An SSA Analyzation, Optimization and Register Allocation Implementation

The intermediate representation (IR) used here is an SSA implentation, partially based on the LLVM IR. There are a few differences: the IR used here uses a typesystem analogous to C's, and uses a C-like syntax for textual representation as well.

LLVM IR's basic syntax:

identifier:	*temp-identifier*
identifier:	*global-identifier*
temp-identifier:	**%** *word*
global-identifier:	**@** *word*
instruction:	[*temp-identifier* =] *instruction-name* [*parameter-list*] *newline*
parameter-list:	*expression* [**,** *parameter-list*]
expression:	*type identifier*
expression:	*type constant*
expression:	**label** *temp-identifier*
label:	*word* **:**

Custom IR's C-like syntax:

identifier:	*temp-identifier*
identifier:	*label-identifier*
temp-identifier:	***t*** *word*
label-identifier:	***L*** *word*
instruction:	[*type temp-identifier* =] *instruction-name* **(** [*parameter-list*] **) ;**
parameter-list:	*expression* [**,** *parameter-list*]
expression:	*identifier*
expression:	**(** *type* **)** *constant*
label:	*label-identifier* **:**

The custom IR used also lacks implicit blocks

Concretlely, the following LLVM snippet:

```
%1 = add i32 1, i32 2
%2 = icmp eq i32 %1, i32 3
br i1 %2, label %3, label %0
ret i32 0
```

Is equivalent to the following custom-IR:

```
L0:
      int t1 = add((int)1, (int)2);
      _Bool t2 = cmp eq(t1, (int)3);
      split(t2, L3, L0);
L3:
      ret((int)0);
```

Both languages feature an *undef* constant, with the ability to take on any value it likes. The custom IR lacks a *null* constant, it uses *0* instead.

A working C front-end is expected to be present, and snippets here will be either in custom IR or C.

Optimizations

Constant folding

Problem

When a programmer types something in his code looking vaguely like this:

```
int i = 10 – 3 * 2;
```

The compiler can be expected to see that i should be initialised to four, rather than having it emit instructions for each mathematical operation. Moreover, if a programmer types:

```
int a = 10;
int b = a * 2;
```

The compiler can also be expected to simplify the initialisation of b into an initialisation to twenty. Although perhaps trivially optimised manually, a programmer shouldn't have to be expected to write his constants in the least computationally complex way, simply because writing out an equation usually leads to a far better understanding of why a variable has a certain value.

Consider the following code:

```
int speed = 10;
int time = 2;
int distance = speed * time;
```

As opposed to simply:

```
int distance = 20;
```

Not only is the first far more maintainable, but also a lot more clear. Although it's obvious that distance has value twenty, it's now also clear why it has that value. Therefore the compiler may not expect the programmer to manually *fold* his constants. Instead, the compiler calculates these constants in a process known as *constant folding* (or *constant propagation*).

Implementation

In order to perform any useful consant folding, the compiler needs to fill in constants for variables where possible, so code of the form:

```
int a = 10;
int b = a * a;
return b – a;
```

Becomes:

```
int b = 10 * 10;
return b – 10;
```

Once the value of b is determined, it should then also be filled in, to fold further. Since the value of b is 100, it can be used to fill in the return expression:

```
return 100 – 10;
```

This value can then be folded once more to yield the value 90:

```
      return 90;
```

This algorithm might look quite involved, but its simplicity is actually staggering. It simply depends on *phiability optimisation*. Phiability optimisation fills in constants for variables automatically. Consider the first fragment's IR before phiability optimisation:

```
L0:
      int* t1 = alloca(int);
      int* t2 = alloca(int);
      store((int)10, t1);
      int t3 = load(t1);
      int t4 = load(t1);
      int t5 = mul(t3, t4);
      store(t5, t2);
      int t6 = load(t2);
      int t7 = load(t1);
      int t8 = sub(t6, t7);
      ret(t8);
```

The variables still exist in their crude memory form. However, their values are propagated automatically once phiability optimisation occurs:

```
L0:
      int t1 = mul((int)10, (int)10);
      int t2 = sub(t1, (int)10);
      ret(t2);
```

The constants can now be propagated with a pass that scans for computable instructions (arithmetic instructions of which both operands are constants) and computes their values, filling them in for all future occurrences:

```
L0:
      ret((int)90);
```

Considerations

Platform incompatibilities

There is a way compiler-based constant folding might stand in the way of the programmer. Mostly the compiler can do this when folding away instructions operating on floating point operands, because different targets may compute floating point operations differently. Therefore, cross-compilation becomes an issue; if a floating point instruction for target Y normally yielding Vy, it yields Vx when folded away by target X, causing different semantics before and after optimisation.

A solution to this problem is to implement a floating point virtual machine for several targets, that use non IEEE floating point. Targets using IEEE floating point can use C99's internal way of computing IEEE floating point operations.

Since implementing such a system is non-trivial, code duplication needs to be avoided. If any other optimisation would need to be able to calculate an operation on two constants, it should run the same code. Therefore, the actual folding computations are performed outside of the optimiser, by a separate folding system.

Undef

Folding away undef constants is simple: the result of an arithmetic operation on one or more undef constants is an undef constant too. This has implications for phiability optimisation, however; consider the following example:

```
int a;
int b = a + 10;
return b – a;
```

The returned result is expected to be ten. Before phiability optimisation, the example would yield:

```
L0:
	int* t1 = alloca(int);
	int* t2 = alloca(int);
	int t3 = load(t1);
	int t4 = add(t3, (int)10);
	store(t4, t2);
	int t5 = load(t2);
	int t6 = load(t1);
	int t7 = sub(t5, t6);
	ret(t7);
```

The result is well-defined, because t3 can be expected to equal t6. Their contents are undefined, but it is important that they equal. If there is a phiability conversion where loading an ill-defined alloca yields an undef constant, those undef constants can't be guaranteed to equal eachother:

```
L0:
	int t1 = add((int)undef, (int)10);
	int t2 = sub(t1, (int)undef);
	ret(t2);
```

After constant folding, t1 yields undef, and t2, depending on t1, also yields undef:

```
L0:
	ret((int)undef);
```

Whereas the actual result is well-defined.

A solution to this problem is to only yield one, constant, undefined instance for the variable, instead of an undef constant. A new instruction needs to be introduced. Now after phiability optimisation:

```
L0:
	int t1 = undef();
	int t2 = add(t1, (int)10);
	int t3 = sub(t3, t1);
	ret(t3);
```

No constants are folded now. The optimisation is done through peep-hole optimizations for the sub instruction, so the well-defined end-result becomes:

```
L0:
	ret((int)10);
```

There are certain cases where constant folding gives the desired outcome and that aren't optimised anymore now. Consider:

```
	int a;
```

```
      return a + 10;
```

The optimized IR:

```
L0:
      int t1 = undef();
      int t2 = add(t1, (int)10);
      ret(t2);
```

An undef constant can be used for t2, since no value that depends on t2 also depends on t1. The expression evaluator has to carefully consider which cases to fold, based on dependency trees.