Linear types :
A « simple » way to derive safety properties of programs

Software Analysis and Verification
30 avril 2009

Intuition : Using types to represent properties

```
function addition(x, y)
{
    assume(« x is integer »);
    assume(« y is integer »);
    z = x + y;
    assert(« z is integer »);
}
```

$$\Big\downarrow$$

```
function addition(int x, int y)
{
    int z = x + y;
}
```

Intuition : Using types to represent properties

```
@postcondition(return_value == x)
function invariant(Objet x)
{
    // Computations
    ...

    return x;
}
```

$\downarrow$

```
function invariant(const Objet x)
{
    // Computations
    ...

    return x;
}
```

Remainders about typing

Typing judgement :

$$\Gamma \vdash t : T$$

« under the assumptions Γ, t has type T »

Typing rules :

$$\frac{J1 \qquad\qquad J2}{J}$$

« if the typing judgement J1 and J2 hold, then J also hold »

Remainders about typing

<u>Example of rules :</u>

$$\frac{}{x : T, \Gamma \vdash x : T} \qquad \frac{\Gamma \vdash t1 : int \quad \Gamma \vdash t2 : int}{\Gamma \vdash t2 + t2 : int}$$

<u>Example of typing derivation :</u>

$$\frac{\dfrac{}{x : int, y : int \vdash x : int} \quad \dfrac{}{x : int, y : int \vdash y : int}}{x : int, y : int \vdash x + y : int}$$

Why linear types?

Let us go back to the const example ...

```
// postcondition(o.data == return_value.data);
const Objet* invariant(const Objet* o, Objet* x)
{
    function1(o);
    function2(o, x);

    return o;
}
```

The C compiler accepts this program. Can we assert that the postcondition holds?

Answer 1 : No, we are in C, so much evil things can be done (including unsafe casts) that there is no way that we can assert anything.

Well... Then, assuming that no unsafe C operation is performed in the program, can we assert that the postcondition holds?

Answer 2 : We still can't. For example, invariant could be called like this :

```
Objet o;
invariant(&o, &o);
```

Why linear types?

Modelling state changes [1] (in functional languages):

```
let workon = function x0 ->
  let x1 = concat x0 'h' in
  let x2 = concat x1 'e' in
  let x3 = concat x2 'l' in
  let x4 = concat x3 'l' in
  let x5 = concat x4 'o' in
  x5
```

« x0, x1, x2 … x5 » are successive instances of the same variable that sees his state updated :

We can say that the value of x0 is « neither lost nor duplicated »

References :

[1] Linear Types may change the world, by P. Wadler

Why linear types?

But what if things get most complex?

Let's consider a function where we have a variable x of type 'File'. We want the following properties to hold :

- At the end of the function, we still have access to a variable referring to this file *and only to one*. (non-duplication, non-destruction)

- If during the function, we perform an operation on a file, we want the file that is available at the end of the function to reflect the operation that we performed (no temporary duplication, operations are always performed on the live object).

Linear types : simple version

Idea : Each variable must be used once and exactly once.

```
type Tree = Leaf of int | Node of Tree * Tree

let join = function (x:Tree, y:Tree) -> Node (x, y)
```

Valid

Linear types : simple version

Typing rules for basic lambda calculus :

$$\frac{}{A, x : T \vdash x : T} \longrightarrow \frac{}{x : T \vdash x : T}$$

$$\frac{A \vdash t1 : T \to U \quad A \vdash t2 : T}{A \vdash t1\ t2 : U} \longrightarrow \frac{A \vdash t1 : T \to U \quad B \vdash t2 : T}{A, B \vdash t1\ t2 : U}$$

$$\frac{A, x : T \vdash t : U}{A \vdash (\lambda x : U.\ t) : T \to U} \longrightarrow \text{unchanged}$$

Linear types : simple version

But in case of branching, a single variable may appear more than once ...

```
type Bool = True | False
type Tree = Leaf of int | Node of int * Tree * Tree

let join = function (x:Tree, y:Tree) -> case x of
  Leaf i → Node (Leaf i, y)
| Node (i, u, v) → Node(i, join(u, y), v)
```

Valid

Linear types : simple version

Typing rules for composed types :

for T = T1 * T2 :

$$\frac{A \vdash t1 : T1 \qquad A \vdash t2 : T2}{A \vdash (t1,\ t2) : T} \longrightarrow \frac{A \vdash t1 : T1 \qquad B \vdash t2 : T2}{A,B \vdash (t1,\ t2) : T}$$

for T = C1 T1 | C2 T2 :

$$\frac{\begin{array}{c} A \vdash t : T \\ A,\ x : T1 \vdash t1 : U \\ A,\ x : T2 \vdash t2 : U \end{array}}{\begin{array}{l} A \vdash \text{case } t \text{ of } C1\ x \rightarrow t1 \\ \qquad\qquad\quad C2\ x \rightarrow t2 : U \end{array}} \longrightarrow \frac{\begin{array}{c} A \vdash t : T \\ B,\ x : T1 \vdash t1 : U \\ B,\ x : T2 \vdash t2 : U \end{array}}{\begin{array}{l} A,B \vdash \text{case } t \text{ of } C1\ x \rightarrow t1 \\ \qquad\qquad\qquad C2\ x \rightarrow t2 : U \end{array}}$$

Guarantees of linear types

So, what guarantees does linear types provide to the programmer?

1. Non-aliasing : If a variable x is of a linear type T, then x is the only reference to this object.

2. Memory management : if an allocated object x is given to a function f that returns an object y, then x is not leaked : it must be accessible through y.

Guarantees of linear types

But we can do even better!

Remember the problem we had with the const example?

Now imagine ...

```
// postcondition(o.data == return_value.data);
const Objet* invariant(const non-aliased Objet* o, Objet* x)
{
    function1(o);
    function2(o, x);

    return o;
}
```

Now the postcondition would hold!

Guarantees of linear types

In the same way, we can embed additional state information in types.

[2] presents an interesting extension of programming language that is especially useful when using linear types : a function may change its parameter types.

Imagine that we have two types : `opened file` and `closed file`.

Now let's consider these three functions:

```
- open : (f : closed file → opened file)
- read : (f : opened file → opened file)
- close : (f : opened file → closed file)
```

References :

[2] Typestate : A programming language concept for enhancing software reliability by R.E. Storm and S. Yemini

Guarantees of linear types

```
open : (f : file [closed → opened])
read : (f : file [opened])
close : (f : file [opened → closed])
```

```
function amivalid(closed file f1, closed file f2)
{
   open(f1);     // f1: opened, f2: closed
   read(f1);     // f1: opened, f2: closed
   open(f2);     // f1: opened, f2: opened
   close(f1);    // f1: closed, f2: opened
   read(f2);     // f1: closed, f2: opened
   close(f2);    // f1: closed, f2: closed
}
```

For this extension to have a meaning, we must guarantee non-aliasing of f1 and f2
=> linear types.

The End?

So is that all?

Unfortunately, no...

Because it is impossible to use linear types!

Problems of linear types

- Conceived for functionnal programs with no mutable references

- Cannot be used together with non-linear types (!)

- No support of multiple read access.

Problems of linear types

Problem : Conceived for functionnal programs with no mutable variable

Fortunately, we can interpret an imperative program as a succession of environnement changes :

```
function(int x, int y)
{
    y = y+1;
    x = y;
    return x;
}
```

can be interpreted as :

```
function(int x0, int y0)
{
    let (x1, y1) = (x0, y0+1) in
    let (x2, y2) = (y1, y1) in
    x2;
}
```

Problems of linear types

Problem : Conceived for functionnal programs with no mutable variable

Fortunately, we can interpret an imperative program as a succession of environnement changes :

And if we have :

```
class Obj {Obj1 l1, Obj2 l2}
```

```
function(Obj x, Obj2 y)
{
    x.l2 = y;
    return x;
}
```

can be interpreted as :

```
function(Obj x0, Obj2 y0)
{
    case x0 of Obj(x0l1, x0l2) →
    let (x1l1, x1l2, y1) =
        (x0l1, y0, y0) in
    Obj(x1l1, x1l2)
}
```

Problems of linear types

Problem: Cannot be used together with non-linear types (!)

This is the main problem that is discussed in [3].

To quickly see the problem, let's consider that we use both linear types and non-linear types, that both have their own distinct rules. Then, let's look at the following program :

```
linear class File { … }
non-linear class NLFile { File f }

function evil(File a)
{
    NLFile nl1 = new NLFile(a);
    NLFile nl2 = nl1;     // legit, NLFile non-linear
    woops(nl1.f, nl2.f); // illegal duplication of a
}
```

References :

[3] Adoption and Focus : Practical linear types for imperative programming by M. Fähdrich and R. DeLine

Problems of linear types

Problem: Cannot be used together with non-linear types (!)

It is though possible to use non-linear types with linear types. [1] proposed additionnal typing rules to allow such a mix, but it enforced two rules :

- linear types may contain both linear and non-linear subtypes

- non-linear types may contain only non-linear types

This restriction is not acceptable in practice !

References :

[1] Linear Types may change the world, by P. Wadler

Adoption and Focus : General Idea

We now present the system proposed in [3], which aims is to allow linear references in non-linear containers.

The main idea is to avoid making separating rules for linear types and non-linear types, but instead :

- Treat at the origin all types as non-linear.

- Track the possible aliasing status of variables.

- When required, turn a non-aliased variable into a linear type.

References :

[3] Adoption and Focus : Practical linear types for imperative programming by M. Fähdrich and R. DeLine

Adoption and Focus : General Idea

To explain how this abstraction of a linear type work, we will look at the following code :

```
class Dictionnary
{
    Map<Name, Cell> index_by_name;
    Map<BirthDate, Cell> index_by_birthdate;
}

class Cell
{
    linear Array<int> content;
}

function allocate() : Cell
{
    let cell = new Cell;
    cell.content = new Array(10);
    return cell;
}
```

The idea : The lifetime of cells is binded to the dictionnary. If we look at only one dictionnary cell at a time, we should be fine.

Adoption and Focus : General Idea

The idea : The lifetime of cells is binded to the dictionnary. If we look at only one dictionnary cell at a time, we should be fine.

```
function safe(linear Dictionnary dct, linear Dictionnary dct2,
              Name n)
{
   Cell cell = dct.lookup(n);
   Cell cell2 = dct2.lookup(n);
   // Here cell and cell2 refer to different objects.
   ...
}
```

This is what we will do : when we want to associate a linear type to a non-linear one, we have to provide a « scope », which must be of a linear type. Then, access to the « adopted » linear type will require a « key » associated to the scope, and only one linear objet to the scope can be accessed at the same time.

Restricting scope and access

To perform this access control, we need a mechanism that tells us :

- what are the « scopes » of the variable

- if a variable of the same scope as already been opened or not.

To do this, we will use capabilities as in [4]. The idea is to change the typing statements to :

$$\boxed{\Gamma \, ; C \vdash t : T \, ; C'}$$

where C are the input capabilities and C' the output capabilities

<u>References :</u>

[4] Typed memory management in a calculus of capabilities, by K. Crary, D. Walker and G. Morisett

Restricting scope and access

First, we define new types :

1. `x : tr(a)` means « x is unique and locked. You need the capability a to access it, and if you don't have the capability, i will not even tell you the real type of x»

2. `x : a ▷ t` means « x is protected by a. You can access non-linear components of t freely, but, you will need to use a capability relating to a to access linear components ».

Capabilities are of the form : « a → h » where a is an identifier and h is a heap type (an object stored on the heap).

This capability means : « you can access variables of type tr(a) and this variable is of type h ».

Restricting scope and access

So how does these capabilities emulate linearity?

let's assume we have a variable x of type tr(a) and that the capability {a → h} is present. Then, x behaves « almost » as a linear type :

- x can be aliased, but by doing so you keep the type tr(a) so such aliases can be determined at compile-time : if two variables have the same type tr(a), then they refer to the same object.

- x can be destroyed if the capability disappears {a → h}. Although, if the post condition of the method requires the capability to present, it means x cannot be consumed.

```
function linear(x : tr(a))
  pre { a → h }
  post { a → h }
{
    ...
}
```

Adoption

New syntaxic construct :

```
let x = adopt e1 by e2
```

e1 and e2 must both have a linear type. This expression binds e1 to e2 in the sense that e1 is now restrained to the scope of e2.

Let's look at the typing rule :

$$\frac{\begin{array}{c}\Gamma \ ; C \vdash \text{e1} : \text{tr(a1)} \ ; \text{C1} \\ \Gamma \ ; \text{C1} \vdash \text{e2} : \text{tr(a2)} \ ; \text{C2}, \{\text{a1} \rightarrow \text{h}\} \\ \text{a2 is available in C2}\end{array}}{\Gamma \ ; C \vdash \text{adopt e1:h by e2} : \text{a2} \triangleright \text{h} \ ; \text{C2}}$$

Adoption

We will now go back to our example :

```
class Dictionnary
{
    Map<Name, Cell> index_by_name;
    Map<BirthDate, Cell> index_by_birthdate;
}

class Cell
{
    linear Array<int> content;
}

function allocate(dct : tr(ad)) : ad▷Cell
    pre { ad → Dict }
    post { ad → Dict }
{
    let cell = new Cell;
    let arr = new Array(10);
    cell.content = arr;
    let cell2 = adopt cell by dct;
    return cell2;
}
```

Adoption

Let's look at what really happens :

```
function allocate(dct : tr(ad)) : ad▷Cell
    pre { ad → Dict }
    post { ad → Dict }
{
    let cell /* : tr(a1) */ = new Cell;
    // capabilities : {ad → Dict, a1 → Cell}
    let arr /* : tr(a2) */ = new Array(10);
    // capabilities : {ad → Dict, a1 → Cell, a2 → Array<int>}
    cell.content = arr;
    // capabilities : {ad → Dict, a1 → Cell}
    let cell2 /* : ad ▷ Cell */ = adopt cell by dct;
    // capabilities : {ad → Dict}
    return cell;
}
```

Focus

Now that we can store linear types in non-linear ones, let's look at how we access such linear types that are « unsafely » stored.

New syntaxic construct :

```
let x = focus e1 in e2
```

the result of e1 should be a non-linear object with usually linear components. The focus allows x to become a linear alias of this result during e2. The idea is that we will ensure that x is really linear in e2 by :

- forbidding access to every object that may be an alias of x or of one of its component

- requiring that x is still alive at the end of e2.

$$\frac{\begin{array}{c}\Gamma \,;\, C \vdash e1 : a1 \triangleright h \,;\, \{a1 \to h?\}, C1 \\ a2 \text{ is fresh} \\ \Gamma, x{:}\ tr(a2) \,;\, C1, \{a2 \to h\} \vdash e2 : T \,;\, C2, \{a2 \to h\}\end{array}}{\Gamma \,;\, C \vdash \text{let } x = \text{focus } e1 \text{ in } e2 : T \,;\, \{a1 \to h?\}, C2}$$

Focus

```
class Dictionnary
{
    Map<Name, Cell> index_by_name;
    Map<BirthDate, Cell> index_by_birthdate;
}

class Cell
{
    linear Array<int> content;
}

function resize(cell : ad▷Cell, size : int) : Unit
    pre { ad → Dict }
    post { ad → Dict }
{
    let newa = new Array(size);
    let fcell = focus cell in
    {
        let olda = fcell.content;
        copy(olda, newa);
        fcell.content = newa;
        free(olda);
    }
}
```

Back to linear types in imperative programming

```
function resize(cell : ad▷Cell, size : int) : Unit
    pre { ad → Dict }
    post { ad → Dict }
{
    let newa = new Array(size);
    let fcell = focus cell in
    {
        let olda = fcell.content;
        copy(olda, newa);
        fcell.content = newa;
        free(olda);
    }
}
```

Side question : Are 'copy' and 'free' legitimate, knowing that `olda` and `newa` are of linear types inside the focus scope?

Copy : Yes. We copy the internal of the array only, not the array itself, and the internal of the array are of non-linear type.

Free : No! Destroying the array breaks the survivability of linear objets that we required.

… but we will make an exception !

Focus

```
function resize(cell : ad▷Cell, size : int) : Unit
    pre { ad → Dict }
    post { ad → Dict }
{
    let newa /* tr(a1) */ = new Array(size);
    // capabilities : { ad → Dict, a1 → Array<int> }
    let fcell /* tr(a2) */ = focus cell in {
        // capabilities : { a1 → Array<int>, a2 → Cell }
        let olda /* tr(a3) */ = fcell.content;
        // capabilities : { a1 → Array<int>, a3 → Array<int>,
                            a2 → {content = tr(a3)} }
        copy(olda, newa);
        // capabilities : { a1 → Array<int>, a3 → Array<int>,
                            a2 → {content = tr(a3)} }
        fcell.content = newa;
        // capabilities : { a1 → Array<int>, a3 → Array<int>,
                            a2 → {content = tr(a1)} }
        // rebuilding cell...
        // capabilities : { a2 → Cell, a3 → Array<int> }
        free(olda);
        // capabilities : { a2 → Cell }
    }
    // capabilities : { ad → Dict }
}
```

Adoption and Focus : Summing up

Methodology :

- Linear types can be stored in any classes

- Possible alias analysis is performed in the whole program

- We can only access linear components of a class if there is no possible aliasing of them in the scope.

- We can emulate the « disparition » of a variable by removing the capability

What do we gain :

- We have all guarantees of linearity for the linear object fields, including the fact that these fields are all eventually free'd.

- We can share such linear objects among the program without violating any guarantee.

That's all.

References :

[1] Linear Types may change the world, by P. Wadler

[2] Typestate : A programming language concept for enhancing software reliability by R.E. Storm and S. Yemini

[3] Adoption and Focus : Practical linear types for imperative programming by M. Fähdrich and R. DeLine

[4] Typed memory management in a calculus of capabilities, by K. Crary, D. Walker and G. Morisett

Questions?