Fig. 4.

Note that the default generation also takes into account the instantiation of each addressible field, whose heap-state is recursively processed.

As an example, let us consider the prestate generated for a struct value of `Rect` type. Let us assume, that the `Rect` type has two addressible fields as shown in sec 3.5.

$$\text{init}(\text{struct Rect}) = (\text{Rect}(f_0, \text{Pt}(f_3, f_4)), f_0 :: \text{ptr}(\text{Pt}(f_5, f_6)) * f_4 :: \text{ptr}(f_7))$$

$$\begin{array}{l}
\text{initParams}(\[]) = (\mathbf{emp}, \{\}) \\
\text{initParams}((t, v) : r) = \text{let } (\phi_1, V_1) = \text{initParam}(v, t); (\phi_2, V_2) = \text{initParams}(r) \\
\quad \text{in } (\phi_1 * \phi_2, V_1 \cup V_2) \\
\text{initParam}(v, t^*) \Longrightarrow (v' = v, \{v\}) \\
\\
\frac{\neg(\text{IsStruct}(t))}{\text{initParam}(v, t) = (v' = v, \{v\})} \qquad \begin{array}{l} \text{vars}(k_1 * k_2) = \text{vars}(k_1) \cup \text{vars}(k_2) \\ \text{vars}(v::p) = \text{vars}(p) \\ \text{ivars}(v::p) = \{v\} \\ \text{ivars}(k_1 * k_2) = \text{ivars}(k_1) \cup \text{ivars}(k_2) \end{array} \\
\\
\frac{\text{IsStruct}(t) \quad \text{init}(t) = (p, \phi) \quad V = \text{vars}(\phi) \cup \text{vars}(p) - \text{ivars}(\phi)}{\text{initParam}(v, t) = ((\phi \wedge v' = v \wedge v = p), V)} \\
\\
\frac{\neg(\text{IsStruct}(t)) \quad \text{fresh } v}{\text{init}(t) \Longrightarrow (v, \mathbf{emp})} \\
\\
\frac{\text{IsStruct}(t) \quad \text{fresh } v^* \quad L(t) = \{l_i : t_i^*\} \quad \text{init}(v_i, t_i^*) \Longrightarrow \phi_i, i \in 1..n}{\text{init}(t) \Longrightarrow (t(v^*), \phi_1 * \dots * \phi_n)} \\
\\
\frac{\text{init}(t) \Longrightarrow (p, \phi)}{\text{init}(v, t^*) \Longrightarrow \mathbf{v}::\mathbf{ptr}\langle \mathbf{p} \rangle * \phi}
\end{array}$$

Fig. 4. Auxiliary rules

The update operations on heap formulae are implemented as follows:

$$\begin{array}{ll}
(\phi \wedge v := r) & \Rightarrow (\exists u \cdot \phi[v/u] \wedge v = r) \\
(\phi \wedge v = P \wedge v.l := r) & \Rightarrow (\phi \wedge v = P[l/r]) \\
(\phi * \mathbf{v}::\mathbf{ptr}\langle \mathbf{P} \rangle \wedge (*v).l := r) & \Rightarrow (\phi * \mathbf{v}::\mathbf{ptr}\langle \mathbf{P}[l/r] \rangle)
\end{array}$$

4.2 Dual-Mode Entailment

Our Hoare-style reasoning rules use an entailment procedure of the following form:

$$\sigma_a \vdash \phi_c * \sigma_r$$

where the inputs are an (instantiateable) constraint as the antecedent σ_a , and the consequent ϕ_c . The output is σ_r for residual heap state (as a result of frame rule) which may instantiate the footprint further as needed.

The key entailment rule is [ENT-INST] which allows instantiation of footprint variables. The rule is triggered when the consequent requires \mathbf{v} to point to a node missing from the antecedent, and \mathbf{v} is a footprint variable that can be

instantiated. In this situation, the footprint is further instantiated to $(v::c\langle w^* \rangle \rightsquigarrow \sigma_r)$.

$$\frac{\text{[ENT-INST]}}{v \in \text{FootVars}(\sigma) \quad \sigma' = \sigma - \{v\} \cup \{w^*\} \quad \sigma' \vdash \phi_1 * \sigma_r \quad \sigma \vdash (v::c\langle w^* \rangle) * \phi_1 * (v::c\langle w^* \rangle \rightsquigarrow \sigma_r)}$$

The core rules for entailment checking are introduced in [3] as shown in Fig 5.

$\frac{\text{[ENT-EMP]}}{\rho = [0/\text{null}] \quad b = (XPure_n(\kappa_1 * \kappa \wedge \pi_1) \implies \exists V. \rho \pi_2) \quad \kappa_1 \wedge \pi_1 \vdash_V^\kappa (\pi_2) * \{\kappa_1 \wedge \pi_1 \mid b\}}$	$\frac{\text{[ENT-MATCH]}}{XPure_n(p_1::c\langle v_1^* \rangle * \kappa_1 \wedge \pi_1) \implies p_1 = p_2 \quad \rho = [v_1^*/v_2^*] \quad \kappa_1 \wedge \pi_1 \wedge \text{freeEqn}(\rho, V) \vdash_{V - \{v_2^*\}}^{\kappa * p_1::c\langle v_1^* \rangle} \rho(\kappa_2 \wedge \pi_2) * S \quad p_1::c\langle v_1^* \rangle * \kappa_1 \wedge \pi_1 \vdash_V^\kappa (p_2::c\langle v_2^* \rangle * \kappa_2 \wedge \pi_2) * S}$
$\frac{\text{[ENT-LHS-OR]}}{\phi_1 \vdash_V^\kappa \phi_3 * S_1 \quad \phi_2 \vdash_V^\kappa \phi_3 * S_2 \quad S_3 = \{\phi_5 \vee \phi_6 \mid \phi_5 \in S_1, \phi_6 \in S_2\} \quad \phi_1 \vee \phi_2 \vdash_V^\kappa \phi_3 * S_3}$	$\frac{\text{[ENT-RHS-OR]}}{\phi_1 \vdash_V^\kappa \phi_2 * S_1 \quad \phi_1 \vdash_V^\kappa \phi_3 * S_2 \quad S = S_1 \cup S_2 \quad \phi_1 \vdash_V^\kappa (\phi_2 \vee \phi_3) * S}$
$\frac{\text{[ENT-LHS-EX]}}{[w/v] \phi_1 \vdash_V^\kappa \phi_2 * S \quad \text{fresh } w \quad \exists v. \phi_1 \vdash_V^\kappa \phi_2 * S}$	$\frac{\text{[ENT-RHS-EX]}}{\phi_1 \vdash_{V \cup \{w\}}^\kappa ([w/v] \phi_2) * S_1 \quad \text{fresh } w \quad S = \{\exists w. \phi \mid \phi \in S_1\} \quad \phi_1 \vdash_V^\kappa (\exists v. \phi_2) * S}$

Fig. 5. Separation Constraint Entailment

4.3 Abstraction via Summary Nodes

In the case of recursive data structures, such as linked-list, it is possible for an arbitrary long sequence of data nodes to be utilised by some program loop. A simple example of this is the following:

```
typedef struct List {int val, List* next} List;

void goo(ref List* res)
{List* h,p; int n=0;
 h:=new List; *h.val:=n; t:=h;
 while (n<100) do
   *t.next:=new List; n++;
   *t.val:=n;
 end;
 res=h;
}
```

During fixpoint analysis, we will need an abstraction step to summarise multiple `List` nodes into a finite representation. There are a couple of solutions to this summarization problem.

One traditional solution, as used by both shape analysis (e.g. TVLA) and alias analysis work (e.g. [?]), is to use a summary node (that is non-linear) to capture multiple related nodes. As an example, we can capture the poststate of the above code by:

$$\text{res}'::\text{ptr}\langle\text{List}(-, \mathbf{q})\rangle * \mathbf{q}::\text{ptr}\langle\text{List}(-, \mathbf{r})\rangle @M \wedge \mathbf{r} \in \{\mathbf{q}, \text{null}\}$$

The first node is unique but the subsequent nodes of the linked-list are summarised into a non-linear node $\mathbf{q}::\text{ptr}\langle\text{List}(0, \mathbf{r})\rangle @M$ in which the `next` field $\mathbf{r} \in \{\mathbf{q}, \text{null}\}$ is a may alias. The non-linear node marked by `@M` approximates zero or more similar nodes by a single node. However, most pointers into and out of summary nodes are typically imprecise due to may-aliasing. As a result, only weak update could be carried out for the fields of such summary nodes.

A more precise solution is to make use of shape definition that can more accurately capture aliasing property amongst the repeated data nodes. Such definitions appear as recursive predicates in separation logic and can be used to give more precise heap state that mirrors the aliasing pattern of recursive data structures. The poststate of the above example can be captured more precisely using the following formula in separation logic.

$$\text{res}'::\text{ptr}\langle\text{List}(-, \mathbf{q})\rangle * \text{ls}(\mathbf{q}, \text{null})$$

The `ls` predicate can be defined recursively as follows:

$$\text{ls}(\text{self}, \mathbf{p}) \equiv \text{self} = \mathbf{p} \vee \text{self}'::\text{ptr}\langle\text{List}(-, \mathbf{q})\rangle * \text{ls}(\mathbf{q}, \mathbf{p})$$

The above uses recursive predicate `ls` to describe the data structure pattern of linked-list segments. Though this approach is very precise, the methodology is currently under development [?,?] for both forward and backward analysis. At present moment, only a class of linearly-linked data structures is being explored. It remains open whether the methodology is effective and efficient enough to cover a large enough class of practical programs.

We propose also a third approach that loses precision whenever summary nodes are required, but we can cope with all programs and shall use a very simple but scalable technique. Our hypothesis is that the aliasing properties of most programs can be adequately captured without using summary nodes, and that the imprecision from recursive data structures can be largely ignored. Nevertheless, if precision is required for some recursive data structure, we propose to allow users to supply annotations for the expected recursive predicate (in separation logic notation) for use by our analysis. This approach can help us recover precision, where needed, as directed by the user. We emphasize that our approach works for all programs, though with some loss of precision whenever summary nodes are expected. Our basic technique is to approximate each summary node by an unknown \top value. For the above example, we would summarise its poststate using:

$$\text{res}'::\text{ptr}\langle\text{List}(-, \mathbf{q})\rangle * \mathbf{q} = \top$$

The multiple nodes at `q` are simply summarised as \top which denotes an unknown portion of the heap state. As a matter of fact, we can still analyse the

immutable properties of memory state associated with \top but not the mutable properties, as aliasing in the unknown heap \top are untrackable.

To support summary nodes based on unknown heap \top , we shall augment our previous dual-mode reasoning and entailment rules. For reasoning rules, we require two extra rules to deal with field access and field update that may now occur for unknown heaps, as follows:

$$\frac{\phi \vdash v' = \top * \phi_1}{\{\phi\} * v.l \{\phi_1 \wedge \mathbf{res} = \top\}}$$

$$\frac{\{\phi\} e \{\phi_1\} \quad \phi_1 \vdash v' = \top \wedge \mathbf{res} = \top * \phi_2}{\{\phi\} * v.l := e \{\exists \mathbf{res} \cdot \phi_2\}}$$

For the entailment procedure, we must allow normal heap structure to be casted into unknown ones, but not vice-versa. The following rules allow casting to \top for either known heap or those that can be instantiated from footprint variables.

$$\frac{\phi \wedge v = \top \vdash (u = \top)^* \wedge \phi_1 * \phi_2}{\mathbf{v}::\mathbf{ptr}\langle \mathbf{t}(u^*) \rangle * \phi \vdash v = \top \wedge \phi_1 * \phi_2}$$

$$\frac{v \in \mathit{FootVars}(\sigma) \quad \sigma \wedge v = \top \vdash \phi_1 * \sigma_r}{\sigma \vdash v = \top \wedge \phi_1 * (v = \top * \rightarrow \sigma_r)}$$

One more rule for direct matching of \top value is:

$$\frac{v = \top \wedge \phi \vdash \phi_1 * \phi_2}{v = \top \wedge \phi \vdash v = \top \wedge \phi_1 * \phi_2}$$

There is one more step we need to do which is to obtain a more complete set of pre/post conditions. Note that if non-recursive data structures are contained within recursive ones, it is also possible for these non-recursive nodes to be summarised as \top . As a result, we will need to derive a more complete set of pre/post conditions to handle all possible scenarios where \top may occur. Two of our previous examples have some heap nodes in the precondition. We can proceed to derive more scenarios with \top , as follows:

```
void foo( t* v, ref t res)
  v::\sptr{a} *-> v::\sptr{a}\wedge res'=a ;
 /\ v::\top *-> v::\top & res'::\top
 { res=*v }
```

```
void foo(t* a, t* b)
  a::\sptr{x}*b::\sptr{y} *-> a::\sptr{y} * b::\sptr{x}
 /\ a=top & b=top *-> true
 {t tmp; tmp=*b; *b=*a; *a=tmp }
```

The above results can be computed directly from the original pre/post conditions that have been derived, and need not involve another analysis of the program code.

4.4 Predicates to Describe SummaryNodes

We can also use predicate of separation logic to capture the \top value, as follows:
 $\text{Top}() ::= \text{self} == \text{null} \vee \text{self} :: \text{node}(-, r) * r :: \text{Top}()$

However, the above assumes an acyclic list. To cater to cyclic list, we must provide an extra parameter together with a second base case, as follows:

$\text{Top}(S) ::= \text{self} == \text{null} \vee \text{self} \in S \vee \text{self} :: \text{node}(-, r) * r :: \text{Top}(\{\text{self}\} \cup S)$
but this would make it more complex. Furthermore, one possibility of summary nodes is:

$\text{Mnode}(N) ::= \text{self} == \text{null} \wedge N = \{\} \vee \text{self} :: \text{node}(-, r) * r :: \text{Mnode}(N1) \wedge N = r + N1$

N captures the may-be alias of the next field. To make it cover cyclic structures, we probably need:

$\text{Mnode}(N, S) ::= \text{self} == \text{null} \wedge N = \{\} \vee \text{self} \in S \wedge N = \{\} \vee \text{self} :: \text{node}(-, r) * r :: \text{Mnode}(N1, \text{self} \cup S) \wedge N = \{r\} \cup N1$

Furthermore, this summary node covers those nodes that are connected via reachable link. Occasionally, we may want to summarise disconnected nodes. This would require the following extra scenario.

$\text{Mnode}(N, S) ::= \text{self} :: \text{node}(-, r) * q :: \text{Mnode}(N1, \text{self} \cup S) \wedge N = \{q\} \cup N1$

4.5 Fixpoint Analysis

Given the following code:

```
typedef struct List {int val, List* next} List;

void goo(ref List* l)
{List* t;
 while (l!=null) do
   t:= *l.next
   free(l);
   l:=t
 end
}
```

We first transform to tail recursion:

```
void goo(ref List* l)
{List* t;
 if (l!=null) then
```

```

    t:= *l.next
    free(l);
    l:=t
    goo(l)
  else ()
}

```

After analysis via dual-mode forward reasoning, we shall built the following constraint abstraction:

```

G(l,l',R,F) :- R=(l=null & l'=null) & F=(l=null)
  \/\ ex n,l2,F1. F1=l::ptr<List(_,n)>
    & R=(l2=n)*R2 & G(l2,l',R2,F2) & F=F1*F2

```

Note that R is residual state while F is the footprint of the prestate.

```

G(l,l') :- l=null *-> (l=null & l'=null)
  \/\ ex n,l2. let F1=l::ptr<List(_,n)>
    F2*->R2 = G(l2,l') in
    F1*F2 *-> (l2=n)*R2

```

```

G(l,l') :- l=null *-> (l=null & l'=null)
  \/\ ex n,l2. let F1=l::ptr<List(_,n)> & l2=n
    F2*->R2 = G(l2,l') in
    F1*F2 *-> R2

```

```

A *-> B \/\ C *-> D
<==> A\/\C *-> B\/\C

```

=====

```

G(l,l') :- false

```

```

G(l,l') :- l=null *-> (l=null & l'=null)

```

```

G(l,l') :- l=null *-> (l=null & l'=null)
  \/\ ex n,l2. let F1=l::ptr<List(_,n)> & l2=n in
    F1*(l2=null) *-> (l2=null & l'=null)

```

```

G(l,l') :- l=null *-> (l=null & l'=null)
  \/\ ex n,l2. l2=n in
    l::ptr<List(_,null)> *-> (l'=null)

```

```

G(1,l') :- l::ptr<List(_,null) *-> l'=null

G(1,l') :- l=null *-> (l=null & l'=null)
  \/ ex n,l2. let F1=l::ptr<List(_,n)> & l2=n in
      F1*l2::ptr<List(_,null) *-> l'=null

```

```

=====
Fixpoint analysis via top
=====

```

```
G(..):- false
```

```
G(1,l',R,F) :- R=(l=null & l'=null) & F=(l=null)
```

```
G(1,l',R,F) :- R=(l=null & l'=null) & F=(l=null)
  \/ ex n,l2,F1. F1=l::ptr<List(_,n)>
    & R=(l2=n)*R2 & (R2=l2=null & l'=null & F2=(l2=null))
    & F=F1*F2

```

```

  :- R=(l=null & l'=null) & F=(l=null)
  \/ ex n,l2,F1.
    & R=((l2=n) & l2=null & l'=null)
    & F=l::ptr<List(_,null)> &

```

```
  :- R=(l'=null) & F=l=top
```

```
G(1,l',R,F) :- R=(l=null & l'=null) & F=(l=null)
  \/ ex n,l2,F1. F1=l::ptr<List(_,n)>
    & R=(l2=n)*R2 & (R2=l'=null & F2=(l2=top))
    & F=F1*F2

```

```
  :- R=(l'=null) & F=l=top
```

```

the derived pre/post for good is :
l=top *--> (l'=null)

```

```

=====
Fixpoint analysis via summary nodes
=====

```

```
G(..):- false
```

```
G(1,l',R,F) :- R=(l=null & l'=null) & F=(l=null)
```

```

G(l,l',R,F) :- R=(l=null & l'=null) & F=(l=null)
  \/ ex n,l2,F1. F1=l::ptr<List(_,n)>
    & R=(l2=n)*R2 & (R2=l2=null & l'=null & F2=(l2=null))
    & F=F1*F2

G(l,l',R,F) :- R=(l=null & l'=null) & F=(l=null)
  \/ ex n,l2,F1.
    & R=((l2=n) & l2=null & l'=null)
    & F=l::ptr<List(_,null)> &

G(l,l',R,F) :- R=(l'=null) & F=l::ptr<List(_,null)>@M

G(l,l',R,F) :- R=(l=null & l'=null) & F=(l=null)
  \/ ex n,l2,F1. F1=l::ptr<List(_,n)>
    & R=(l2=n)*R2 & (R2=(l'=null) & F2=l2::ptr<List(_,null)>@M) & F=F1*F2

  :- R=(l=null & l'=null) & F=(l=null)
  \/ ex n,l2,F1. F=l::ptr<List(_,n)>*n::ptr<List(_,null)>@M
    & R=(l2=n) & (R2=(l'=null))

  :- R=(l=null & l'=null) & F=(l=null)
  \/ ex n,l2,F1. F=l::ptr<List(_,l,null)>@M
    & (R2=(l'=null))

  :- R=(l'=null) &
    F=l::ptr<List(_,l,null)>@M

G(l,l',R,F) :- R=(l=null & l'=null) & F=(l=null)
  \/ ex n,l2,F1. F1=l::ptr<List(_,n)>
    & R=(l2=n)*R2 &
    R2=(l'=null) & F2=l2::ptr<List(_,l2,null)>@M
    & F=F1*F2

  :- R=(l'=null) & F=(l=null)
  \/ ex n,l2,F1.
    & R=(l2=n)*(l'=null) &
    & F=l::ptr<List(_,l2)>*l2::ptr<List(_,l2,null)>@M

  :- R=(l'=null) & F=l::ptr<List(_,l,null)>@M

the derived pre/post goo is :
l::ptr<List(_,l,null)>@M *--> (l'=null)

```

5 Conclusion

References

1. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. Technical Report RR-07-02, Queen Mary Tech Report.
2. Florin Craciun, Hong Yaw Goh, and Wei-Ngan Chin. A framework for object-oriented program analyses via core-java. In *IEEE 2nd International Conference on Intelligent Computer Communication and Processing*, 2006.
3. H.-H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, 2007.