Register allocation

Let us consider a procedure using N variables that we want to compile for a CPU with K physical registers. Through liveness analysis, the compiler identified all pairs of variables that are in use at the same time. Such variables are said to *interfere* with each other. We want to assign CPU registers to as many variables as possible, while ensuring that no two interfering variables are assigned the same register. Any variable for which no register is allocated must be *spilled* into the main memory. Some of the variables get assigned fixed registers from the start, due to the calling conventions of the target architecture. This assignment cannot be changed. Finally, if a non-interfering pair of variables occurs in a trivial assignment u `:=` v, it may be desirable to assign u and v the same register and avoid a superfluous "move" instruction in the compiled code.

The Iterated Register Coalescing (IRC) algorithm for register allocation was proposed by George and Appel [TOPLAS 18(3), 1996] and verified in Coq by Blazy, Robillard, and Appel [ESOP 2010, LNCS 6012]. Below, we describe a simplified variant of the algorithm. The state of the algorithm is represented by two dictionaries, G and A:

- Dictionary G represents the (symmetric and irreflexive) interference relation, and $G[v]$ denotes the set of variables that interfere with variable v. Procedure `remove(`v`)` removes v from the domain of G and from every set $G[w]$. Procedure `merge(`v`,`u`)` replaces v with u in every set $G[w]$, replaces $G[u]$ with $G[u] \cup G[v]$, and removes v from the domain of G.

- Dictionary A represents the register assignment, and for any variable v in the domain of A, $A[v]$ is either a physical register or a special value **`spill`**. Given a set of variables $S \subseteq \mathrm{dom}\, A$, function `available(`$S$`)` returns either some register that is not assigned to any variable in S, or **`spill`**, if there is no register available. At the beginning of the algorithm, A contains a pre-assignment of physical registers to some of the variables in the domain of G.

We assume to have at our disposal two oracle functions that implement various heuristics of the George-Appel algorithm:

- `pick()` returns either some variable v such that $v \in \mathrm{dom}\, G$ and $v \notin \mathrm{dom}\, A$, or a special value **`none`**, if there is no such variable.

- `coalesce(`v`)` returns either **`none`** or some variable u such that $u \neq v$, $u \in \mathrm{dom}\, G$, and $v \notin G(u)$.

Here is a recursive implementation of the IRC algorithm in pseudo-code:

```
procedure IRC()
    var v := pick()
    if v = none then return
    var u := coalesce(v)
    if u = none then
        var S := G[v]
        remove(v)
        IRC()
        A[v] := available(S)
    else
        merge(v,u)
        IRC()
        A[v] := A[u]
    end-if
end-procedure
```

Implementation task. Implement the IRC procedure in your favorite language. The choice of data types for variables, registers, sets, and dictionaries is up to you. You may use iteration instead of recursion, or persistent data structures instead of mutable ones. You should implement the functionality of `remove()`, `merge()`, and `available()`.

You may leave the oracle functions `pick()` and `coalesce()` without implementation. If you do provide one, please note that the rest of the algorithm should not rely on any specific implementation details, beyond what is stated about `pick()` and `coalesce()` above.

Verification tasks. You can assume that at the start of the procedure, dom $A \subseteq$ dom G and no two interfering variables are pre-assigned the same physical register. Verify the following properties:

1. Every variable in the initial domain of G gets an assignment in A at the end of the procedure.
2. If a variable has a register pre-assigned to it, this assignment does not change at the end.
3. No two interfering variables (w.r.t. the initial value of G) are assigned the same register.
4. The procedure terminates.

Challenge Lite. Consider starting with the following simplified variant of the IRC procedure:

```
procedure IRClite()
    var v := pick()
    if v = none then return
    var S := G[v]
    remove(v)
    IRClite()
    A[v] := available(S)
end-procedure
```

In this version, variables are never coalesced and the `merge()` procedure is not necessary.