Clay: A Type-Safe Systems Programming Language
Bucknell Computer Science Technical Report #08-1

Lea Wittie

March 12, 2008

1 Introduction

The Clay programming language is a type-safe variant of C/C^{++}. This language was developed at Dartmouth College primarily by Chris Hawblitzel, Gary Morris, Eric Krupski, and Ed Wei with minor updates by Lea Wittie. The language currently supports singleton and linear types, arithmetic constraints in function pre and post conditions, polymorphism, and type inference. Clay is able to statically detect many errors including buffer overflows, kernel stack overflows, NULL pointer uses, freed memory uses, and aliasing errors.

Currently many necessary safety checks in operating systems are done at run time or not at all. Clay moves the burden of error checking from run time to compile time thereby allowing the programmer to find errors while compiling and lessening the number of errors the end user sees.

The Clay compiler consists of a lexer, parser, syntax checker, type checker, the Omega constraint checker [3], and a compiler to C^{++}. The type checker produces verification conditions, which are discharged in the Omega constraint checker.

A formal version of Clay with proofs of progress, preservation, and erasure can be found in [2].

2 Type Systems and Safe Languages

Not all type systems are equally powerful; a weak static type system forces the language to perform run-time safety checks. For example, type-safe languages like Java and ML perform automatic run-time checks on array bounds. Unsafe languages (C, C++) expect the programmer to include run-time checks. In a large program such as an OS, there are likely to be many run-time bounds checks that are left out or slow the program down considerably. A study by Chou et al. [1] found many forgotten bounds checks in Linux. The inserted run-time bound checks are one reason that operating systems are not usually written in Java. If these checks could be moved to compile-time, the program would have the same safety guarantees without continual run-time bounds checks.

In addition, none of the languages mentioned above are able to catch aliasing errors automatically. An aliasing error occurs when several copies, or aliases, of a system-state value are in existence and the system-state changes and then an outdated copy is mistakenly used. Some aliasing errors could be caught by keeping and checking reference counts but this would contribute to the already expensive run-time checks. Compile time checks for aliasing errors would provide safety without adding to the run time burden.

Clay uses advanced concepts such as type-level arithmetic constraints, singleton type, and linear types offer a compile-time solution to the safety and speed problems mentioned above.

2.1 Arithmetic Constraints

Usually, run time checks such as those on array bounds are required at run-time because the necessary information (the exact index into the array and the exact size of the array) aren't known until run-time. If this information was captured in the static type system, checks could be performed at compile-time rather than at run-time. Doing so requires a system for statically reasoning about run-time values.

Xi and Pfenning [5] added arguments to integer types and added arithmetic and boolean operators $(==, !=, <, >, +, -)$ to their type system in order to make comparisons between type arguments. These additions let them write invariants such as $0 \leq index < size$ into their function types.

2.2 Singleton Types

Xi and Pfenning used singleton types to connect type variables to run-time variables. A singleton type is a type with one element. The singleton type $Int[3]$ is the type of integers whose value is 3. Together, the singleton types and arithmetic comparisons allowed them to encode invariants about run-time values in their type system. They demonstrated how to eliminate run-time checks such as array bounds using these invariants.

2.3 Linear Types

There aren't many easy ways to use run-time checks to avoid aliasing errors. Reference counting is sometimes possible but can have a high cost in running time. Types based on the linear logic of Girard prevent aliasing errors without run-time checks [4].

A linear value has only one reference to it. A linear variable is able to be accessed exactly once and cannot be duplicated or discarded. Any attempt to copy a linear variable creates a duplicate but invalidates the original. Therefore, linear variables are explicitly handed from one function to the next. This prevents aliasing errors.

3 The Type-Safe Language Clay

Clay uses singleton types but expands them to create types that are parameterized by several type variables. Using similar arithmetic comparisons, Clay moves many run-time checks to compile time and enforces pre and post conditions on functions. The arithmetic comparisons generate first-order constraints at compile time. These constraints are solved using Omega [3], an automated constraint checker.

Singleton and linear types allow compile-time detection of aliasing errors and invariant failures. Where missing run-time checks are necessary, invariants will fail at compile-time alerting the programmer to the omission.

After semantic checking and type checking is complete, a Clay program is compiled into native code. In this version of the Clay compiler, the native language is C++.

3.1 Naming Conventions

Function and variable names must begin with a lowercase letter. Types and type parameters (other than type *int*) must begin with an uppercase letter.

3.2 Type System

3.2.1 Kinds

Types in Clay are classified into sets called kinds.

The integer types included from C are classified under kind $type32$ meaning they represent non-linear, 32 bits data. Other sizes of data are possible using kind $type8$, $type16$, etc. Kind *type* can be used as shorthand for the common kind $type32$. The bool type (from C^{++}) currently also has kind *type*. Although C does not come with a 0 bit type, Clay uses $type0$ for capabilities.

Kind *type* can be declared as linear or non-linear. The default is non-linear as seen in the above examples. Linearity is denoted by a @ so kind $@type32$ is the kind of linear 32-bit types. The capabilities in Clay are usually linear and thus have kind $@type0$.

Many types in Clay are parameterized by type variables. These variable are classified under kinds *int* and *bool*.

Clay also allows type functions. A type parameter that is a function from *int* to $@type0$ would have kind $@type0 < -(int)$.

3.2.2 Types

Clay includes two basic type constructors and several new types including existentials and tuples. The first constructor creates a new type from scratch. The syntax is

```
kind type_name[declaration of type variables] = native
```

For example, the types of the singleton integers and singleton booleans are each declared with their respective kinds.

```
type8 Int8[int N] = native
type16 Int16[int N] = native
type Int32[int N] = native
type Bool[bool B] = native
```

Numeric literals automatically convert to singletons so the numbers 15 and 0xf both have type $Int32[15]$. The basic types are built into Clay in the sense that the numeric and boolean literals have type singleton integer and singleton boolean. The actual Clay type declarations are included in a header file (*.clh). Because these types are native, they are translated to a type in native code when a Clay program compiles. However, an exception to this statement is the 0-bit types. A memory capability type

```
@type0 Mem8[int I, int A] = native
@type0 Mem16[int I, int A] = native
@type0 Mem32[int I, int A] = native
```

states that memory location I contains a value of type $Int[A]$. Because this type is 0-bit, it has no native equivalent and exists solely at compile time.

Structs are used similarly to those in C. A struct creates a new type as a cartesian product of existing types. The struct is named and can be recursive. The kind of a struct must be specified in the type declaration.

```
@type0 Mems[int I] = struct {
                        Mem32[I,1]   a;
                        Mem32[I+1,2] b;
                        Mem32[I+2,3] c;
                     }
```

The type created above is the capability for three contiguous memory locations whose values are permanently fixed at 1, 2, and 3. A $Mems$ is linear and 0-bit.

The second constructor creates new types out of existing types. The syntax is

```
typedef type_name[declaration of type variables] = existing_type
```

For example, 32 bit integers are used so frequently that we can define Int to mean $Int32$ and Mem to mead $Mem32$.

```
typedef Int = Int32
typedef Mem = Mem32
```

The existing type can be also parameterized by the declared type variables.

```
typedef TwiceInt[int N] = Int[N+N]
```

Therefore the number 4 has type $Int[4]$ or $TwiceInt[2]$ (also equivalent to $Int32[4]$). Clay will infer the kind of the newly constructed type.

Existentials in Clay use a fairly standard syntax:

```
typedef SomeInt = exists [int I] Int[I]
```

Here, the type *SomeInt* is defined as any singleton integer. Existentials can be combined with struct to create a capability for three contiguous memory locations containing any values.

```
@type0 Mems[int I] = struct {
                        exists [int V] Mem[I,  V] a;
                        exists [int V] Mem[I+1,V] b;
                        exists [int V] Mem[I+2,V] c;
                     }
```

Clay also includes a tuple type which makes an ordered pair of other types. These tuples are standard in that a tuple can have any number of components and the components do not have to be of the same types. However, a tuple that contains any linear components must be itself linear. In comparison to structs, a tuple is anonymous, non-recursive and its kind is inferred.

For example, a pointer type to pair a memory location and its memory capability.

```
typedef Ptr[int I, int T] = @[Int[I], Mem[I,T]]
```

Here, a *Ptr* is a linear tuple of a singleton integer and a *Mem*. The integer points to a memory location and the *Mem* is the capability for that location.

The type of a Clay function shows any relationships between parameters and the return type.

```
Int[N+M] add[int N, int M](Int[N] n, Int[M] m) { return m + n; }
```

This add function takes two singleton integers and returns another singleton integer. The type parameter inputs are declared in square braces after the function name. The return type shows the relation betwenn input and output. Note: the + operator is built into Clay with a function type that is identical to the add function seen here.

3.2.3 Arithmetic Constraints on Types

The type parameter declarations of functions and existentials can both be annotated with arithmetic comparisons and boolean logic.

```
typedef SomePosInt = exists [int V; V>0] Int[V]
```

This *SomePosInt* type is an existential singleton integer whose value is greater than 0.

We can also adapt the previous add function to require positive inputs

```
Int[N+M] addpos [int N, int M; N>0 && M>0] (Int[N] n, Int[M] m) { return m + n; }
```

Functions on signed or unsigned 32-bit integers may use arithmetic comparisons on the inputs inside the type variable declarations.

```
Int[N+M] unsigned_add
[int N, int M; 0<=N && N<4294967296 && 0<=M && M<4294967296]
(Int[N] n, Int[M] m) { return n + m; }
```

Arithmetic comparisons can also be used on function outputs by returning an existential type. For example, here is a function to take and return existential integers which are greater than 5.

```
exists[int N; N>5] Int[N]
existential_unsigned_identity
(exists[int N; N>5] Int[N] n) { return n; }
```

This allows us to specify arithmetic pre- and postconditions on a function.

Outer bounds on type variables such are $0 <= N$ && $N < 4294967296$ bounds are common enough that Clay includes syntactic sugar to simplify constraints.

```
Int[N+M] unsigned_add
[u32 N, u32 M]
(Int[N] n, Int[M] m) { return n + m; }
```

The type declaration $u32\ N$ is an abbreviation for an $int\ N; 0 <= N$ && $N < 4294967296$. Signed and unsigned abbreviations of every size can be used ($s32$, $u8$, etc.).

3.3 Built-in operators

Clay includes the && || and ! boolean operators on constraints and the relational operators (==, ! =, <, <=, >, >=) as well as addition, subtraction, and multiplication on type variables. However, the one of the operands in a multiplication must be a constant so that the results of the operation can be tracked in the type variables.

Each built-in operator maps to a set of overloaded functions and Clay determines which is appropriate for a given usage. For example, the addition operator (+) maps to the following three functions:

```
typedef __S32 = exists[s32 N] Int[N]
typedef __U32 = exists[u32 N] Int[N]

native Int[N+M] i32_add[int N, int M](Int[N] n, Int[M] m);
native __S32 s32_add(__S32 n, __S32 m);
native __U32 u32_add(__U32 n, __U32 m);
```

The type declarations are for signed and unsigned existential integers. The possibilities for addition are general integers (where the sign doesn't matter), existential signed integers, and existential unsigned integers. The expression (3 + 4) would therefore use addition for general integers, i32_add. The functions are shown as headers with a native declaration because the actual implementation is in native code.

3.4 Termination Proofs

Functions and function calls that have no run-time effect can be erased after type checking. Functions that wish to be considered for erasure must provide a proof that they terminate. Clay allows each such function to state a limit on the amount of work the function will do. If all function calls within this function do less work then the function will eventually terminate. Normal functions do an unlimited amount of work and thus cannot be erased.

This section shows several examples of limited functions. Each example builds on previous examples.

3.4.1 Unlimited functions

As seen above, the Clay definition of singleton integer addition is

```
native Int[N+M] i32_add[int N, int M](Int[N] n, Int[M] m);
```

Native means that the body of the function is defined in native code (C^{++}). i32_add does work at runtime so it cannot be erased. This function could also be written as

```
native Int[N+M] i32_add[int N, int M](Int[N] n, Int[M] m) limitany;
```

Where it is explicitly declared to do an unlimited amount of work and is therefore not erasable.

The native C^{++}code that matches this function is

```
extern inline unsigned long iu32_add(unsigned long n, unsigned long m) { return n + m; }
```

By default, functions without an explicit limit are unlimited so a Clay programmer may leave off the limitany if they wish.

3.4.2 Limit of 0

This example shows functions with a limit of 0. These do no work at run-time and therefore can be erased after type checking.

The linear 0-bit union type consists of a boolean and two linear 0-bit subtypes.

```
@type0 LUnion0[bool P, @type0 A, @type0 B] = native
```

This type has four operations

```
native LUnion0[true,A,B] lunion0_make_t[@type0 A, @type0 B](A a) limited[0];
native LUnion0[false,A,B] lunion0_make_f[@type0 A, @type0 B](B b) limited[0];
native A lunion0_t[@type0 A, @type0 B](LUnion0[true,A,B] u) limited[0];
native B lunion0_f[@type0 A, @type0 B](LUnion0[false,A,B] u) limited[0];
```

These first two functions create true and false unions. Notice that lunion0_make_t declares B as an input type parameter even though it is not in the inputs. A caller to this function will need to supply B. The second two functions evaluate to one of the two subtypes depending on the boolean.

All four functions are declared native so their actual code would be in the native (C^{++}) language. However, since they have a limit of 0, they do no work at run-time and can be erased along with all calls to them.

3.4.3 Constant limit

This example shows functions with a constant limit. These can also be erased after type checking.

```
typedef LIf[bool P, @type0 A] = LUnion0[P, A, @[]]
```

The LIf linear-if type consists of a boolean, a data type, and the empty tuple type. This type is used to represent possibly null pointers. If A is a Mem capability, then $LIf[true, A]$ results in the Mem and $LIf[false, A]$ is a null pointer and results in @[] which cannot be used to access the memory location guarded by the Mem capability.

```
#define LimitStd 0                  // The smallest limit
#define LimitUnion (LimitStd + 10) // some Union functions call Standard functions

LIf[true,A] lif_make_t[@type0 A](A a) limited[LimitUnion]
{
  return lunion0_make_t[B=@[]](a);
}
LIf[false,A] lif_make_f[@type0 A]() limited[LimitUnion]
{
  return lunion0_make_f[A=A](@());
}
```

These two functions create true and false LIf data. Both have limit LimitStd + 10 so they will have a higher limit than any of the functions they might call. The 10 is not a special number and simply needs to make LimitUnion larger than LimitStd. In this example, a 1 would also have sufficed.

Notice that the function calls to lunion0_make_t and lunion0_make_f pass in values for the input type parameters. In lif_make_t, B is convieniently set to the empty tuple type. In lif_make_f, A is set to the input type parameter A and the variable parameter @() has type @[]. (In $A = A$, the first A is in scope in the called function and the second A is in scope in the calling function.)

3.4.4 Limit dependant on input

This example shows functions whose limit depends on their inputs. Because these functions are limited and all sub-functions have lower limits, they can be erased after type checking.

The linear 0-bit array type consists of beginning and ending indices and a function which maps indices to linear 0-bit types.

```
@type0 LArrayS[int I, int J, @type0<-(int) F] = struct
{
  F[I] data;
  LArray[I+1,J,F] next;
}
typedef LArray[int I, int J, @type0<-(int) F] = LIf[I!=J,LArrayS[I,J,F]]
```

This type is implemented as a linked list. If index I isn't equal to index J then the field is either a node (type *LArray*). Otherwise, the field has type @[] and is null.

```
#define LimitLArray (LimitUnion + 10) // LArray functions call Union functions

LArray[I,K,F]
  larray_combine[@type0<-(int) F, int I, int J, int K; I<=J && J<=K](
    LArray[I,J,F] array1,
    LArray[J,K,F] array2) limited[LimitLArray+(J-I)]
{
  // if array1 is empty, return array2
  if[I==J]
  {
    discard_larray(array1); // LArray function [LimitLFacts]

    return array2;
  }
  // else recursively build the combined array
  else
  {
    let (data1, next1) = lif_t(array1); // Union function [LimitUnion]
    let next = larray_combine(next1, array2); // LArray function [LimitLArray+J-I-1]
    return lif_make_t(struct LArrayS(data1, next)); // Union function [LimitUnion]
  }
}
```

This function combines two adjacent arrays into a single array. If the first array is empty, it returns the second array. Otherwise, it recursively builds the combined array. The limit is based on the number of elements in the first array because this function will make that number of recursive calls. The sub-functions it calls all have limits less than the limit on this function.

This array type could hold the memory capacilities for the three contiguous memory locations that we have used as examples so far. Given a pointer to the base of the array (type $Int[Base]$), the type

```
LArray[0,3,fun[int I] exists[int A] Mem[Base+I,A]]
```

describes an array of three contiguous *Mem* types. Unlike the structs in the previous contiguous memory examples, more *Mem* values can be added to this array.

3.5 Statements

3.5.1 Variable declarations

Variables must be initialized when they are declared. Unlike C, Clay has some type inference and can frequently guess the correct variable type from the initial value. The type can also be stated explicitly.

Let expressions infer the type of a declared variable

```
let x = 3;
```

This expression infers that x has type $Int[3]$, a 32 bit singleton integer.

A let expression may also separate a tuple into it's components.

```
let (x,y,z) = .(1,2,3);
```

A tuple must have explicitly declared linearity. In this example, the tuple is nonlinear (as seen by the period before the tuple). A linear tuple would be declared with an @ sign.

Clay has builtin explicit types for existentials: int, byte, bool, and void.

Abbreviation	Internal Representation	Type Definition
int	__Int	typedef __Int = exists[s32 N] Int[N]
byte	__Byte	typedef __Byte = exists[u8 N] Int[N]
bool	__Bool	typedef __Bool = exists[bool B] Bool[B]
void	__Unit	type0 __Unit = native

This table shows the mapping of type abbreviations to Clay's internal representation. For instance, the variable declaration

```
int x = 3;
```

assigns x the type __Int. The actual definition of __Int is done by the Clay programmer. We suggest some type definitions for these types in the third column of the chart but a Clay programmer is free to define __Int as a general integer or an unsigned integer. Note that the type int has a different meaning than the kind int seen in type variable declarations.

A let expression can infer a singleton or existential type as needed. In order to remove the existential from a type, a let expression uses explicit [ ] to unpack the type.

```
int a = 3;
int b = 4;
let [] c = a + b;
```

This example uses the s32_add, existential addition, and then unpacks it to get the singleton integer. Although we know c has the value 7, the existentials do not retain that information so c has type exists[s32 N] Int[N].

With careful packing however, we can retain values in an existential.

```
let x = 3;
let y = pack [exists [s32 I; I==3] Int[I]] (x);
let [] z = y;
```

The variable x has type $Int[3]$. The pack operation means y has type $exists[s32 I; I == 3]Int[I]$ and the type checker will infer that z also has type $Int[3]$.

3.5.2 Selection

Selection can be done on variables or type parameters. However, selection on type parameters can only be done in a limited function. Any linear variables assigned inside a branch of the selection will not be available after the selection so most selection statements contain return statements. This example is the linear 0-bit array function to concatenate two arrays.

```
LArray[I,K,F]
  larray_combine[@type0<-(int) F, int I, int J, int K; I<=J && J<=K]
  (LArray[I,J,F] array1, LArray[J,K,F] array2) limited[LimitLArray+(J-I)]
{
  if[I==J]
  {
    discard_larray(array1);
    return array2;
  }
  else
  {
    let (data1, next1) = lif_t(array1);
    let next = larray_combine(next1, array2);
    return lif_make_t(struct LArrayS(data1, next));
  }
}
```

3.5.3 Repetition

Repetition is similar to the repetition of C^{++}except that it requires a continue statement to start each loop after the first. This function will print all numbers between i and 10. The continue statement fills the same function as the update in a C^{++}for-loop.

```
void count[s32 V](Int[V] i)
{
    for [s32 I] (Int[I] i = i; i < 11)
        {
            print_int(i);
            continue(i+1);
        }
}
```

When called on $count(0)$, this will print 0 through 10. A Clay for loop can take multiple inputs separated by commas and included as parameters to the continue call. Linear variables will still be available after such a loop if they were inputs to the loop.

3.6 Native Code

Clay does not directly support effects so all actions with side effects must drop to native code to complete them.

For example, loads and store to memory using the memory capabilities (Mem) are an effect so a load function defined as

```
native @[Mem[I,A],A] load[int I, u32 A](Int[I] ptr, Mem[I,A] cap) limitany;
```

requires a matching C^{++}function. In this case the C^{++}function is

```
inline unsigned long load(unsigned long addr) {
    return *((unsigned long *) addr);
}
```

Native C^{++}code is not type-safe and requires the programmer to verify that these operations are safe. Therefore, we suggest making as few functions as possible native as both erasible and non-erasible native functions can undermine the type system.

4 Hello World

hello.clay:

```
native void print_int(int i);

void hello() {
  print_int(1);
}
```

hello.c:

```
void hello();

int main() {
  hello();
  return 0;
}
```

To compile it:

```
clay hello.clay
g++ hello.c hello.cc
```

5 Bugs and Future Work

Although we included syntactic sugar where possible, Clay does have some less than desirable features. These may get fixed in the future.

5.1 Native variable types

Currently the Clay to C^{++}translation has two possibilities for variable types: void and unsigned long. This has the unfortunate side effect of making all booleans 32-bits and wasting memory. It also might make programs with signed integers tricky although all of the built-in operators will work correctly. In practice, this has not been an issue because we have used Clay for garbage collectors and device drivers which only use unsigned integers.

5.2 Type variables in some existentials

Clay has an unusual design for quantified types: the string variable name is part of the type, so that you can use the syntax ”e[A=t]” to instantiate type arguments by name. This is somewhat like named members in a class or record, or like labeled parameters in OCaml.

This means that addition on existential unsigned integers

```
exists[int N; 0<=N && N<4294967296] Int[N]
existential_unsigned_add
  (exists[int N; 0<=N && N<4294967296] Int[N] n,
   exists[int N; 0<=N && N<4294967296] Int[N] m)
{
  return n + m;
}
```

is forced to use the type parameter N in both the input and the return type because the definition of the + operator

```
typedef __U32 = exists[u32 N] Int[N]
native __U32 u32_add(__U32 n, __U32 m) limitany;
```

uses the type parameter N in both the inputs and the return type.

5.3 Kind inference

In functions such as

```
Int[N+M] add[int N, int M](Int[N] n, Int[M] m) { return m + n; }
```

Clay could infer kinds int N and int M but it won’t. Early versions of Clay had kind inference so you could write

```
Int[N+M] add[N,M](Int[N] n, Int[M] m) { return m + n; }
```

but the non-local error messages it produced were confusing and Clay couldn’t infer arithmetic constraints. The kind inference was eventually dropped for clarity.

References

[1] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Banff, Alberta, Canada, 2001.

[2] Heng Huang, Lea Wittie, and Chris Hawblitzel. Formal properties of linear memory types. Technical report, Dartmouth College, 2003.

[3] William Pugh. The omega project. http://www.cs.umd.edu/projects/omega/.

[4] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[5] Honwei Xi and Frank Pfenning. Eliminating array bounds checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, Canada, June 1998.