hack.ast.instruction

hack.ast.instruction: The Instructions of Hack

```
Require Import hack.ast.reg.
Require Import hack.result.
```

The hack computer has a rather unconventional ISA with only two kinds of instructions:

- Address instruction. This is of the form `@ 15-bit` address. In our case, we parameterize instructions over the address type so as to make code generation flexible.
- Computation instruction is a combination of assignment and control flow. In general it is of the form `destination=output;jump`

```
Inductive output :=
  | constant : const -> output
  | uapply : unary -> reg.t -> output
  | bapply : binary -> reg.AOrM -> output
  with const := Zero | One | MinusOne
  with unary := BNeg | UMinus | Succ | Pred | ID
  with binary := Add | Sub | SubFrom | BAnd | BOr
  with jump :=
  | JGT
  | JEQ
  | JGE
  | JLT
  | JNE
  | JLE
  | JMP.

Variant t v :=
  | C : list reg.t -> output -> option jump -> t v
  | At : v -> t v.
```

Mapping over and resolving address types

Two important and natural functions in the context of instruction are `map` and `resolve`. The `map` function lift a function `f : u -> v` over address types to functions on the corresponding instruction types `map f : t u -> t v`. If instead we had a resolution function `f : u -> option v`, we use the `resolve` function for address resolution of address type `u` (think of symbolic address) to `v` (think of 15-bit address).

```
Definition map {u v}(f : u -> v)(i : t u) : t v :=
  match i with
  | C d o oj => C d o oj
  | At x => At (f x)
  end.

Definition resolve {u v}(f : u -> option v)(i : t u) : result.t (t v) u :=
  match i with
  | C d o oj => inl (C d o oj)
  | At x => result.option.app (option_map At \o f) x
  end.
```

Index
