Introduction to the Go compiler's SSA backend

This package contains the compiler's Static Single Assignment form component. If you're not familiar with SSA, its Wikipedia article is a good starting point.

It is recommended that you first read cmd/compile/README.md if you are not familiar with the Go compiler already. That document gives an overview of the compiler, and explains what is SSA's part and purpose in it.

Key concepts

The names described below may be loosely related to their Go counterparts, but note that they are not equivalent. For example, a Go block statement has a variable scope, yet SSA has no notion of variables nor variable scopes.

It may also be surprising that values and blocks are named after their unique sequential IDs. They rarely correspond to named entities in the original code, such as variables or function parameters. The sequential IDs also allow the compiler to avoid maps, and it is always possible to track back the values to Go code using debug and position information.

Values Values are the basic building blocks of SSA. Per SSA's very definition, a value is defined exactly once, but it may be used any number of times. A value mainly consists of a unique identifier, an operator, a type, and some arguments.

An operator or `Op` describes the operation that computes the value. The semantics of each operator can be found in `gen/*Ops.go`. For example, `OpAdd8` takes two value arguments holding 8-bit integers and results in their addition. Here is a possible SSA representation of the addition of two `uint8` values:

```
// var c uint8 = a + b
v4 = Add8 <uint8> v2 v3
```

A value's type will usually be a Go type. For example, the value in the example above has a `uint8` type, and a constant boolean value will have a `bool` type. However, certain types don't come from Go and are special; below we will cover `memory`, the most common of them.

See value.go for more information.

Memory types `memory` represents the global memory state. An `Op` that takes a memory argument depends on that memory state, and an `Op` which has the memory type impacts the state of memory. This ensures that memory operations are kept in the right order. For example:

```
// *a = 3
// *b = *a
v10 = Store <mem> {int} v6 v8 v1
v14 = Store <mem> {int} v7 v8 v10
```

Here, `Store` stores its second argument (of type `int`) into the first argument (of type `*int`). The last argument is the memory state; since the second store depends on the memory value defined by the first store, the two stores cannot be reordered.

See cmd/compile/internal/types/type.go for more information.

Blocks A block represents a basic block in the control flow graph of a function. It is, essentially, a list of values that define the operation of this block. Besides the list of values, blocks mainly consist of a unique identifier, a kind, and a list of successor blocks.

The simplest kind is a `plain` block; it simply hands the control flow to another block, thus its successors list contains one block.

Another common block kind is the `exit` block. These have a final value, called control value, which must return a memory state. This is necessary for functions to return some values, for example - the caller needs some memory state to depend on, to ensure that it receives those return values correctly.

The last important block kind we will mention is the `if` block. It has a single control value that must be a boolean value, and it has exactly two successor blocks. The control flow is handed to the first successor if the bool is true, and to the second otherwise.

Here is a sample if-else control flow represented with basic blocks:

```
// func(b bool) int {
//  if b {
//      return 2
//  }
//  return 3
// }
b1:
  v1 = InitMem <mem>
  v2 = SP <uintptr>
  v5 = Addr <*int> {~r1} v2
  v6 = Arg <bool> {b}
  v8 = Const64 <int> [2]
  v12 = Const64 <int> [3]
  If v6 -> b2 b3
b2: <- b1
  v10 = VarDef <mem> {~r1} v1
  v11 = Store <mem> {int} v5 v8 v10
  Ret v11
b3: <- b1
  v14 = VarDef <mem> {~r1} v1
  v15 = Store <mem> {int} v5 v12 v14
```

```
  Ret v15
```

See block.go for more information.

Functions A function represents a function declaration along with its body. It mainly consists of a name, a type (its signature), a list of blocks that form its body, and the entry block within said list.

When a function is called, the control flow is handed to its entry block. If the function terminates, the control flow will eventually reach an exit block, thus ending the function call.

Note that a function may have zero or multiple exit blocks, just like a Go function can have any number of return points, but it must have exactly one entry point block.

Also note that some SSA functions are autogenerated, such as the hash functions for each type used as a map key.

For example, this is what an empty function can look like in SSA, with a single exit block that returns an uninteresting memory state:

```
foo func()
  b1:
    v1 = InitMem <mem>
    Ret v1
```

See func.go for more information.

Compiler passes

Having a program in SSA form is not very useful on its own. Its advantage lies in how easy it is to write optimizations that modify the program to make it better. The way the Go compiler accomplishes this is via a list of passes.

Each pass transforms a SSA function in some way. For example, a dead code elimination pass will remove blocks and values that it can prove will never be executed, and a nil check elimination pass will remove nil checks which it can prove to be redundant.

Compiler passes work on one function at a time, and by default run sequentially and exactly once.

The `lower` pass is special; it converts the SSA representation from being machine-independent to being machine-dependent. That is, some abstract operators are replaced with their non-generic counterparts, potentially reducing or increasing the final number of values.

See the `passes` list defined in compile.go for more information.

Playing with SSA

A good way to see and get used to the compiler's SSA in action is via `GOSSAFUNC`. For example, to see func `Foo`'s initial SSA form and final generated assembly, one can run:

```
GOSSAFUNC=Foo go build
```

The generated `ssa.html` file will also contain the SSA func at each of the compile passes, making it easy to see what each pass does to a particular program. You can also click on values and blocks to highlight them, to help follow the control flow and values.

The value specified in GOSSAFUNC can also be a package-qualified function name, e.g.

```
GOSSAFUNC=blah.Foo go build
```

This will match any function named "Foo" within a package whose final suffix is "blah" (e.g. something/blah.Foo, anotherthing/extra/blah.Foo).

If non-HTML dumps are needed, append a "+" to the GOSSAFUNC value and dumps will be written to stdout:

```
GOSSAFUNC=Bar+ go build
```

Hacking on SSA

While most compiler passes are implemented directly in Go code, some others are code generated. This is currently done via rewrite rules, which have their own syntax and are maintained in `gen/*.rules`. Simpler optimizations can be written easily and quickly this way, but rewrite rules are not suitable for more complex optimizations.

To read more on rewrite rules, have a look at the top comments in gen/generic. rules and gen/rulegen.go.

Similarly, the code to manage operators is also code generated from `gen/*Ops.go`, as it is easier to maintain a few tables than a lot of code. After changing the rules or operators, see gen/README for instructions on how to generate the Go code again.