unsized_locals

The tracking issue for this feature is: #48055

This implements RFC1909. When turned on, you can have unsized arguments and locals:

```
#![allow(incomplete_features)]
#![feature(unsized_locals, unsized_fn_params)]

use std::any::Any;

fn main() {
    let x: Box<dyn Any> = Box::new(42);
    let x: dyn Any = *x;
    //  ^ unsized local variable
    //               ^^ unsized temporary
    foo(x);
}

fn foo(_: dyn Any) {}
//     ^^^^^^ unsized argument
```

The RFC still forbids the following unsized expressions:

```
#![feature(unsized_locals)]

use std::any::Any;

struct MyStruct<T: ?Sized> {
    content: T,
}

struct MyTupleStruct<T: ?Sized>(T);

fn answer() -> Box<dyn Any> {
    Box::new(42)
}

fn main() {
    // You CANNOT have unsized statics.
    static X: dyn Any = *answer();  // ERROR
    const Y: dyn Any = *answer();  // ERROR

    // You CANNOT have struct initialized unsized.
    MyStruct { content: *answer() };  // ERROR
    MyTupleStruct(*answer());  // ERROR
    (42, *answer());  // ERROR

    // You CANNOT have unsized return types.
    fn my_function() -> dyn Any { *answer() }  // ERROR

    // You CAN have unsized local variables...
    let mut x: dyn Any = *answer();  // OK
    // ...but you CANNOT reassign to them.
    x = *answer();  // ERROR

    // You CANNOT even initialize them separately.
    let y: dyn Any;  // OK
    y = *answer();  // ERROR

    // Not mentioned in the RFC, but by-move captured variables are also Sized.
    let x: dyn Any = *answer();
    (move || {  // ERROR
        let y = x;
    })();

    // You CAN create a closure with unsized arguments,
    // but you CANNOT call it.
    // This is an implementation detail and may be changed in the future.
    let f = |x: dyn Any| {};
    f(*answer());  // ERROR
}
```

By-value trait objects

With this feature, you can have by-value `self` arguments without `Self: Sized` bounds.

```
#![feature(unsized_fn_params)]

trait Foo {
    fn foo(self) {}
}

impl<T: ?Sized> Foo for T {}

fn main() {
    let slice: Box<[i32]> = Box::new([1, 2, 3]);
    <[i32] as Foo>::foo(*slice);
}
```

And `Foo` will also be object-safe.

```rust
#![feature(unsized_fn_params)]

trait Foo {
    fn foo(self) {}
}

impl<T: ?Sized> Foo for T {}

fn main () {
    let slice: Box<dyn Foo> = Box::new([1, 2, 3]);
    // doesn't compile yet
    <dyn Foo as Foo>::foo(*slice);
}
```

One of the objectives of this feature is to allow `Box<dyn FnOnce>`.

Variable length arrays

The RFC also describes an extension to the array literal syntax: `[e; dyn n]`. In the syntax, `n` isn't necessarily a constant expression. The array is dynamically allocated on the stack and has the type of `[T]`, instead of `[T; n]`.

“‘rust,ignore (not-yet-implemented) #[feature(unsized_locals)]

fn mergesort<T: Ord>(a: &mut [T]) { let mut tmp = [T; dyn a.len()]; // … }

fn main() { let mut a = [3, 1, 5, 6]; mergesort(&mut a); assert_eq!(a, [1, 3, 5, 6]); }

~~~
VLAs are not implemented yet. The syntax isn't final, either. We may need an alternative syn

## Advisory on stack usage

It's advised not to casually use the `#![feature(unsized_locals)]` feature. Typical use-case

- When you need a by-value trait objects.
- When you really need a fast allocation of small temporary arrays.

Another pitfall is repetitive allocation and temporaries. Currently the compiler simply exte

```rust
#![feature(unsized_locals)]

fn main() {
    let x: Box<[i32]> = Box::new([1, 2, 3, 4, 5]);
    let _x = {{{{{{{{{{*x}}}}}}}}}};
}
~~~

and the code

```rust
#![feature(unsized_locals)]

fn main() {
    for _ in 0..10 {
        let x: Box<[i32]> = Box::new([1, 2, 3, 4, 5]);
        let _x = *x;
    }
}
```

will unnecessarily extend the stack frame.