Semantic correctness of some compiler optimizations based on dataflow analysis

Xavier Leroy
INRIA Rocquencourt

Draft, version 0.1, June 26, 2003

1 Introduction

This report presents a Coq proof of semantic preservation for some classic compiler optimizations: constant propagation, register allocation, dead code elimination. These optimizations are carried out on an intermediate language similar to what is called "three-address code" or "register transfer language" in the literature. A formal operational semantics for this language is defined.

Classically, each optimization consists of two steps: a static analysis step, expressed in the general framework of dataflow analysis; and a program transformation step, which rewrites the input program according to the results of the analysis. For instance, in the case of constant propagation, the static analysis identifies program points where registers have known compile-time values; the program transformation, then, replaces the instructions that compute these values by "load immediate" instructions.

Both the static analyses and the program transformations are formally defined as Coq functions. Their correctness is then proved by showing a semantic equivalence result: if the original program terminates on some final memory state, the transformed program also terminates and produces the same memory state. This result is proved by induction on the execution of the original program, using a form of simulation lemma (if the original program executes one instruction, the transformed program executes zero, one or several related instructions).

This report is organized as follows.

- Sections 2 to 6 define auxiliary notions such as finite maps, semi-lattices, and well-founded iteration. This "scaffolding" is not compiler-specific and can be skimmed through quickly.
- Section 7 defines a generic framework for forward and backward dataflow analyses, and formalizes and proves correct a solver for dataflow problems based on Kildall's worklist algorithm.
- Section 10 defines the intermediate language and its operational semantics, using notions of pseudo-registers and machine-level operations defined in sectionss 8 and 9.
- Sections 11, 13 and 14 are the meat of this development: each section defines an optimization pass and proves its semantic correctness. Section 11 deals with constant propagation. Section 13 is on register allocation, using an axiomatization of graph coloring given in section 12. Finally, section 14 deals with dead code elimination.

- Some concluding remarks are given in section 15.

2 Module *Misc*: utilities

The module *Misc* provides various general-purpose definitions, lemmas and tactics that are missing from the Coq distribution.

Require *Relations*. Require *PolyList*. Require *Lt*. Require *Compare_dec*.

2.1 Notations

The following definition allows to write *Nil* instead of (*nil some-type*).

Syntactic Definition *Nil* := (*nil* ?).

Notation for "not equal"

Notation "$a \neq b$" := $\neg(a = b)$.

2.2 Reflexive closure of a relation

Section *Refl_closure*.

Variable A: *Set*.
Variable r: $A \to A \to Prop$.

Definition of the reflexive closure of relation r.

Definition *refl_closure* := $[x,y:A]\ x{=}y \vee (r\ x\ y)$.

Hypothesis *trans_r*: (*transitive A r*).

The reflexive closure is transitive if the base relation is.

Lemma *transitive_refl_closure*: (*transitive A refl_closure*).

Lemma *trans1_refl_closure*:
$(x,y,z:A)\ (r\ x\ y) \to (refl\_closure\ y\ z) \to (r\ x\ z)$.

Lemma *trans2_refl_closure*:
$(x,y,z:A)\ (refl\_closure\ x\ y) \to (r\ y\ z) \to (r\ x\ z)$.

End *Refl_closure*.

2.3 Useful lemmas

Lemma *lt_not_eq*: $(x,y{:}nat)\ (lt\ x\ y) \to x \neq y$.

Lemma *lt_ge_dec*: $(x,y{:}nat)\ \{(lt\ x\ y)\} + \{(le\ y\ x)\}$.

Lemma *not_in_cons*:
$(a{:}nat)(succs{:}(list\ nat))(s{:}nat)$
$\neg(In\ s\ (cons\ a\ succs)) \to s \neq a \wedge \neg(In\ s\ succs)$.

Implicit Arguments On.

Lemma *in_singleton*: $(A{:}Set)(x{:}A)(y{:}A)\ (In\ x\ (cons\ y\ (nil\ A))) \to x{=}y$.

Implicit Arguments Off.

2.4 Useful tactics

The *CaseEq* tactic performs case analysis like *Case*, but it introduces an additional equality on the term being subject to case analysis.

Tactic Definition *CaseEq name* :=
Generalize (*refl_equal* ? *name*); *Pattern* -1 *name*; *Case name*.

The *IntroElim* tactic is like *Intros*, except that conjunctions, disjunctions and existential quantifications are automatically eliminated and simplified before being introduced in the hypotheses.

```
Recursive Tactic Definition IntroElim :=
  Match Context With
    [ ⊢ (id: (EX x | ?)) ? ] → Intro id; Elim id; Clear id; IntroElim
  | [ ⊢ (id: ? ∧ ?) ? ] → Intro id; Elim id; Clear id; IntroElim
  | [ ⊢ (id: ? ∨ ?) ? ] → Intro id; Elim id; Clear id; IntroElim
  | [ ⊢ ? → ? ] → Intro; IntroElim
  | _ → Idtac.
```

The *MyElim* tactic is like *Elim*, except that it automatically eliminates and simplifies conjunctions, disjunctions and existential quantifications in the term being eliminated.

Tactic Definition *MyElim n* :=
Elim n; *IntroElim*.

3 Module *While*: well-founded while loops

The module *While* provides a simple, easy to comprehend case of well-founded induction, corresponding to a "while" loop in an imperative language.

Require *Wf*. Require *Sumbool*.

Section *While*.

The "while" loop we axiomatize here corresponds to the following pseudo-code:

```
x <- start;
while not (stop x) do
  x <- (trans x);
done;
x
```

In the following, A is the domain of the iteration; *trans* is thetransition function from one loop iteration to the next; and *stop* isthe stop criterion.

Variable A: Set.
Variable $trans$: $A \to A$.
Variable $stop$: $A \to bool$.

To guarantee termination of the loop, A must be equipped with a well-founded ordering, and each loop iteration must be strictly decreasing.

Variable ltA: $A \to A \to Prop$.
Hypothesis $well\_founded\_ltA$: $(well\_founded\ A\ ltA)$.
Hypothesis $trans\_decreasing$:
$(x{:}A)\ (stop\ x) = false \to (ltA\ (trans\ x)\ x)$.

We define the result of the while loop by well-founded induction.

Definition $while\_trans : (x{:}A)((y{:}A)(ltA\ y\ x){\to}A){\to}A :=$
$[x{:}A]\ [f{:}\ (y{:}A)(ltA\ y\ x) \to A]$
$Cases\ (sumbool\_of\_bool\ (stop\ x))\ of$
$(left\ \_) \Rightarrow x$
$\mid (right\ pfalse) \Rightarrow (f\ (trans\ x)\ (trans\_decreasing\ x\ pfalse))$
end.

Definition $while : A \to A :=$
$(fix\ A\ ltA\ well\_founded\_ltA\ ([x{:}A]A)\ while\_trans)$.

The transition step satisfies the extensionality properties requiredto show that the result of *while* is a fixpoint.

Remark $while\_trans\_extensional$:
$(x{:}A)\ (f,g{:}\ (y{:}A)(ltA\ y\ x){\to}A)$
$((y{:}A;\ p{:}(ltA\ y\ x))(f\ y\ p){=}(g\ y\ p)) \to$
$(while\_trans\ x\ f){=}(while\_trans\ x\ g)$.

while does compute a fixpoint of the *trans* function.

Lemma $while\_is\_fixpoint$:
$(x{:}A)\ (while\ x) = (while\_trans\ x\ ([y{:}A][\_{:}(ltA\ y\ x)]\ (while\ y)))$.

The *stop* condition holds for the result of *while*.

Lemma $while\_satisfies\_stop$: $(x{:}A)\ (stop\ (while\ x)) = true$.

If *transf* preserves an invariant P, and the start value ofthe while loop satisfies P, then its final value satisfies P as well.

Variable P: $A \to Prop$.
Hypothesis *trans_preserves_P*: $(x{:}A)\ (stop\ x) = false \to (P\ x) \to (P\ (trans\ x))$.

Lemma *while_preserves_invariant*: $(x{:}A)\ (P\ x) \to (P\ (while\ x))$.

End *While*.

4 Module *WFOrder*: well-founded orderings

The *WFOrder* module provides generic constructions of well-founded orderings. Currently, only one such construction is needed: lexicographic product of two orderings. The Coq standard library does define lexicographic product, but for dependent products, which I don't understand how to use. So, here is a more pedestrian definition for plain, non-dependent products.

Require *Wellfounded*. Require *Relations*.

Section *Pair_Lexico*.

Variable A,B: Set.
Variable gtA: $A \to A \to Prop$.
Variable gtB: $B \to B \to Prop$.
Hypothesis *wfA*: $(well\_founded\ A\ gtA)$.
Hypothesis *wfB*: $(well\_founded\ B\ gtB)$.

Inductive $gt\_pair : A{\times}B \to A{\times}B \to Prop :=$
gt_pair_fst:
$(a{:}A)(b{:}B)(a'{:}A)(b'{:}B)$
$(gtA\ a\ a') \to (gt\_pair\ (a,b)\ (a',b'))$
| gt_pair_snd:
$(a{:}A)(b{:}B)(b'{:}B)$
$(gtB\ b\ b') \to (gt\_pair\ (a,b)\ (a,b'))$.

Remark *acc_gt*:
$(a{:}A)\ (Acc\ A\ gtA\ a) \to$
$(b{:}B)\ (Acc\ B\ gtB\ b) \to$
$(Acc\ (A{\times}B)\ gt\_pair\ (a,b))$.

Lemma *wf_gt_pair*: $(well\_founded\ (A{\times}B)\ gt\_pair)$.

Hypothesis *transitive_gtA*: $(transitive\ A\ gtA)$.
Hypothesis *transitive_gtB*: $(transitive\ B\ gtB)$.

Lemma *transitive_gt_pair*: $(transitive\ (A{\times}B)\ gt\_pair)$.

End *Pair_Lexico*.

5 Module *Lattice*: semi-lattices

The module *Lattice* defines a generic interface for well-founded semi-lattices, along with several useful lattice constructions (flat lattices, product lattices).

Require *Relations*. Require *Misc*. Require *WFOrder*.

5.1 Signature of a semi-lattice

Module Type *SEMILATTICE*.

The carrier type.
Parameter T: *Set*.

The "greater than" ordering, which must be well-founded and transitive.
Parameter *gt*: $T \to T \to Prop$.
Parameter *wf*: (*well_founded T gt*).
Parameter *trans*: (*transitive T gt*).

The "greater or equal" ordering derived from *gt*.
Definition *ge* := (*refl_closure T gt*).

Equality is decidable.
Parameter *eq_dec*: $(x,y{:}T)$ $\{x{=}y\} + \{x{\neq}y\}$.

Greatest and smallest elements of the semi-lattice.
Parameter *top*: T.
Parameter *bot*: T.
Parameter *ge_top_x*: $(x{:}T)$ (*ge top x*).
Parameter *ge_x_bot*: $(x{:}T)$ (*ge x bot*).

The least upper bound operation, which must be commutative and majorate its first argument (and also its second argument, by commutativity). Notice that we do not require *lub* to be the smallest majorant.
Parameter *lub*: $T \to T \to T$.
Parameter *lub_commut*: $(x,y{:}T)$ (*lub x y*) = (*lub y x*).
Parameter *ge_lub_left*: $(x,y{:}T)$ (*ge* (*lub x y*) *x*).

End *SEMILATTICE*.

5.2 The semi-lattice of booleans

Module *Boolean* <: *SEMILATTICE*.

The wonderful Coq 7.4 module mechanism refuses to implement a type parameter by an inductive type. Thus, we define the inductive type as $T\_$, and let T be an alias for it.

Inductive $T\_$: *Set* := *Bot* : $T\_$ | *Top* : $T\_$.

Definition $T := T\_$.

Same trick for the *gt* ordering, which is defined via the inductive predicate $gt\_$.

Inductive $gt\_$: $T \to T \to Prop$:=
gt_intro: (*gt_ Top Bot*).

Definition *gt* := *gt_*.

Remark *acc_Top*: (*Acc T gt Top*).

Remark *acc_Bot*: (*Acc T gt Bot*).

Lemma *wf*: (*well_founded T gt*).

Lemma *trans*: (*transitive T gt*).

Lemma *eq_dec*: (*x*,*y*:*T*) {*x*=*y*} + {*x*≠*y*}.

Definition *ge* := (*refl_closure T gt*).

Definition *top* := *Top*.
Definition *bot* := *Bot*.

Lemma *ge_top_x*: (*x*:*T*) (*ge top x*).

Lemma *ge_x_bot*: (*x*:*T*) (*ge x bot*).

Definition *lub*[*x*,*y*: *T*] :=
Cases x of
Bot ⇒ *y*
| *Top* ⇒ *Top*
end.

Lemma *lub_commut*: (*x*,*y*:*T*) (*lub x y*) = (*lub y x*).

Lemma *ge_lub_left*: (*x*,*y*:*T*) (*ge* (*lub x y*) *x*).

End *Boolean*.

5.3 Flat semi-lattice

The *Flat* functor below defines the flat semi-lattice over any type for which equality is decidable.

Module Type *TYPE_DEC_EQ*.
Parameter *T*: *Set*.
Parameter *eq_dec*: (*x*,*y*:*T*) {*x*=*y*} + {*x*≠*y*}.
End *TYPE_DEC_EQ*.

Module *Flat*[*A*: *TYPE_DEC_EQ*].

Elements of the flat semi-lattice are either *Top*, *Bot*, or *Inj x* where *x* is an element of the argument type.

Inductive *T_*: *Set* := *Bot* : *T_* | *Inj*: *A.T* → *T_* | *Top* : *T_*.
Definition *T* := *T_*.
Definition *top* := *Top*.
Definition *bot* := *Bot*.
Definition *inj* := *Inj*.

Inductive *gt_*: *T* → *T* → *Prop* :=
gt_inj_bot: (*x*: *A.T*) (*gt_* (*Inj x*) *Bot*)
| *gt_top_bot*: (*gt_ Top Bot*)
| *gt_top_inj*: (*x*: *A.T*) (*gt_ Top* (*Inj x*)).

Definition $gt := gt\_$.

Remark acc_Top: $(Acc\ T\ gt\ Top)$.

Remark acc_Inj: $(x{:}\ A.T)\ (Acc\ T\ gt\ (Inj\ x))$.

Remark acc_Bot: $(Acc\ T\ gt\ Bot)$.

Lemma wf: $(well\_founded\ T\ gt)$.

Lemma $trans$: $(transitive\ T\ gt)$.

Definition $ge := (refl\_closure\ T\ gt)$.

Lemma eq_dec: $(x,y{:}T)\ \{x{=}y\} + \{x{\neq}y\}$.

Lemma ge_top_x: $(x{:}T)\ (ge\ top\ x)$.

Lemma ge_x_bot: $(x{:}T)\ (ge\ x\ bot)$.

```
Definition lub [x,y:T] :=
  Cases x of
    Bot ⇒ y
  | Top ⇒ Top
  | (Inj a) ⇒
      Cases y of
        Bot ⇒ (Inj a)
      | Top ⇒ Top
      | (Inj b) ⇒
          Cases (A.eq_dec a b) of
            (left _) ⇒ (Inj a)
          | (right _) ⇒ Top
          end
      end
  end.
```

Lemma lub_commut: $(x,y{:}T)\ (lub\ x\ y) = (lub\ y\ x)$.

Lemma ge_lub_left: $(x,y{:}T)\ (ge\ (lub\ x\ y)\ x)$.

End *Flat*.

5.4 Product semi-lattice

The *Product* functor below defines the product of two semi-lattices, ordered lexicographically.

Module $Product[A,B{:}\ SEMILATTICE]$.

Definition $T := A.T \times B.T$.

Definition $gt := (gt\_pair\ A.T\ B.T\ A.gt\ B.gt)$.

Lemma wf: $(well\_founded\ T\ gt)$.

Definition $top := (A.top,\ B.top)$.

Definition $bot := (A.bot,\ B.bot)$.

Definition $ge := (refl\_closure\ T\ gt)$.

Lemma $ge\_pointwise$:
$(a{:}A.T)(b{:}B.T)(a'{:}A.T)(b'{:}B.T)$
$(A.ge\ a\ a') \to (B.ge\ b\ b') \to (ge\ (a,b)\ (a',b'))$.

Lemma ge_top_x: $(x{:}T)\ (ge\ top\ x)$.

Lemma ge_x_bot: $(x{:}T)\ (ge\ x\ bot)$.

Definition $lub[p,q{:}T] := ((A.lub\ (Fst\ p)\ (Fst\ q)),\ (B.lub\ (Snd\ p)\ (Snd\ q)))$.

Lemma lub_commut: $(x,y{:}T)\ (lub\ x\ y) = (lub\ y\ x)$.

Lemma ge_lub_left: $(x,y{:}T)\ (ge\ (lub\ x\ y)\ x)$.

End *Product*.

6 Module *Map*: finite integer maps

The module *Map* axiomatizes finite maps from integers to some type. This axiomatization is intended to be implemented by an efficient persistent data structure such as balanced trees or Patricia trees.

Require *Wellfounded*. Require *WFOrder*. Require *Misc*. Require *Relations*.
Require *Lt*. Require *Le*. Require *Peano_dec*.

6.1 Axiomatization of maps

map A is the type of maps from integers to values of type *A*.
Axiom *map*: $Set \to Set$.

Section *Map*.

Variable *A*: *Set*.

Maps are used via three functions:

- *map_init v* returns a map that associates *v* to all integers.
- *map_get i m* returns the value of integer *i* in map *m*.
- *map_set i v m* returns a map identical to *m*, except that integer *i* is mapped to *v*.

Axiom *map_init*: $A \to (map\ A)$.
Axiom *map_get*: $nat \to (map\ A) \to A$.
Axiom *map_set*: $nat \to A \to (map\ A) \to (map\ A)$.

A correct implementation of maps could be: ■ Definition map *A*: *Set* := nat -> A. Definition map_init *x*: *A* := *n*: *nat* x. Definition map_get *i*: *nat*; *m*: (*map A*) := (m i). Definition map_set *i*: *nat*; *v*: *A*; *m*: (*map A*) := *n*: *nat* (if (beq_nat n i) then v else (m n)).■

However, this implementation is inefficient, so we just axiomatizethe properties we are interested in.

Axiom *map_get_init*:
(*a*:*A*)(*n*:*nat*) (*map_get n* (*map_init a*)) = *a*.
Axiom *map_get_set_same*:
(*m*:(*map A*))(*n*:*nat*)(*a*:*A*) (*map_get n* (*map_set n a m*)) = *a*.
Axiom *map_get_set_other*:
(*m*:(*map A*))(*n*,*n'*:*nat*)(*a*:*A*)
n≠*n'* → (*map_get n'* (*map_set n a m*)) = (*map_get n' m*).

End *Map*.

To help manipulate maps, we introduce the following convenient notations, reminiscent of OCaml's array notations: $m.(i)$ for *map_get*, and $m.(i) \leftarrow v$ for *map_set*.

Notation "*a* .(*b*) ← *c*" := (*map_set* ? *b c a*).
Notation "*a* .(*b*)" := (*map_get* ? *b a*).

6.2 Ordering on maps

We now define lexicographic and pointwise orderings over maps on a well-founded type.

Section *Map_Ordering*.

Variable *A*: *Set*.
Variable *gtA*: *A* → *A* → *Prop*.
Hypothesis *wf_gtA*: (*well_founded A gtA*).

map_gt_lexico n p q holds if the tuple ($p.(0)$, ..., $p.(n\text{-}1)$) is lexicographically greater than the tuple ($q.(0)$, ..., $q.(n\text{-}1)$).
Inductive *map_gt_lexico*: *nat* → (*map A*) → (*map A*) → *Prop* :=
map_gt_lexico_more:
(*n*:*nat*)(*p*,*q*: (*map A*))
(*gtA p.*(*n*) *q.*(*n*)) →
(*map_gt_lexico* (*S n*) *p q*)
| *map_gt_lexico_same*:
(*n*:*nat*)(*p*,*q*: (*map A*))
p.(*n*) = *q.*(*n*) →
(*map_gt_lexico n p q*) →
(*map_gt_lexico* (*S n*) *p q*).

map_gt_lexico n is a well-founded ordering.

Remark *well_founded_map_gt_lexico_ind*:
(*n*:*nat*)
(*well_founded* (*map A*) (*map_gt_lexico n*)) →
(*well_founded* (*map A*) (*map_gt_lexico* (*S n*))).

Lemma *well_founded_map_gt_lexico*:

(*n*:*nat*) (*well_founded* (*map A*) (*map_gt_lexico n*)).

For most uses, it turns out that we do not want the lexicographic ordering between maps, but rather the pointwise ordering: $p \geq q$ iff $p.(i) \geq q.(i)$ for all $0 \leq i < n$. We now define the corresponding "greater than" pointwise ordering between maps.

Local *geA* := (*refl_closure A gtA*).

Inductive *map_gt*: *nat* → (*map A*) → (*map A*) → *Prop* :=
map_gt_intro:
(*range*:*nat*) (*p*,*q*: (*map A*))
((*n*:*nat*) (*lt n range*) → (*geA p.*(*n*) *q.*(*n*))) →
(*n*:*nat*) (*lt n range*) → (*gtA p.*(*n*) *q.*(*n*)) →
(*map_gt range p q*).

The pointwise ordering is coarser than the lexicographic ordering. Consequently, the pointwise ordering is well founded.

Lemma *map_gt_inclusion*:
(*p*,*q*: (*map A*)) (*range*:*nat*)
(*map_gt range p q*) → (*map_gt_lexico range p q*).

Lemma *well_founded_map_gt*:
(*range*:*nat*) (*well_founded* (*map A*) (*map_gt range*)).

The pointwise ordering is transitive.

Hypothesis *transitive_gtA*: (*transitive A gtA*).

Lemma *transitive_map_gt*:
(*range*:*nat*) (*transitive* (*map A*) (*map_gt range*)).

Obviously, setting a map entry to a value greater than its previous value results in a greater map.

Lemma *map_update_gt*:
(*range*:*nat*) (*p*: (*map A*)) (*n*:*nat*) (*a*:*A*)
(*lt n range*) →
(*gtA a p.*(*n*)) →
(*map_gt range* (*p.*(*n*) ← *a*) *p*).

It is decidable whether two maps are equal on their first n entries.

Hypothesis *eqA_dec*: (*a*,*b*: *A*) {*a*=*b*} + {*a*≠*b*}.

Lemma *map_eq_dec*:
(*range*:*nat*) (*p*,*q*: (*map A*))
{ (*n*:*nat*) (*lt n range*) → *p.*(*n*) = *q.*(*n*) }
+ { (*EX n* | (*lt n range*) ∧ *p.*(*n*) ≠ *q.*(*n*)) }.

The reflexive closure of *map_gt* is indeed the pointwise "greater or equal" relation.

Lemma *ge_map_pointwise_1*:
(*range*:*nat*)(*p*,*q*: (*map A*))
(*refl_closure* (*map A*) (*map_gt range*) *p q*) →

$(n{:}nat)\ (lt\ n\ range) \to (geA\ p.(n)\ q.(n))$.

Lemma *ge_map_pointwise_2*:
$(range{:}nat)(p,q{:}\ (map\ A))$
$((n{:}nat)\ (lt\ n\ range) \to (geA\ p.(n)\ q.(n))) \to$
$((n{:}nat)\ (lt\ n\ range) \to p.(n) = q.(n))$
$\vee\ (map\_gt\ range\ p\ q)$.

End *Map_Ordering*.

6.3 Applying a function to all elements of a map

We now define pointwise transformation of a map through a function: *map_apply f range m* returns a map m' such that $m'.(i) = (f\ i\ m.(i))$ for all $0 \leq i < range$.

Section *Map_apply*.

Variable A, B: *Set*.
Variable f: $nat \to A \to B$.

Fixpoint *map_apply_rec* $[p{:}(map\ A);\ q{:}(map\ B);\ n{:}nat] : (map\ B) :=$
Cases n of
$O \Rightarrow q$
$|\ (S\ m) \Rightarrow (map\_apply\_rec\ p\ (q.(m) \leftarrow (f\ m\ p.(m)))\ m)$
end.

Definition *map_apply* $[p{:}(map\ A);\ n{:}nat] : (map\ B) :=$
Cases n of
$O \Rightarrow (map\_init\ B\ (f\ O\ p.(O)))$
$|\ (S\ m) \Rightarrow (map\_apply\_rec\ p\ (map\_init\ B\ (f\ m\ p.(m)))\ m)$
end.

Remark *map_apply_rec_transf*:
$(range{:}nat)\ (p{:}\ (map\ A))\ (q{:}\ (map\ B))$
$(n{:}nat)$
$((lt\ n\ range) \to (map\_apply\_rec\ p\ q\ range).(n) = (f\ n\ p.(n)))$
$\wedge\ ((le\ range\ n) \to (map\_apply\_rec\ p\ q\ range).(n) = q.(n))$.

Lemma *map_apply_transf*:
$(range{:}nat)\ (p{:}\ (map\ A))$
$(n{:}nat)\ (lt\ n\ range) \to (map\_apply\ p\ range).(n) = (f\ n\ p.(n))$.

End *Map_apply*.

7 Module *Kildall*: solving dataflow inequations

The module *Kildall* defines and proves correct Kildall's worklist algorithm for solving dataflow inequations.

Require *PolyList*. Require *Wf_nat*. Require *Relations*.

Require *Le*. Require *Lt*. Require *Peano_dec*.
Require *Misc*. Require *Map*. Require *WFOrder*. Require *Lattice*. Require *While*.

7.1 Export interface

This is the generic interface of a dataflow inequation solver.

Module Type *DATAFLOW_SOLVER*.

The semi-lattice to which values of the unknowns belong.
Declare Module *L*: *SEMILATTICE*.

This is Kildall's solver function. It takes the following arguments:

- *last_node* and *successors* define the flow graph: nodes of this graph are integers between 0 and *last_node* (included); for each node n, *successors* n is the list of successor nodes for n. The hypothesis *successors_in_graph* says that these successor nodes are themselves between 0 and *last_node* (included).
- *transf* is the transfer function for the inequations: the inequations being solved are $X(s) \geq (transf\ n\ X(n))$ for each node n and successor s of n.
- *entry_points* is a list of (node, value) pairs. For each (n, v) in *entry_points*, the inequation $X(n) \geq v$ is added.

The result of the *fixpoint* function is a map from nodes to values of the lattice L representing a solution X to the dataflow inequations.

Parameter *fixpoint*:
(*last_node*: *nat*)
(*successors*: (*nat*→(*list nat*)))
(*successors_in_graph*: (*n*,*s*:*nat*)(*In s* (*successors n*))→(*le s last_node*))
(*transf*: *nat*→*L.T*→*L.T*)
(*entry_points*: (*list nat*×*L.T*))
(*map L.T*).

The following theorem shows that the solution returned by *fixpoint* satisfies the inequation $X(s) \geq (transf\ n\ X(n))$ for every node n and successor s of n.

Parameter *fixpoint_solution*:
(*last_node*:*nat*)
(*successors*: *nat*→(*list nat*))
(*successors_in_graph*: (*n*,*s*:*nat*)(*In s* (*successors n*))→(*le s last_node*))
(*transf*: *nat*→*L.T*→*L.T*)
(*entry_points*: (*list nat*×*L.T*))
(*n*,*s*:*nat*)
(*le n last_node*) →
(*In s* (*successors n*)) →
(*L.ge*
(*fixpoint last_node successors successors_in_graph transf entry_points*).(*s*)

(*transf* *n*
(*fixpoint last_node successors successors_in_graph transf entry_points*).(*n*))).

The following theorem shows that the solution returned by *fixpoint* satisfies the inequation $X(n) \geq v$ for every (node, value) pair (n, v) in *entry_points*.

Parameter *fixpoint_entry*:
(*last_node*:*nat*)
(*successors*: *nat*→(*list nat*))
(*successors_in_graph*: (*n*,*s*:*nat*)(*In s* (*successors n*))→(*le s last_node*))
(*transf*: *nat*→*L.T*→*L.T*)
(*entry_points*: (*list nat*×*L.T*))
(*entry_points_in_graph*:
(*n*:*nat*; *s*:*L.T*)(*In* (*n*,*s*) *entry_points*)→(*le n last_node*))
(*p*:*nat*; *l*:*L.T*)
(*In* (*p*,*l*) *entry_points*) →
(*L.ge* (*fixpoint last_node successors successors_in_graph transf entry_points*).(*p*) *l*).
End *DATAFLOW_SOLVER*.

7.2 Kildall's algorithm

We now define a solver for dataflow inequations based on Kildall's worklist algorithm.

Module *Dataflow_Solver*[*LAT*: *SEMILATTICE*]:
DATAFLOW_SOLVER with Module *L* := *LAT*.

Module *L* := *LAT*.

Section *Kildall*.

Parameters defining the flow graph.
Variable *last_node*: *nat*.
Variable *successors*: *nat* → (*list nat*).
Hypothesis *successors_in_graph*:
(*n*:*nat*) (*s*:*nat*) (*In s* (*successors n*)) → (*le s last_node*).

Parameters defining the analysis.
Variable *transf*: *nat* → *L.T* → *L.T*.
Variable *entry_points*: (*list* (*nat* × *L.T*)).
Hypothesis *entry_points_in_graph*:
(*n*:*nat*) (*s*:*L.T*) (*In* (*n*,*s*) *entry_points*) → (*le n last_node*).

7.2.1 Definition of the algorithm

Here is the algorithm in pseudocode:

```
in <- map_init bottom
wrk <- [0, 1 ,..., last_node]
```

```
foreach (pt, st) in entry_points do
  in <- (in.(pt) <- (lub in.(pt) st))
done
while wrk not empty do
  let n = head wrk
  wrk <- tail wrk
  let out = (transf n in.(n))
  foreach s in (successors n) do
    let i = lub in.(s) out
    if i <> in.(s) then
      in  <- (in.(s) <- i);
      wrk <- s :: wrk
    endif
  endfor
endwhile
return in
```

The state of the fixpoint computation is a pair (*in*, *wrk*) of a tentative solution to the dataflow problem (*in*) and a work list of nodes that need to be re-examined (*wrk*).

Definition *state* : *Set* := (*map L.T*) × (*list nat*).

Definition of the start state for the fixpoint computation.

Fixpoint *start_state_in* [*epts*: (*list* (*nat* × *L.T*))] : (*map L.T*) :=
Cases epts of
nil ⇒ (*map_init L.T L.bot*)
| (*cons* (*n*,*l*) *rem*) ⇒
let s = (*start_state_in rem*) *in* (*s*.(*n*) ← (*L.lub s*.(*n*) *l*))
end.

Fixpoint *start_state_wrk* [*n*: *nat*] : (*list nat*) :=
Cases n of
O ⇒ (*cons O* (*nil nat*))
| (*S m*) ⇒ (*cons* (*S m*) (*start_state_wrk m*))
end.

Definition *start_state* : *state* :=
(*start_state_in entry_points*, *start_state_wrk last_node*).

Definition of the "foreach *s* in (*successors n*)" part of the algorithm.

Definition *propagate_succ* [*ks*: *state*; *out*:*L.T*; *n*: *nat*] :=
let oldl = (*Fst ks*).(*n*) *in*
let newl = (*L.lub oldl out*) *in*
Cases (*L.eq_dec oldl newl*) *of*
(*left* _) ⇒ *ks*
| (*right* _) ⇒ (((*Fst ks*).(*n*) ← *newl*), (*cons n* (*Snd ks*)))
end.

```
Fixpoint propagate_succ_list
    [ks: state; out:L.T; succs: (list nat)] : state :=
  Cases succs of
    nil ⇒ ks
  | (cons n rem) ⇒
      (propagate_succ_list (propagate_succ ks out n) out rem)
  end.
```

The stop condition for the *while* loop: when the work list is empty.

```
Definition stop [ks: state] : bool :=
  Cases (Snd ks) of
    nil ⇒ true
  | (cons n rem) ⇒ false
  end.
```

The step function for the *while* loop: pick a node from the work list and update its successors.

```
Definition step [ks: state] : state :=
  Cases (Snd ks) of
    nil ⇒ ks
  | (cons n rem) ⇒
      (propagate_succ_list
        ((Fst ks), rem)
        (transf n ((Fst ks).(n)))
        (successors n))
  end.
```

Termination ordering for the *while* loop: either the solution increases, or the solution is unchanged but the length of the work list decreases.

```
Definition gt_state :=
  (gt_pair (map L.T) (list nat)
     (map_gt L.T L.gt (S last_node))
     (ltof (list nat) (!length nat))).

Definition ge_state := (refl_closure state gt_state).

Lemma gt_state_trans: (transitive state gt_state).

Lemma wf_gt_state: (well_founded state gt_state).
```

The *step* function for the *while* loop is increasing.

```
Lemma propagate_succ_increases:
  (ks: state) (out:L.T) (n:nat)
    (le n last_node) →
      (ge_state (propagate_succ ks out n) ks).

Lemma propagate_succ_list_increases:
  (nl:(list nat)) (ks: state) (out:L.T)
    ((n:nat) (In n nl) → (le n last_node)) →
```

$(ge\_state\ (propagate\_succ\_list\ ks\ out\ nl)\ ks)$.

Lemma *step_increases*:
$(ks\colon state)$
$(stop\ ks) = false \rightarrow$
$(gt\_state\ (step\ ks)\ ks)$.

We can finally define Kildall's algorithm using well-founded iteration as defined in the *While* module. The result of *fixpoint* is the final solution to the dataflow inequations.

Definition *while* :=
(*While.while state*
step stop
gt_state wf_gt_state
step_increases
start_state).

Definition *fixpoint* := $(Fst\ while)$.

7.2.2 Correctness invariant

The correctness invariant for Kildall's algorithm is as follows: for a state (in, wrk), in satisfies the dataflow inequations, except at nodes that belong to wrk.

Definition *good_state* $[ks\colon state]$:=
$(n{:}nat)$
$(le\ n\ last\_node) \rightarrow$
$\neg(In\ n\ (Snd\ ks)) \rightarrow$
$(s{:}nat)$
$(In\ s\ (successors\ n)) \rightarrow$
$(L.ge\ (Fst\ ks).(s)\ (transf\ n\ (Fst\ ks).(n)))$.

We now show that this invariant is satisfied by the start state, and preserved by the *step* function of the iteration. This requires a great many boring and non-trivial lemmas.

Lemma *start_state_good*:
$(good\_state\ start\_state)$.

Lemma *propagate_good_state*:
$(ks\colon state)\ (out{:}L.T)\ (n\colon nat)$
$let\ ks' = (propagate\_succ\ ks\ out\ n)\ in$
$(L.ge\ (Fst\ ks').(n)\ out)$
$\wedge\ ((s\colon nat)\ n \neq s \rightarrow ((Fst\ ks').(s)) = ((Fst\ ks).(s)))$.

Lemma *propagate_list_good_state*:
$(succs\colon (list\ nat))\ (ks\colon state)\ (out{:}L.T)$
$let\ ks' = (propagate\_succ\_list\ ks\ out\ succs)\ in$
$(s{:}nat)$
$((In\ s\ succs) \rightarrow (L.ge\ (Fst\ ks').(s)\ out))$
$\wedge\ (\neg(In\ s\ succs) \rightarrow (Fst\ ks').(s) = (Fst\ ks).(s))$.

Lemma *propagate_increasing*:
(*ks*: *state*) (*out*:*L.T*) (*n*: *nat*)
let ks′ = (*propagate_succ ks out n*) *in*
(*s*:*nat*) (*In s* (*Snd ks*)) → (*In s* (*Snd ks′*)).

Lemma *propagate_list_increasing*:
(*succs*: (*list nat*)) (*ks*: *state*) (*out*:*L.T*)
let ks′ = (*propagate_succ_list ks out succs*) *in*
(*s*:*nat*) (*In s* (*Snd ks*)) → (*In s* (*Snd ks′*)).

Lemma *propagate_records_changes*:
(*ks*: *state*) (*out*:*L.T*) (*n*: *nat*)
let ks′ = (*propagate_succ ks out n*) *in*
(*s*:*nat*) (*In s* (*Snd ks′*)) ∨ (*Fst ks′*).(*s*) = (*Fst ks*).(*s*).

Lemma *propagate_list_records_changes*:
(*succs*: (*list nat*)) (*ks*: *state*) (*out*:*L.T*)
let ks′ = (*propagate_succ_list ks out succs*) *in*
(*s*:*nat*) (*In s* (*Snd ks′*)) ∨ (*Fst ks′*).(*s*) = (*Fst ks*).(*s*).

Lemma *propagate_ge*:
(*ks*: *state*) (*out*:*L.T*) (*n*: *nat*)
let ks′ = (*propagate_succ ks out n*) *in*
(*s*:*nat*) (*L.ge* (*Fst ks′*).(*s*) (*Fst ks*).(*s*)).

Lemma *propagate_list_ge*:
(*succs*: (*list nat*)) (*ks*: *state*) (*out*:*L.T*)
let ks′ = (*propagate_succ_list ks out succs*) *in*
(*s*:*nat*) (*L.ge* (*Fst ks′*).(*s*) (*Fst ks*).(*s*)).

Lemma *step_state_good*:
(*ks*: *state*)
(*stop ks*) = *false* →
(*good_state ks*) →
(*good_state* (*step ks*)).

7.2.3 Correctness theorems.

Using the invariant on intermediate states, we prove that the result of the function *fixpoint* indeed satisfies the dataflow inequations *fixpoint*.(*s*) ≥ (*transf n fixpoint*.(*n*)) for every edge $n \to s$ in the flow graph.

Theorem *fixpoint_solution*:
(*n*: *nat*)(*s*: *nat*)
(*le n last_node*) →
(*In s* (*successors n*)) →
(*L.ge fixpoint*.(*s*) (*transf n fixpoint*.(*n*))).

Moreover, for each (*n*,*v*) in *entry_points*, we have that *fixpoint*.(*n*) ≥ *v*.

Lemma *while_ge_start_state*:
(*ge_state while start_state*).

Lemma *ge_state_ge_in*:
(*s1*,*s2*: *state*)
(*ge_state s1 s2*) →
(*p*:*nat*) (*le p last_node*) → (*L.ge* (*Fst s1*).(*p*) (*Fst s2*).(*p*)).

Lemma *start_state_ge_entry_points*:
(*p*:*nat*)(*l*:*L.T*)
(*In* (*p*,*l*) *entry_points*) → (*L.ge* (*start_state_in entry_points*).(*p*) *l*).

Theorem *fixpoint_entry*:
(*p*:*nat*)(*l*:*L.T*) (*In* (*p*,*l*) *entry_points*) → (*L.ge fixpoint*.(*p*) *l*).

End *Kildall*.

End *Dataflow_Solver*.

7.3 Backward dataflow inequations

The solver developed above solves a forward dataflow problem of the form $X(s) \geq (transf\ n\ X(n))$ for every edge $n \to s$. We now develop a solver for backward dataflow problems, of the form $X(n) \geq (transf\ s\ X(s))$ for every edge $n \to s$. This "backward" solver is derived from the "forward" solver by inverting the flow graph, replacing the successor edges $n \to s$ by predecessor edges $s \to n$.

The export interface for the backward solver is similar to that of the forward solver, with a different *fixpoint_solution* correctness property.

Module Type *BACKWARD_DATAFLOW_SOLVER*.
Declare Module *L*: *SEMILATTICE*.
Parameter *fixpoint*:
(*last_node*: *nat*)
(*successors*: (*nat*→(*list nat*)))
(*transf*: *nat*→*L.T*→*L.T*)
(*entry_points*: (*list nat*×*L.T*))
(*map L.T*).
Parameter *fixpoint_solution*:
(*last_node*:*nat*)
(*successors*: *nat*→(*list nat*))
(*successors_in_graph*: (*n*,*s*:*nat*)(*In s* (*successors n*))→(*le s last_node*))
(*transf*: *nat*→*L.T*→*L.T*)
(*entry_points*: (*list nat*×*L.T*))
(*n*,*s*:*nat*)
(*le n last_node*) →
(*In s* (*successors n*)) →
(*L.ge*
(*fixpoint last_node successors transf entry_points*).(*n*)

```
      (transf s
        (fixpoint last_node successors transf entry_points).(s))).
  Parameter fixpoint_entry:
    (last_node:nat)
    (successors: nat→(list nat))
    (transf: nat→L.T→L.T)
    (entry_points: (list nat×L.T))
    (entry_points_in_graph:
       (n:nat; s:L.T)(In (n,s) entry_points)→(le n last_node))
    (p:nat; l:L.T)
      (In (p,l) entry_points) →
      (L.ge (fixpoint last_node successors transf entry_points).(p) l).
End BACKWARD_DATAFLOW_SOLVER.
```

This interface is implemented by the following functor, which inverts the flow graph and applies the forward solver *Dataflow_solver* to this inverted graph.

```
Module Backward_Dataflow_Solver[LAT: SEMILATTICE]:
                                  BACKWARD_DATAFLOW_SOLVER with Module L := LAT.

Module L := LAT.

Module DS := (Dataflow_Solver L).

Section Kildall_backward.

Variable last_node: nat.
Variable successors: nat → (list nat).
Hypothesis successors_in_graph:
  (n:nat) (s:nat) (In s (successors n)) → (le s last_node).

Variable transf: nat → L.T → L.T.
Variable entry_points: (list (nat × L.T)).
Hypothesis entry_points_in_graph:
  (n:nat) (s:L.T) (In (n,s) entry_points) → (le n last_node).
```

Here we construct the "predecessors" mapping (associating a list of predecessors to every node) from the "successors" mapping.

```
Fixpoint add_successors
       [pred: (map (list nat)); from: nat; tolist: (list nat)]
       : (map (list nat)) :=
  Cases tolist of
    nil ⇒ pred
  | (cons to rem) ⇒
       (add_successors (pred.(to) ← (cons from pred.(to))) from rem)
  end.

Lemma add_successors_correct:
  (tolist: (list nat)) (pred: (map (list nat))) (from: nat)
  let pred' = (add_successors pred from tolist) in
```

```
  (n,s: nat)
    (In n pred'.(s)) ↔
    ((In n pred.(s)) ∨ (n = from ∧ (In s tolist))).

Fixpoint make_predecessors_aux [pred: (map (list nat)); pc: nat] : (map (list nat)) :=
  Cases pc of
    O ⇒ pred
  | (S n) ⇒
      (make_predecessors_aux (add_successors pred n (successors n)) n)
  end.

Lemma make_predecessors_aux_correct:
  (pc:nat) (pred: (map (list nat)))
  let pred' = (make_predecessors_aux pred pc) in
  (n,s: nat)
    (In n pred'.(s)) ↔
    ((In n pred.(s)) ∨ ((lt n pc) ∧ (In s (successors n)))).

Definition predecessors : nat → (list nat) :=
  let pred = (make_predecessors_aux (map_init (list nat) (nil nat)) (S last_node)) in
  [n:nat] pred.(n).
```

The *predecessors* mapping is correct, in the following sense:

```
Lemma predecessors_correct:
  (n,s: nat)
  (le n last_node) →
  (In s (successors n)) ↔ (In n (predecessors s)).
```

Consequently, all predecessors of a node denote valid graph nodes.

```
Lemma predecessors_in_graph:
  (s:nat) (n:nat) (In n (predecessors s)) → (le n last_node).
```

We can now apply the forward dataflow solver to the *predecessors* mapping, obtaining the backward dataflow solver.

```
Definition fixpoint :=
  (DS.fixpoint last_node predecessors predecessors_in_graph transf entry_points).

Theorem fixpoint_solution:
  (n,s:nat)
    (le n last_node) →
    (In s (successors n)) →
    (L.ge
      fixpoint.(n)
      (transf s fixpoint.(s))).

Theorem fixpoint_entry:
  (p:nat; l:L.T)
    (In (p,l) entry_points) →
    (L.ge fixpoint.(p) l).
```

End *Kildall_backward*.

End *Backward_Dataflow_Solver*.

8 Module *Reg*: pseudo-registers

The module *Reg* axiomatizes the notion of pseudo-register used by the instructions of the intermediate code (see module *Instr*).

Require *Map*. Require *Lattice*. Require *Relations*. Require *Misc*.
Require *Lt*. Require *Le*. Require *Peano_dec*.

8.1 Definition of pseudo-registers

Conceptually, pseudo-registers are like identifiers in a high-level calculus: they have a unique identity, and there exists infinitely many different identifiers. However, we quickly run into the following problem: most of the program analyses we need to implement manipulate maps from registers to values from some abstract domain; thus, maps from registers to abstract domains must be equipped with a well-founded ordering, and this is not possible if there are countably many registers. Hence, we need to bound the number of pseudo-registers by some arbitrary constant *max_num_regs*. In practice, this constant can be chosen sufficiently large (e.g. the greatest machine integer) to ensure that the compiler will run out of memory before exhausting the (finite) supply of distinct registers. This maintains the illusion that the compiler can always pick a fresh register, yet ensures that register maps are well-founded.

Parameter *max_num_regs*: *nat*.

Thus, a (pseudo-) register is an integer between 0 (included) and *max_num_regs* (excluded). We represent them as a dependent record of an integer and a proof that this integer is less than *max_num_regs*.

Record *reg* : *Set* := *make_reg* { *reg_no*: *nat*; *reg_lt*: (*lt reg_no max_num_regs*) }.

For some proofs, we need that the type *reg* is inhabited. Hence, we require that *max_num_regs* is not 0, and define register number 0.

Axiom *regs_not_empty*: *max_num_regs* $\neq$ *O*.

Definition *some_reg* :=
  (*make_reg O*
    (*neq_O_lt max_num_regs* (*sym_not_eq nat max_num_regs O regs_not_empty*))).

This is a "proof irrelevance" axiom stating that two registers are equal as soon as they have the same number, even if the proofs that this number is less than *max_num_regs* differ.

Axiom *reg_eq*: (*r1*,*r2*: *reg*) (*reg_no r1*) = (*reg_no r2*) $\rightarrow$ *r1* = *r2*.

Consequently, register equality is decidable.

Lemma *reg_not_eq*: (*r1*,*r2*: *reg*) *r1* $\neq$ *r2* $\rightarrow$ (*reg_no r1*) $\neq$ (*reg_no r2*).

Lemma *reg_eq_dec*: (*r1*,*r2*: *reg*) {*r1* = *r2*} + {*r1* $\neq$ *r2*}.

8.2 Register maps

We now define finite maps from registers to some type. These maps are similar to the integer maps defined in module *Map*, and actually are built on top of them, but have additional properties due to the finiteness of the type *reg*.

Definition *regmap* [*A*: *Set*] := (*map A*).

Section *RegMap*.

Variable *A*: *Set*.

The three basic operations on register maps are similar to those on maps: initialization, lookup, and update.

Definition *regmap_init*: $A \to$ (*regmap A*) :=
  (*map_init A*).
Definition *regmap_get*: *reg* $\to$ (*regmap A*) $\to A$:=
  [*r*:*reg*; *m*:(*regmap A*)] (*map_get A* (*reg_no r*) *m*).
Definition *regmap_set*: *reg* $\to A \to$ (*regmap A*) $\to$ (*regmap A*) :=
  [*r*:*reg*; *v*:*A*; *m*:(*regmap A*)] (*map_set A* (*reg_no r*) *v m*).

Lemma *regmap_get_init*:
  (*a*:*A*)(*r*:*reg*) (*regmap_get r* (*regmap_init a*)) = *a*.

Lemma *regmap_get_set_same*:
  (*m*:(*regmap A*))(*r*:*reg*)(*a*:*A*) (*regmap_get r* (*regmap_set r a m*)) = *a*.

Lemma *regmap_get_set_other*:
  (*m*:(*regmap A*))(*r*,*r'*:*reg*)(*a*:*A*)
  $r \neq r' \to$ (*regmap_get r'* (*regmap_set r a m*)) = (*regmap_get r' m*).

We add an extensionality axiom stating that two register maps are equal if they associate equal values to every register.

Axiom *regmap_extensional*:
  (*m1*,*m2*: (*regmap A*))
    ((*r*:*reg*) (*regmap_get r m1*) = (*regmap_get r m2*)) $\to$
    m1 = *m2*.

Lemma *regmap_set_invariant*:
  (*m*: (*regmap A*)) (*r*: *reg*) (*a*: *A*)
    (*regmap_get r m*) = *a* $\to$ (*regmap_set r a m*) = *m*.

End *RegMap*.

To help manipulate register maps, we introduce the following convenient notations, reminiscent of OCaml's string notations: $m.[i]$ for *regmap_get*, and $m.[i] \leftarrow v$ for *regmap_set*.

Notation "*a* .[*b*] $\leftarrow$ *c*" := (*regmap_set* ? *b c a*).
Notation "*a* .[*b*]" := (*regmap_get* ? *b a*).

8.3 Register maps as a semi-lattice

Given a semi-lattice A, we now equip the type $(regmap\ A.T)$ with the structure of a semi-lattice.

Module *Regmap* [*A*: *SEMILATTICE*].

Definition $T := (regmap\ A.T)$.

Definition $gt := (map\_gt\ A.T\ A.gt\ max\_num\_regs)$.

Lemma wf: $(well\_founded\ T\ gt)$.

Lemma eq_dec: $(p,q{:}\ T)\ \{p{=}q\} + \{p{\neq}q\}$.

Lemma $ge\_pointwise\_1$:
$(p,q{:}\ T)\ (ge\ p\ q) \rightarrow (r{:}reg)\ (A.ge\ p.[r]\ q.[r])$.

Lemma $ge\_pointwise\_2$:
$(p,q{:}\ T)\ ((r{:}reg)\ (A.ge\ p.[r]\ q.[r])) \rightarrow (ge\ p\ q)$.

Definition $top := (regmap\_init\ A.T\ A.top)$.
Definition $bot := (regmap\_init\ A.T\ A.bot)$.

Lemma ge_top_x: $(x{:}T)\ (ge\ top\ x)$.

Lemma ge_x_bot: $(x{:}T)\ (ge\ x\ bot)$.

Fixpoint $lub\_aux\ [p,q{:}\ T; n{:}nat] : T :=$
$Cases\ n\ of$
$\quad O \Rightarrow (map\_init\ A.T\ A.bot)$
$\mid (S\ m) \Rightarrow (map\_set\ A.T\ m\ (A.lub\ p.(m)\ q.(m))\ (lub\_aux\ p\ q\ m))$
end.

Definition $lub\ [p,q{:}\ T]{:}\ T := (lub\_aux\ p\ q\ max\_num\_regs)$.

Remark lub_aux_commut:
$(range{:}nat)\ (p,q{:}\ T)$
$\quad (n{:}nat)\ (lt\ n\ range) \rightarrow (lub\_aux\ p\ q\ range).(n) = (lub\_aux\ q\ p\ range).(n)$.

Lemma lub_commut: $(p,q{:}\ T)\ (lub\ p\ q) = (lub\ q\ p)$.

Remark $ge\_lub\_aux\_left$:
$(range{:}\ nat)\ (p,\ q{:}\ T)$
$\quad (n{:}\ nat)\ (lt\ n\ range) \rightarrow (A.ge\ (lub\_aux\ p\ q\ range).(n)\ p.(n))$.

Lemma ge_lub_left:
$(p,\ q{:}\ T)\ (ge\ (lub\ p\ q)\ p)$.

End *Regmap*.

9 Module *Mach_ops*: semantics of machine operations

In this module, we axiomatize the type of data manipulated by the target processor, along with the basic operations on these data types. Rather than define precisely the semantics of the operations

(e.g. integer arithmetic modulo 2^32), we prefer to leave these operations uninterpreted, and just add the relevant axioms that justify the optimizations performed by the compiler.

Definition *value* := *nat*.

Parameter *bad_value*: *value*.

Parameter *mach_add*: *value* → *value* → *value*.
Parameter *mach_sub*: *value* → *value* → *value*.
Parameter *mach_equal*: *value* → *value* → *bool*.
Parameter *mach_notequal*: *value* → *value* → *bool*.
Parameter *mach_lessthan*: *value* → *value* → *bool*.
Parameter *mach_lessequal*: *value* → *value* → *bool*.
Parameter *mach_greaterthan*: *value* → *value* → *bool*.
Parameter *mach_greaterequal*: *value* → *value* → *bool*.

Axiom *mach_add_commut*: (*x*,*y*:*value*) (*mach_add x y*) = (*mach_add y x*).
Axiom *mach_add_x_O*: (*x*:*value*) (*mach_add x O*) = *x*.
Axiom *mach_add_O_x*: (*x*:*value*) (*mach_add O x*) = *x*.
Axiom *mach_sub_x_O*: (*x*:*value*) (*mach_sub x O*) = *x*.

10 Module *Instr*: the intermediate language

This module defines the intermediate code on which we perform analyzes and transformations. This intermediate code is similar in spirit to a register-based abstract machine, and also to what is called "register transfer language" in compiler literature. Most instructions correspond to single instructions of the target processor, and operate over pseudo-registers. Control-flow is expressed by branches and conditional branches. The code is structured in functions. Each function has its own set of registers, distinct from that of its callers and callees. (In other terms, all registers are preserved at function calls.)

Require *PolyList*. Require *Lt*. Require *Le*. Require *Compare_dec*.
Require *Misc*. Require *Map*. Require *Reg*. Require *Mach_ops*.

10.1 Instruction set, functions, and programs.

Inductive *operation* : *Set* :=
  Omove: *operation*
| *Oconst*: *value* → *operation*
| *Oneg*: *operation*
| *Oadd*: *operation*
| *Osub*: *operation*
| *Oaddimm*: *value* → *operation*
| *Osubimm*: *value* → *operation*.

Inductive *addressing_mode* : *Set* :=
  Aindexed: *value* → *addressing_mode*.

```
Inductive condition : Set :=
    Cequal: condition
  | Cnotequal: condition
  | Clessthan: condition
  | Clessequal: condition
  | Cgreaterthan: condition
  | Cgreaterequal: condition.
```

Instructions of the intermediate code are of one of the following kinds:

- *Inop*: do nothing, continue with next instruction.
- *Iop op args res*: perform the arithmetic operation *op* with arguments the values of the registers *args*; store result in register *res*; continue with next instruction.
- *Iload mode args res*: read the memory at address determined by the addressing mode *mode* and the values of the registers *args*; put result value in register *res*; continue with next instruction.
- *Istore mode args src*: write the value of register *src* to the memory at address determined by the addressing mode *mode* and the values of the registers *args*; continue with next instruction.
- *Icall fn args res*: call the function at address the value of register *fn*, passing it as arguments the values of the registers *args*; store the value returned by this function in register *res*; continue with next instruction.
- *Ibranch dest*: continue with instruction at PC *dest*.
- *Icondbranch cond args dest*: if condition *cond* holds for the values of the registers *args*, continue with instruction at PC *dest*; otherwise, continue with next instruction.
- *Ireturn res*: return to caller of current function, with the value of register *res* as return value.

```
Inductive instruction : Set :=
    Inop: instruction
  | Iop: operation → (list reg) → reg → instruction
  | Iload: addressing_mode → (list reg) → reg → instruction
  | Istore: addressing_mode → (list reg) → reg → instruction
  | Icall: reg → (list reg) → reg → instruction
  | Ibranch: nat → instruction
  | Icondbranch: condition → (list reg) → nat → instruction
  | Ireturn: reg → instruction.
```

A function is composed of a sequence of instructions, numbered consecutively from 0 to some integer *last_pc*. To ensure that program analyses and transformations make sense, we need to impose certain well-formedness conditions on the instructions that compose a function. Currently, the conditions we need are:

- branches and conditional branches stay within the current function, i.e. their target PC is between 0 and *last_pc*;

- the last instruction of a function does not "fall through", i.e. it must be a *Ireturn* or a *Igoto*.

Additional well-formedness conditions can be considered in the future, e.g. that the number of argument registers is consistent with the arity of an operation or an addressing mode.

```
Definition instr_valid [i: instruction; last_pc: nat] :=
  Cases i of
    (Ibranch n) ⇒ (le n last_pc)
  | (Icondbranch cond regs n) ⇒ (le n last_pc)
  | _ ⇒ True
  end.

Definition instr_terminal [i: instruction] :=
  Cases i of
    (Ibranch _) ⇒ True
  | (Ireturn _) ⇒ True
  | _ ⇒ False
  end.

Record function : Set := make_function
  { fn_instr: (map instruction);
    fn_params: (list reg);
    fn_last_pc: nat;
    fn_valid:
      (pc:nat) (le pc fn_last_pc) → (instr_valid fn_instr.(pc) fn_last_pc);
    fn_last_instr_terminal:
      (instr_terminal fn_instr.(fn_last_pc)) }.
```

A program is a collection of functions, identified by an integer between 0 and *prog_num_functions* (excluded), plus a distinguished function that acts as the entry point (the "main function") of the program.

```
Record program : Set := make_program
  { prog_function: (map function);
    prog_num_functions: nat;
    prog_entrypoint: nat;
    prog_entrypoint_valid: (lt prog_entrypoint prog_num_functions) }.
```

10.2 Dynamic semantics

```
Section Dynamic_semantics.

Variable prog: program.
```

The type *location* represents memory locations. The type *regset* represent register sets, i.e. mappings from registers to values. The type *store* represent memory stores, i.e. mappings from locations to values.

```
Definition location := nat.
```

```
Definition regset := (regmap value).
Definition store := (map value).
```

Evaluation of an operation applied to a list of argument values. The result is defined in terms of the machine-level operations axiomatized in module *Mach_ops*.

```
Definition eval_operation [op:operation; args:(list value)] :=
  Cases (op, args) of
    (Omove, (cons x nil)) ⇒ x
  | ((Oconst n), nil) ⇒ n
  | (Oneg, (cons x nil)) ⇒ (mach_sub O x)
  | (Oadd, (cons x (cons y nil))) ⇒ (mach_add x y)
  | (Osub, (cons x (cons y nil))) ⇒ (mach_sub x y)
  | ((Oaddimm n), (cons x nil)) ⇒ (mach_add x n)
  | ((Osubimm n), (cons x nil)) ⇒ (mach_sub x n)
  | _ ⇒ bad_value
  end.
```

Evaluation of an addressing operation. Given an addressing mode and a list of argument values, this returns the memory location to be addressed.

```
Definition eval_addressing [mode:addressing_mode; args:(list value)] :=
  Cases (mode, args) of
    ((Aindexed n), (cons x nil)) ⇒ (mach_add x n)
  | _ ⇒ bad_value
  end.
```

Evaluation of a boolean condition.

```
Parameter bad_bool: bool.

Definition eval_condition [cond:condition; args:(list value)] :=
  Cases (cond, args) of
    (Cequal, (cons x (cons y nil))) ⇒ (mach_equal x y)
  | (Cnotequal, (cons x (cons y nil))) ⇒ (mach_notequal x y)
  | (Clessthan, (cons x (cons y nil))) ⇒ (mach_lessthan x y)
  | (Clessequal, (cons x (cons y nil))) ⇒ (mach_lessequal x y)
  | (Cgreaterthan, (cons x (cons y nil))) ⇒ (mach_greaterthan x y)
  | (Cgreaterequal, (cons x (cons y nil))) ⇒ (mach_greaterequal x y)
  | _ ⇒ bad_bool
  end.
```

Evaluation of a register or list of registers in a given register set.

```
Definition eval_reg [rs: regset; r: reg] := rs.[r].

Definition eval_regs [rs: regset; rl: (list reg)] :=
  (PolyList.map (eval_reg rs) rl).
```

The register set at function entry is obtained by binding the values of the actual arguments to the registers designated as formal parameter registers in the function definition.

```
Fixpoint set_regs [rs: regset; rl: (list reg); vl: (list value)] : regset :=
  Cases (rl, vl) of
    ((cons r rt), (cons v vt)) ⇒ (set_regs (rs.[r] ← v) rt vt)
  | (_, _) ⇒ rs
  end.

Definition function_entry_regs [f: function; vl: (list value)] :=
  (set_regs (regmap_init value bad_value) (fn_params f) vl).
```

The dynamic semantics of instructions is given by a "mostly small-step" reduction predicate: $(exec\_instr\ f\ pc\ rs\ st\ pc'\ rs'\ st')$ holds if the execution of instruction at pc in function f, with initial registers rs and memory store st, leads to the instruction at pc' in function f, with final registers rs' and final store st'. For all instructions except *Icall*, this corresponds to the execution of exactly one instruction. For *Icall*, however, the called function is executed entirely up to its return point. Thus, one *exec_instr* step can correspond to a whole function application.

To deal with *Icall* instructions, another predicate is defined in a mutually-recursive manner: $(exec\_function\ f\ pc\ rs\ st\ retval\ st')$ holds if, starting at instruction number pc in function f with initial registers rs and memory store st, the execution of zero, one or several instructions leads to an *Ireturn* instruction in f, which returns the value *retval*, and the final store is st'.

Unlike a traditional machine-level reduction semantics, this presentation based on two mutually-recursive predicates does not need to manipulate explicitly a return stack holding saved evaluation contexts for calling functions. This makes stating and proving the correctness of intra-function optimizations somewhat easier.

```
Mutual Inductive exec_instr:
        function → nat → regset → store → nat → regset → store → Prop :=
```

Evaluating a *Inop* instruction simply skips to the next instruction, without changing the register set nor the store.

```
  exec_nop:
    (f: function) (pc: nat) (rs: regset) (st: store)
    (fn_instr f).(pc) = Inop →
    (exec_instr f pc rs st (S pc) rs st)
```

For an *Iop* instruction, the argument registers are evaluated w.r.t. the current register set, and the result value is computed using the *eval_operation* function, then stored in the result register. The store is unchanged.

```
| exec_op:
    (f: function) (pc: nat) (rs: regset) (st: store)
    (op: operation) (srcs: (list reg)) (dst: reg)
    (fn_instr f).(pc) = (Iop op srcs dst) →
    (exec_instr f pc rs st
       (S pc) (rs.[dst] ← (eval_operation op (eval_regs rs srcs))) st)
```

The *Iload* operation evaluates the location to be read using the *eval_addressing* function. It then reads the corresponding store location, and store its value in the result register. The store is unchanged.

```
| exec_load:
```

$(f\colon function)\ (pc\colon nat)\ (rs\colon regset)\ (st\colon store)$
$(mode\colon addressing\_mode)\ (addrs\colon (list\ reg))\ (dst\colon reg)$
$(fn\_instr\ f).(pc) = (Iload\ mode\ addrs\ dst) \rightarrow$
$(exec\_instr\ f\ pc\ rs\ st$
$\quad (S\ pc)\ (rs.[dst] \leftarrow (st.(eval\_addressing\ mode\ (eval\_regs\ rs\ addrs))))\ st)$

The *Istore* operation evaluates the location to be modified using the *eval_addressing* function. It then updates this store location with the value of the *src* register. The register set is unchanged.

| *exec_store*:
$(f\colon function)\ (pc\colon nat)\ (rs\colon regset)\ (st\colon store)$
$(mode\colon addressing\_mode)\ (addrs\colon (list\ reg))\ (src\colon reg)$
$(fn\_instr\ f).(pc) = (Istore\ mode\ addrs\ src) \rightarrow$
$(exec\_instr\ f\ pc\ rs\ st$
$\quad (S\ pc)\ rs\ (st.(eval\_addressing\ mode\ (eval\_regs\ rs\ addrs)) \leftarrow (eval\_reg\ rs\ src)))$

The *Icall* instruction resolves the value of the *fn* register to the description f' of the called function. It then evaluates f' from the instruction at PC 0 to the first *Ireturn* instruction, using *eval_function*. The parameter registers of f' are initialized from the values of the argument registers *args*. The value *retval* returned by f' is stored in register *res*, and the final store is that at the end of f'.

| *exec_call*:
$(f\colon function)\ (pc\colon nat)\ (rs\colon regset)\ (st\colon store)$
$(fn\colon reg)\ (args\colon (list\ reg))\ (res\colon reg)$
$(retval\colon value)\ (st'\!:\!store)$
$(fn\_instr\ f).(pc) = (Icall\ fn\ args\ res) \rightarrow$
$let\ name = (eval\_reg\ rs\ fn)\ in$
$(lt\ name\ (prog\_num\_functions\ prog)) \rightarrow$
$let\ f' = (prog\_function\ prog).(name)\ in$
$(exec\_function\ f'\ O\ (function\_entry\_regs\ f'\ (eval\_regs\ rs\ args))\ st$
$\qquad retval\ st') \rightarrow$
$(exec\_instr\ f\ pc\ rs\ st$
$\qquad (S\ pc)\ (rs.[res] \leftarrow retval)\ st')$

The *Ibranch* instruction sets the *pc* component to the destination of the branch. The register set and the store are unaffected.

| *exec_branch*:
$(f\colon function)\ (pc\colon nat)\ (rs\colon regset)\ (st\colon store)$
$(dest\colon nat)$
$(fn\_instr\ f).(pc) = (Ibranch\ dest) \rightarrow$
$(exec\_instr\ f\ pc\ rs\ st$
$\qquad dest\ rs\ st)$

The *Icondbranch* instruction evaluates its boolean condition. If it evaluates to *true*, *pc* is set of the destination of the branch. If it evaluates to *false*, *pc* advances to the next instruction. The register set and the store are unaffected.

| *exec_condbranch_taken*:
$(f\colon function)\ (pc\colon nat)\ (rs\colon regset)\ (st\colon store)$
$(c\colon condition)\ (args\colon (list\ reg))\ (dest\colon nat)$
$(fn\_instr\ f).(pc) = (Icondbranch\ c\ args\ dest) \rightarrow$

```
    (eval_condition c (eval_regs rs args)) = true →
    (exec_instr f pc rs st dest rs st)
| exec_condbranch_nottaken:
    (f: function) (pc: nat) (rs: regset) (st: store)
    (c: condition) (args: (list reg)) (dest: nat)
    (fn_instr f).(pc) = (Icondbranch c args dest) →
    (eval_condition c (eval_regs rs args)) = false →
    (exec_instr f pc rs st (S pc) rs st)

with exec_function:
     function → nat → regset → store → value → store → Prop :=
```

The *exec_step* rules corresponds to the evaluation of a non-*Ireturn* instruction at *pc*.

```
  exec_step:
    (f: function) (pc: nat) (rs: regset) (st: store)
    (pc':nat) (rs': regset) (st': store)
    (res: value) (st'': store)
    (exec_instr f pc rs st pc' rs' st') →
    (exec_function f pc' rs' st' res st'') →
    (exec_function f pc rs st res st'')
```

The *exec_return* rules corresponds to the evaluation of an *Ireturn* instruction at *pc*. The result value is the current value of the argument register.

```
| exec_return:
    (f: function) (pc: nat) (rs: regset) (st: store)
    (arg: reg)
    (fn_instr f).(pc) = (Ireturn arg) →
    (exec_function f pc rs st (eval_reg rs arg) st).
```

Proofs on the dynamic semantics are carried out by mutual induction over the *exec_instr* and *exec_function* inductive predicates, using the following proof principles. We will mostly use *exec_function_scheme* in the following.

```
Scheme exec_instr_scheme := Minimality for exec_instr Sort Prop
  with exec_function_scheme := Minimality for exec_function Sort Prop.
```

The evaluation of a program corresponds to executing the entry function of the program in the given store, until this function returns. The result of the program execution is the return value of the entry function, plus the final memory store.

```
Inductive exec_program: store → value → store → Prop :=
  exec_program_intro:
    (st: store) (res: value) (st': store)
    (exec_function (prog_function prog).((prog_entrypoint prog))
                   O (regmap_init value bad_value) st res st') →
    (exec_program st res st').

End Dynamic_semantics.
```

10.3 Successor instructions

In this section, we define the set of successors of an instruction in a function: *successors f pc* is the list of all PCs that can be reached by executing the instruction at *pc* in function *f*. Thus, *successors f* defines the edges of the control-flow graph for function *f*, as required for data flow analysis (see module *Kildall*).

```
Section Successors.

Variable f: function.

Definition successors [pc:nat] :=
  Cases (le_lt_dec pc (fn_last_pc f)) of
    (left _) ⇒
      Cases (fn_instr f).(pc) of
        (Ibranch dst) ⇒ (cons dst (nil nat))
      | (Icondbranch cond args dst) ⇒ (cons dst (cons (S pc) (nil nat)))
      | (Ireturn arg) ⇒ (nil nat)
      | _ ⇒ (cons (S pc) (nil nat))
      end
  | (right _) ⇒
      (nil nat)
  end.
```

Owing to the well-formedness conditions on function code, the successors of an instruction are always within the range of valid PCs for the function, i.e. between 0 and *fn_last_pc f*.

```
Lemma successors_in_graph:
  (n,s:nat) (In s (successors n)) → (le s (fn_last_pc f)).

End Successors.
```

The *successors* function is correct in the following sense: if the execution of an instruction at position *pc* takes us to position *pc′*, then *pc′* belongs to *successors f pc*.

```
Lemma successors_correct:
  (p: program)(f: function)
  (pc: nat)(rs: regset)(st: store)(pc': nat)(rs':regset)(st':store)
  (le pc (fn_last_pc f)) →
  (exec_instr p f pc rs st pc' rs' st') →
  (In pc' (successors f pc)).
```

As a corollary, the execution of an instruction always take us to a valid PC.

```
Lemma exec_instr_in_code:
  (p: program)(f: function)
  (pc: nat)(rs: regset)(st: store)(pc': nat)(rs':regset)(st':store)
  (le pc (fn_last_pc f)) →
  (exec_instr p f pc rs st pc' rs' st') →
  (le pc' (fn_last_pc f)).
```

10.4 Transformation of a program instruction by instruction.

This section defines higher-order functions to transform a function into a function and a program into a program by extending a given instruction-to-instruction mapping.

Section *Transform_function*.

The transformation function *transf* maps a PC and an original instruction to a transformed instruction. It must preserve validity and terminality of instructions.

Variable *transf*: *nat* → *instruction* → *instruction*.
Hypothesis *transf_preserves_validity*:
(*last_pc*: *nat*) (*pc*: *nat*) (*i*: *instruction*)
(*instr_valid i last_pc*) → (*instr_valid* (*transf pc i*) *last_pc*).
Hypothesis *transf_preserves_terminal*:
(*pc*: *nat*) (*i*: *instruction*)
(*instr_terminal i*) → (*instr_terminal* (*transf pc i*)).

Variable *f*: *function*.

Local *newcode* :=
(*map_apply instruction instruction transf* (*fn_instr f*) (*S* (*fn_last_pc f*))).

Lemma *transf_is_valid*:
(*pc*:*nat*) (*le pc* (*fn_last_pc f*)) → (*instr_valid newcode*.(*pc*) (*fn_last_pc f*)).

Lemma *transf_last_instr_terminal*:
(*instr_terminal newcode*.(*fn_last_pc f*)).

The transformed function is obtained by applying *transf* to each instruction of the original function.

Definition *transf_function* :=
(*make_function*
newcode
(*fn_params f*)
(*fn_last_pc f*)
transf_is_valid
transf_last_instr_terminal).

The following lemma characterizes the instructions of the transformed function.

Lemma *transformed_instr*:
(*pc*: *nat*)
(*le pc* (*fn_last_pc f*)) →
(*fn_instr transf_function*).(*pc*) = (*transf pc* (*fn_instr f*).(*pc*)).

End *Transform_function*.

We now turn to transforming a program function by function.

Section *Transform_program*.

Variable *transf_function*: *nat* → *function* → *function*.

Definition *transf_program* [*p*:*program*] :=

```
(make_program
  (map_apply function function
      transf_function (prog_function p) (prog_num_functions p))
  (prog_num_functions p)
  (prog_entrypoint p)
  (prog_entrypoint_valid p)).
```

The following lemma characterizes the instructions of the transformed program.

```
Lemma transformed_function:
  (p: program) (nm: nat)
  (lt nm (prog_num_functions p)) →
  (prog_function (transf_program p)).(nm) =
  (transf_function nm (prog_function p).(nm)).

End Transform_program.
```

11 Module *Constprop*: constant propagation

The module *Constprop* formalizes constant propagation.

```
Require PolyList. Require Peano_dec.
Require Misc. Require Map. Require Lattice. Require Mach_ops.
Require Instr. Require Reg. Require Kildall.
```

11.1 The static analysis

We first define the forward dataflow analysis that corresponds to constant propagation. To each program point, the analysis associates a mapping from register to abstract values. These abstract values range over the flat lattice of values.

```
Module Value.
  Definition T := value.
  Definition eq_dec := eq_nat_dec.
End Value.

Module A := (Flat Value).
```

The meaning of the abstract value for a given register is as follows:

- *A.Top*: the value of this register is statically unknown (all values)
- *A.Inj v*: this register is known to have value v (one value)
- *A.Bot*: this program point is unreachable (no value).

```
Module D := (Regmap A).
```

Given a list of registers and an abstract register set *rs*, *abstr_regs* determines whether all registers have known values according to *rs*. If so, it returns the list of the values of the registers. If not, it returns *None*.

```
Fixpoint abstr_regs [rs: D.T; args: (list reg)] : (option (list value)) :=
  Cases args of
    nil ⇒ (Some (list value) (nil value))
  | (cons hd tl) ⇒
      Cases rs.[hd] of
        A.Bot ⇒ (None (list value))
      | A.Top ⇒ (None (list value))
      | (A.Inj v) ⇒
          Cases (abstr_regs rs tl) of
            None ⇒ (None (list value))
          | (Some vl) ⇒ (Some (list value) (cons v vl))
          end
      end
  end.
```

The transfer function for the dataflow analysis. Given an abstract register set *rs* representing what is known about the concrete registers before the execution of an instruction, *transfer* returns the abstract register set representing the concrete registers after the execution of the instruction. There are three cases to consider:

- If the instruction is an operation (*Iop op args res*) and the values of the registers *args* are statically known according to *rs*, compute the result *v* of the operation *op* applied to the values of *args*, and set the abstract value of register *res* to *A.Inj v*.
- If the instruction is an operation with unknown arguments, or a load, or a call, set the result register to *A.Top* to reflect the fact that the result value is statically unpredictable.
- Otherwise, leave the abstract register set unchanged.

```
Definition transfer [f:function; pc:nat; rs: D.T] :=
  Cases (fn_instr f).(pc) of
    Inop ⇒
      rs
  | (Iop op args res) ⇒
      Cases (abstr_regs rs args) of
        (Some vl) ⇒ rs.[res] ← (A.inj (eval_operation op vl))
      | None ⇒ rs.[res] ← A.top
      end
  | (Iload mode args res) ⇒
      rs.[res] ← A.top
  | (Istore mode args src) ⇒
      rs
  | (Icall fn args res) ⇒
```

```
      rs.[res] ← A.top
  | (Ibranch dst) ⇒
      rs
  | (Icondbranch cond args dst) ⇒
      rs
  | (Ireturn arg) ⇒
      rs
  end.
```

The dataflow analysis is then obtained by instantiating the general framework for forward dataflow analysis provided by module *Kildall*. Notice that the abstract state at the entry point of the function (PC 0) is set to *D.top* to reflect the fact that nothing is known about the values of the function parameters.

```
Module Solver := (Dataflow_Solver D).

Definition analyze_function [f: function] :=
  (Solver.fixpoint
    (fn_last_pc f)
    (successors f)
    (successors_in_graph f)
    (transfer f)
    (cons (O, D.top) Nil)).
```

11.2 Semantic correctness of the analysis

We now show that the analysis is semantically correct: the abstract values it predicts statically are correct approximations of the concrete values computed at run-time. We formalize this notion of agreement between abstract and concrete values, and extend it to abstract and concrete register sets.

```
Definition value_match_approx [a: A.T; v: value] :=
  Cases a of
    A.Top ⇒ True
  | (A.Inj v') ⇒ v = v'
  | A.Bot ⇒ False
  end.

Definition regset_match_approx [a: D.T; rs: regset] :=
  (r:reg) (value_match_approx a.[r] rs.[r]).
```

Some easy properties of these "match approximation" relations follow.

```
Lemma value_match_approx_increasing:
  (a,b: A.T)(v: value)
  (A.ge a b) → (value_match_approx b v) → (value_match_approx a v).

Lemma regset_match_approx_increasing:
  (a,b: D.T)(rs: regset)
```

$(D.ge\ a\ b) \rightarrow (regset\_match\_approx\ b\ rs) \rightarrow (regset\_match\_approx\ a\ rs)$.

Lemma *regset_match_approx_update*:
$(ra\colon D.T)\ (rs\colon regset)(a\colon A.T)\ (v\colon value)\ (r\colon reg)$
$(value\_match\_approx\ a\ v) \rightarrow$
$(regset\_match\_approx\ ra\ rs) \rightarrow$
$(regset\_match\_approx\ (ra.[r] \leftarrow a)\ (rs.[r] \leftarrow v))$.

Lemma *abstr_regs_match_approx*:
$(ra\colon D.T)\ (rs\colon regset)$
$(regset\_match\_approx\ ra\ rs) \rightarrow$
$(args\colon (list\ reg))\ (vl\colon (list\ value))$
$(abstr\_regs\ ra\ args) = (Some\ (list\ value)\ vl) \rightarrow$
$(eval\_regs\ rs\ args) = vl$.

We then show the semantic correctness of the transfer function: if one execution step takes us from the state (pc, rs, st) to the state (pc', rs', st'), and rs (the concrete register set "before") matches the abstract register set a, then rs' (the concrete register set "after") matches the abstract register set $(transfer\ f\ pc\ a)$.

Lemma *transfer_correct*:
$(p\colon program)(f\colon function)$
$(pc\colon nat)(rs\colon regset)(st\colon store)(pc'\colon nat)(rs'\colon regset)(st'\colon store)$
$(exec\_instr\ p\ f\ pc\ rs\ st\ pc'\ rs'\ st') \rightarrow$
$(a\colon D.T)$
$(regset\_match\_approx\ a\ rs) \rightarrow$
$(regset\_match\_approx\ (transfer\ f\ pc\ a)\ rs')$.

The semantic correctness of the static analysis follows from the correctness of the transfer function and the fact that the result of the analysis is a solution to the dataflow equations.

Lemma *analysis_correct*:
$(p\colon program)(f\colon function)$
$(pc\colon nat)(rs\colon regset)(st\colon store)(pc'\colon nat)(rs'\colon regset)(st'\colon store)$
$(le\ pc\ (fn\_last\_pc\ f)) \rightarrow$
$(exec\_instr\ p\ f\ pc\ rs\ st\ pc'\ rs'\ st') \rightarrow$
$(regset\_match\_approx\ (analyze\_function\ f).(pc)\ rs) \rightarrow$
$(regset\_match\_approx\ (analyze\_function\ f).(pc')\ rs')$.

Lemma *analysis_entry_point*:
$(f\colon function)\ (rs\colon regset)$
$(regset\_match\_approx\ (analyze\_function\ f).(O)\ rs)$.

11.3 Code transformations

We now define the program transformation that actually performs constant propagation, based on the results of the static analysis. Namely:

- Arithmetic operations whose arguments are fully statically known are turned into "set register to constant" operations, where the constant is the result of the operation as computed at compile-time.
- Arithmetic operations whose arguments are partially statically known are replaced by simpler operations (strength reduction). For instance, $r := add(r1,r2)$ is turned into $r := r1$ if $r2$ is known to be 0.
- Conditional branches whose arguments are fully statically known are turned into unconditional branches or no-ops, depending on the value of the condition.

```
Definition transf_op
      [ra: D.T; op: operation; args: (list reg)] :=
  Cases (abstr_regs ra args) of
    (Some vl) ⇒
      ((Oconst (eval_operation op vl)), Nil)
  | None ⇒
      Cases (op, args) of
        (Oadd, (cons r1 (cons r2 nil))) ⇒
          Cases ra.[r1] of
            (A.inj O) ⇒ (Omove, (cons r2 Nil))
          | (A.inj v) ⇒ ((Oaddimm v), (cons r2 Nil))
          | _ ⇒
            Cases ra.[r2] of
              (A.inj O) ⇒ (Omove, (cons r1 Nil))
            | (A.inj v) ⇒ ((Oaddimm v), (cons r1 Nil))
            | _ ⇒ (op, args)
            end
          end
      | (Osub, (cons r1 (cons r2 nil))) ⇒
          Cases ra.[r2] of
            (A.inj O) ⇒ (Omove, (cons r1 Nil))
          | (A.inj v) ⇒ ((Osubimm v), (cons r1 Nil))
          | _ ⇒ (op, args)
          end
      | _ ⇒
          (op, args)
      end
  end.

Definition transf_instr [ra: (map D.T); n: nat; i: instruction] :=
  Cases i of
    (Iop op args res) ⇒
      let oa = (transf_op ra.(n) op args) in (Iop (Fst oa) (Snd oa) res)
  | (Icondbranch cond args dst) ⇒
      Cases (abstr_regs ra.(n) args) of
```

```
        (Some vl) ⇒
          if (eval_condition cond vl)
          then (Ibranch dst)
          else Inop
      | None ⇒ i
      end
  | _ ⇒ i
  end.

Lemma transf_instr_valid:
  (ra: (map D.T)) (last_pc: nat) (n: nat)(i: instruction)
  (instr_valid i last_pc) →
  (instr_valid (transf_instr ra n i) last_pc).

Lemma transf_instr_terminal:
  (ra: (map D.T)) (n: nat)(i: instruction)
  (instr_terminal i) →
  (instr_terminal (transf_instr ra n i)).

Definition transf_function [n:nat; f:function] :=
  let ra = (analyze_function f) in
  (Instr.transf_function
    (transf_instr ra)
    (transf_instr_valid ra)
    (transf_instr_terminal ra)
    f).

Definition transf_program [p: program] :=
  (Instr.transf_program transf_function p).
```

11.4 Transformation preserves semantics

As a first step towards proving that the transformed program behaves like the original program, we show that a transformed arithmetic operation computes the same result as the original arithmetic operation, provided that the concrete registers in which the evaluation takes place match the abstract registers used for the transformation. The proof is a lengthy, but easy case analysis.

```
Lemma transf_op_correct:
  (approx: D.T) (op: operation) (args: (list reg)) (rs: regset)
  (regset_match_approx approx rs) →
  (eval_operation (Fst (transf_op approx op args))
                  (eval_regs rs (Snd (transf_op approx op args)))) =
  (eval_operation op (eval_regs rs args)).
```

Two trivial lemmas about transformed functions and the transformed program.

```
Lemma transformed_instr:
  (f: function) (nm: nat) (pc: nat)
  (le pc (fn_last_pc f)) →
```

(*fn_instr* (*transf_function nm f*)).(*pc*) =
(*transf_instr* (*analyze_function f*) *pc* (*fn_instr f*).(*pc*)).

Lemma *transformed_function*:
(*p*: *program*) (*nm*: *nat*)
(*lt nm* (*prog_num_functions p*)) →
(*prog_function* (*transf_program p*)).(*nm*) =
(*transf_function nm* (*prog_function p*).(*nm*)).

The main semantic equivalence result follows:

Lemma *transf_function_correct*:
(*p*: *program*)(*f*: *function*)
(*pc*: *nat*) (*rs*: *regset*) (*st*: *store*) (*res*: *value*) (*st'*: *store*)
(*exec_function p f pc rs st res st'*) →
(*nm*: *nat*)
(*le pc* (*fn_last_pc f*)) →
(*regset_match_approx* (*analyze_function f*).(*pc*) *rs*) →
(*exec_function* (*transf_program p*) (*transf_function nm f*) *pc rs st res st'*).

This result is shown by simultaneous induction on the derivation of the *exec_function* predicate, and on the derivation of the *exec_instr* predicate, using the following proposition as the induction hypothesis for *exec_instr*:

Check
(*p*: *program*)(*f*: *function*)
(*pc*: *nat*) (*rs*: *regset*) (*st*: *store*)
(*pc'*: *nat*) (*rs'*: *regset*) (*st'*: *store*)
(*exec_instr p f pc rs st pc' rs' st'*) →
(*nm*: *nat*)
(*le pc* (*fn_last_pc f*)) →
(*regset_match_approx* (*analyze_function f*).(*pc*) *rs*) →
(*exec_instr* (*transf_program p*) (*transf_function nm f*) *pc rs st pc' rs' st'*).

The proposition above shows that the execution of the transformed program simulates that of the original program: if the original program performs one execution step from state (*pc*, *rs*, *st*) to state (*pc'*, *rs'*, *st'*), the transformed program performs the same execution step, provided the register set "before" (*rs*) matches the results of the static analysis at point *pc*. The latter hypothesis holds at every step, because it holds at the entrance to each function (lemma *analysis_entry_point*) and is preserved by every execution step (lemma *analysis_correct*).

As a corollary, it follows that the transformed program produces the same results (final store plus return value) as the original program.

Lemma *transf_program_correct*:
(*p*: *program*) (*st*: *store*) (*res*: *value*) (*st'*: *store*)
(*exec_program p st res st'*) →
(*exec_program* (*transf_program p*) *st res st'*).

12 Module *Coloring*: graph coloring

This module axiomatizes interference graphs and their coloring.

Require *Misc*. Require *Reg*.

Interference graphs are undirected graphs with registers as nodes. They are built from the empty graph by successive addition of edges.

Parameter *graph*: *Set*.

Parameter *empty_graph*: *graph*.

Parameter *add_edge*: *reg* → *reg* → *graph* → *graph*.

Parameter *has_edge*: *reg* → *reg* → *graph* → *bool*.

Interference graphs are not directed, hence the *has_edge* function is commutative.

Axiom *has_edge_commut*:
(*r1*,*r2*: *reg*) (*g*: *graph*)
(*has_edge r1 r2 g*) = (*has_edge r2 r1 g*).

Axiom *has_edge_add_same*:
(*r1*,*r2*: *reg*) (*g*: *graph*)
(*has_edge r1 r2* (*add_edge r1 r2 g*)) = *true*.

Axiom *has_edge_add_other*:
(*r1*,*r2*, *r3*, *r4*: *reg*) (*g*: *graph*)
(*has_edge r1 r2 g*) = *true* →
(*has_edge r1 r2* (*add_edge r3 r4 g*)) = *true*.

The graph coloring function takes an interference graph and returns a mapping from registers (the nodes of the graph) to registers (the color assigned to the node).

Parameter *graph_coloring*: *graph* → (*reg* → *reg*).

A correct graph coloring associates different colors to adjacent nodes.

Axiom *graph_coloring_correct*:
(*g*: *graph*) (*r1*, *r2*: *reg*)
(*has_edge r1 r2 g*) = *true* →
(*graph_coloring g r1*) ≠ (*graph_coloring g r2*).

13 Module *Allocation*: register allocation.

The module *Allocation* formalizes the following aspects of register allocation:

- liveness analysis;
- construction of the interference graph;
- rewriting of the code according to a register assignment (obtained by graph coloring of the interference graph);

- elimination of pure instructions that compute dead results, and of useless "move" instructions.

The following aspects are not formalized:

- the graph coloring itself;
- insertion of spilling and reloading code;
- active coalescing of move instructions.

```
Require PolyList. Require Le. Require Lt.
Require Misc. Require Map. Require Lattice. Require Mach_ops.
Require Instr. Require Reg. Require Kildall. Require Coloring.
```

13.1 Liveness analysis

A register r is live at a point p if there exists a path from p to some instruction that uses r as argument, and r is not redefined along this path.

Liveness can be computed by a backward dataflow analysis. The analysis operates over mappings from registers to booleans: *Boolean.Top* denotes a live register, and *Boolean.Bot* denotes a dead register.

```
Module D := (Regmap Boolean).

Fixpoint set_live [d: D.T; regs: (list reg)] : D.T :=
  Cases regs of
    nil ⇒ d
  | (cons r rs) ⇒ (set_live d rs).[r] ← Boolean.Top
  end.
```

Here is the transfer function for the dataflow analysis. Since this is a backward dataflow analysis, it takes as argument the abstract register set "after" the given instruction, i.e. the registers that are live after; and it returns as result the abstract register set "before" the given instruction, i.e. the registers that must be live before. The general relation between "live before" and "live after" an instruction is that a register is live before if either it is one of the arguments of the instruction, or it is not the result of the instruction and it is live after. However, if the result of a side-effect-free instruction is not live "after", the whole instruction will be removed later (since it computes a useless result), thus its arguments need not be live "before".

```
Definition transfer [f: function; pc: nat; after: D.T] :=
  Cases (fn_instr f).(pc) of
    Inop ⇒
      after
  | (Iop op args res) ⇒
      Cases after.[res] of
        Boolean.Bot ⇒
          after
```

```
      | Boolean.Top ⇒
          (set_live (after.[res] ← Boolean.Bot) args)
      end
  | (Iload mode args res) ⇒
      Cases after.[res] of
        Boolean.Bot ⇒
          after
      | Boolean.Top ⇒
          (set_live (after.[res] ← Boolean.Bot) args)
      end
  | (Istore mode args src) ⇒
      (set_live (after.[src] ← Boolean.Top) args)
  | (Icall fn args res) ⇒
      (set_live ((after.[res] ← Boolean.Bot).[fn] ← Boolean.Top) args)
  | (Ibranch dst) ⇒
      after
  | (Icondbranch cond args dst) ⇒
      (set_live after args)
  | (Ireturn arg) ⇒
      after.[arg] ← Boolean.Top
  end.
```

The liveness analysis is then obtained by instantiating the general framework for backward dataflow analysis provided by module *Kildall*.

```
Module Solver := (Backward_Dataflow_Solver D).

Definition analyze_function [f: function] : (map D.T) :=
  (Solver.fixpoint
    (fn_last_pc f)
    (successors f)
    (transfer f)
    Nil).
```

13.2 Construction of the interference graph

Two registers interfere if there exists a program point where they are both simultaneously live, and it is possible that they contain different values at this program point. Consequently, two registers that do not interfere can be merged into one register while preserving the program behavior: there is no program point where this merged register would have to hold two different values (for the two original registers), so to speak.

The interference graph is an undirected graph with registers as nodes. There is an edge between two registers if and only if they interfere.

The algorithm for constructing the interference graph from the results of the liveness analysis is as follows:

```
start with empty interference graph
for each parameter p and register r live at the function entry point:
    add edge p <-> r
for each instruction I in function:
    let L be the live registers "after" I
    if I is a "move" instruction  dst <- src, and dst is live:
       add edge dst <-> r for each r in L \ {dst, src}
    else if I is an instruction with result dst, and dst is live:
       add edge dst <-> r for each r in L \ {dst}
done
```

Notice that edges are added only when a register becomes live. A register becomes live either if it is the result of an operation (and is live afterwards), or if we are at the function entrance and the register is a function parameter. For two registers to be simultaneously live at some program point, it must be the case that one becomes live at a point where the other is already live. Hence, it suffices to add interference edges between registers that become live at some instruction and registers that are already live at this instruction.

Notice also the special treatment of "move" instructions: since the destination register of the "move" is assigned the same value as the source register, it is semantically correct to assign the destination and the source registers to the same register, even if the source register remains live afterwards. (This is even desirable, since the "move" instruction can then be eliminated.) Thus, no interference is added between the source and the destination of a "move" instruction.

We start with some auxiliary functions for recognizing well-formed "move" operations, i.e. *Iop* instruction whose operation is *Omove* and whose argument list contains exactly one register.

Definition *is_move_operation* [*op*: *operation*; *args*: (*list reg*)] :=
Cases (*op*, *args*) *of*
(*Omove*, (*cons src nil*)) ⇒ (*Some reg src*)
| _ ⇒ (*None reg*)
end.

Lemma *is_move_operation_correct*:
(*op*: *operation*) (*args*: (*list reg*)) (*r*: *reg*)
(*is_move_operation op args*) = (*Some reg r*) →
op = *Omove* ∧ *args* = (*cons r Nil*).

Lemma *eval_move_operation*:
(*op*: *operation*) (*args*: (*list reg*)) (*arg*: *reg*) (*rs*: *regset*)
(*is_move_operation op args*) = (*Some reg arg*) →
(*eval_operation op* (*eval_regs rs args*)) = *rs*.[*arg*].

Lemma *is_move_operation_rewritten*:
(*assign*: *reg*→*reg*) (*op*: *operation*) (*args*: (*list reg*)) (*r*: *reg*)
(*is_move_operation op args*) = (*Some reg r*) →
(*is_move_operation op* (*PolyList.map assign args*)) = (*Some reg* (*assign r*)).

We have an algorithmic efficiency issue here. To add interference edges, we need the ability to enumerate efficiently all registers live at a given program point. However, the result of the liveness analysis, at a program point, is a mapping from registers to booleans ("live" or "dead"). From this mapping, there is no efficient way to enumerate the registers that are mapped to "live". (One could enumerate all registers and filter them through the mapping, but there are too many registers for this to be efficient.)

The best solution to this problem would be to use sets of registers (instead of mappings from registers to booleans) as the result of the liveness analysis. But I haven't had the energy to axiomatize sets of registers.

For the time being, we just assume given a function from boolean register mappings to lists of registers, that extracts the registers that are mapped to "live".

```
Parameter list_live_regs: D.T → (list reg).

Axiom list_live_regs_correct:
  (live: D.T) (r: reg) live.[r] = Boolean.Top → (In r (list_live_regs live)).
```

We now define the construction of the interference graph.

```
Fixpoint add_interf_live
  [filter: reg → bool; res: reg; interf: graph; lregs: (list reg)] : graph :=
  Cases lregs of
    nil ⇒ interf
  | (cons r lregs') ⇒
      (add_interf_live filter res
        (if (filter r) then (add_edge r res interf) else interf)
        lregs')
  end.

Definition filter_add_interf_op [res: reg; r: reg] :=
  Cases (reg_eq_dec r res) of (left _) ⇒ false | (right _) ⇒ true end.

Definition add_interf_op [interf: graph; res: reg; live: D.T] :=
  (add_interf_live
    (filter_add_interf_op res)
    res interf (list_live_regs live)).

Definition filter_add_interf_move [arg: reg; res: reg; r: reg] :=
  Cases (reg_eq_dec r res) of
    (left _) ⇒ false
  | (right _) ⇒
      Cases (reg_eq_dec r arg) of
        (left _) ⇒ false
      | (right _) ⇒ true
      end
  end.

Definition add_interf_move[interf: graph; arg: reg; res: reg; live: D.T] :=
```

```
  (add_interf_live
    (filter_add_interf_move arg res)
    res interf (list_live_regs live)).

Definition add_interf_instr
          [f: function; live: (map D.T); interf: graph; pc: nat] :=
  Cases (fn_instr f).(pc) of
    Inop ⇒ interf
  | (Iop op args res) ⇒
      Cases live.(pc).[res] of
        Boolean.Bot ⇒ interf
      | Boolean.Top ⇒
          Cases (is_move_operation op args) of
            (Some arg) ⇒ (add_interf_move interf arg res live.(pc))
          | None ⇒ (add_interf_op interf res live.(pc))
          end
      end
  | (Iload mode args dst) ⇒
      Cases live.(pc).[dst] of
        Boolean.Bot ⇒ interf
      | Boolean.Top ⇒ (add_interf_op interf dst live.(pc))
      end
  | (Istore mode args src) ⇒ interf
  | (Icall nm args res) ⇒ (add_interf_op interf res live.(pc))
  | (Ibranch target) ⇒ interf
  | (Icondbranch cond args target) ⇒ interf
  | (Ireturn arg) ⇒ interf
  end.

Fixpoint add_interf_instrs
          [f: function; live: (map D.T); interf: graph; pc: nat] : graph :=
  Cases pc of
    O ⇒ interf
  | (S p) ⇒ (add_interf_instrs f live (add_interf_instr f live interf p) p)
  end.

Fixpoint add_interf_list [interf: graph; live: D.T; rl: (list reg)] : graph :=
  Cases rl of
    nil ⇒ interf
  | (cons r rs) ⇒ (add_interf_list (add_interf_op interf r live) live rs)
  end.

Definition interference_graph [f: function; live: (map D.T)] :=
  (add_interf_instrs f live
    (add_interf_list empty_graph (transfer f O live.(O)) (fn_params f))
    (S (fn_last_pc f))).
```

13.3 Correctness of the interference graph

We now show that the interference graph is correct with respect to the results of the liveness analysis: the interference graph contains all the edges that it should contain.

Many boring lemmas on the auxiliary functions used to construct the interference graph follow. The lemmas are of two kinds: the "increasing" lemmas show that the auxiliary functions only add edges to the interference graph, but do not remove existing edges; and the "correct" lemmas show that the auxiliary functions correctly add the edges that we'd like them to add.

Lemma *add_interf_live_correct*:
(*filter*: *reg* → *bool*) (*res*: *reg*) (*lregs*: (*list reg*)) (*interf*: *graph*)
let interf' = (*add_interf_live filter res interf lregs*) *in*
((*r*: *reg*)
(*In r lregs*) → (*filter r*) = *true* →
(*has_edge r res interf'*) = *true*)
∧ ((*r1*, *r2*: *reg*)
(*has_edge r1 r2 interf*) = *true* →
(*has_edge r1 r2 interf'*) = *true*).

Lemma *add_interf_op_correct*:
(*interf*: *graph*) (*res*: *reg*) (*live*: *D.T*) (*r*: *reg*)
live.[*r*] = *Boolean.Top* → *r* ≠ *res* →
(*has_edge r res* (*add_interf_op interf res live*)) = *true*.

Lemma *add_interf_op_increasing*:
(*interf*: *graph*) (*res*: *reg*) (*live*: *D.T*) (*r1*,*r2*: *reg*)
(*has_edge r1 r2 interf*) = *true* →
(*has_edge r1 r2* (*add_interf_op interf res live*)) = *true*.

Lemma *add_interf_move_correct*:
(*interf*: *graph*) (*arg*,*res*: *reg*) (*live*: *D.T*) (*r*: *reg*)
live.[*r*] = *Boolean.Top* → *r* ≠ *arg* → *r* ≠ *res* →
(*has_edge r res* (*add_interf_move interf arg res live*)) = *true*.

Lemma *add_interf_move_increasing*:
(*interf*: *graph*) (*arg*,*res*: *reg*) (*live*: *D.T*) (*r1*,*r2*: *reg*)
(*has_edge r1 r2 interf*) = *true* →
(*has_edge r1 r2* (*add_interf_move interf arg res live*)) = *true*.

Lemma *add_interf_instr_increasing*:
(*f*: *function*) (*live*: (*map D.T*)) (*interf*: *graph*) (*pc*: *nat*) (*r1*,*r2*: *reg*)
(*has_edge r1 r2 interf*) = *true* →
(*has_edge r1 r2* (*add_interf_instr f live interf pc*)) = *true*.

Lemma *add_interf_list_increasing*:
(*rl*: (*list reg*)) (*interf*: *graph*) (*live*: *D.T*) (*r1*,*r2*: *reg*)
(*has_edge r1 r2 interf*) = *true* →
(*has_edge r1 r2* (*add_interf_list interf live rl*)) = *true*.

Lemma *add_interf_list_correct*:

```
  (rl: (list reg)) (interf: graph) (live: D.T) (r1,r2: reg)
  (In r1 rl) → live.[r2] = Boolean.Top → r1 ≠ r2 →
  (has_edge r1 r2 (add_interf_list interf live rl)) = true.
```

The following complicated predicate defines exactly what edges are must be in an interference graph *interf* in order for this graph to reflect correctly the liveness information at point *pc*.

```
Definition correct_interf_instr
      [f: function; live: (map D.T); interf: graph; pc: nat] :=
  Cases (fn_instr f).(pc) of
    (Iop op args res) ⇒
      Cases (is_move_operation op args) of
        (Some arg) ⇒
          (r: reg)
          live.(pc).[res] = Boolean.Top → live.(pc).[r] = Boolean.Top →
          r ≠ res → r ≠ arg →
          (has_edge r res interf) = true
      | None ⇒
          (r: reg)
          live.(pc).[res] = Boolean.Top → live.(pc).[r] = Boolean.Top →
          r ≠ res →
          (has_edge r res interf) = true
      end
  | (Iload mode args res) ⇒
      (r: reg)
      live.(pc).[res] = Boolean.Top → live.(pc).[r] = Boolean.Top →
      r ≠ res →
      (has_edge r res interf) = true
  | (Icall nm args res) ⇒
      (r: reg)
      live.(pc).[r] = Boolean.Top →
      r ≠ res →
      (has_edge r res interf) = true
  | _ ⇒
      True
  end.
```

```
Lemma add_interf_instr_correct:
  (f: function) (live: (map D.T)) (interf: graph) (pc: nat)
  (correct_interf_instr f live (add_interf_instr f live interf pc) pc).
```

```
Lemma add_interf_instrs_increasing:
  (f: function) (live: (map D.T)) (pc: nat) (interf: graph) (r1,r2: reg)
  (has_edge r1 r2 interf) = true →
  (has_edge r1 r2 (add_interf_instrs f live interf pc)) = true.
```

```
Lemma correct_interf_instr_increasing:
  (f: function) (live: (map D.T)) (interf1, interf2: graph) (pc: nat)
```

```
((r1,r2: reg)
      (has_edge r1 r2 interf1) = true → (has_edge r1 r2 interf2) = true) →
(correct_interf_instr f live interf1 pc) →
(correct_interf_instr f live interf2 pc).
```

```
Lemma add_interf_instrs_correct:
(f: function) (live: (map D.T)) (pc: nat) (interf: graph)
((p: nat) (le pc p) → (le p (fn_last_pc f)) →
   (correct_interf_instr f live interf p)) →
((p: nat) (le p (fn_last_pc f)) →
   (correct_interf_instr f live (add_interf_instrs f live interf pc) p)).
```

The main result of this section is that the interference graph built by *interference_graph* captures the correct interferences at every point of the given function.

```
Lemma interference_graph_correct:
(f: function) (live: (map D.T)) (pc: nat)
(le pc (fn_last_pc f)) →
(correct_interf_instr f live (interference_graph f live) pc).
```

In addition, the interference graph also captures the correct interferences between the parameters of the function at the function entry point ("before" the instruction at point 0).

```
Lemma interference_graph_params:
(f: function) (live: (map D.T)) (r1,r2: reg)
(In r1 (fn_params f)) →
(transfer f O live.(O)).[r2] = Boolean.Top →
r1 ≠ r2 →
(has_edge r1 r2 (interference_graph f live)) = true.
```

13.4 Agreement between two register sets

```
Section Agree_live_regs.
```

In this section, we assume given an interference graph *interf* and consider a register assignment (a mapping from registers to registers) that is a coloring of the interference graph, i.e. if two registers interfere, they are assigned different registers.

```
Variable interf: graph.
Local assign := (graph_coloring interf).
```

Later in this module, we are going to rewrite the code by replacing every reference to register r by a reference to register *assign* r; then, we wish to prove the semantic equivalence between the original code and the transformed code. The key tool to do this is the following relation between a register set *rs1* in the original program and a register set *rs2* in the transformed program. The two register sets agree if they assign identical values to equivalent live registers, that is, register r in *rs1* and *assign* r in *rs2*. (The two register sets can disagree on dead registers, since the values of dead registers are never used by the program.)

Definition *agree_live_regs* [*live*: *D.T*; *rs1*,*rs2*: *regset*] :=
(*r*:*reg*) *live*.[*r*] = *Boolean.Top* → *rs1*.[*r*] = *rs2*.[*assign r*].

What follows is a long list of lemmas expressing properties of the *agree_live_regs* predicate that are useful for the semantic equivalence proof. First: two register sets that agree on a given set of live registers also agree on a subset of those live registers.

Lemma *agree_increasing*:
(*live1*,*live2*: *D.T*) (*rs1*,*rs2*: *regset*)
(*D.ge live1 live2*) →
(*agree_live_regs live1 rs1 rs2*) →
(*agree_live_regs live2 rs1 rs2*).

Two useful special cases of *agree_increasing*.

Lemma *agree_set_live_1*:
(*live*: *D.T*) (*r*: *reg*) (*rs1*,*rs2*: *regset*)
(*agree_live_regs* (*live*.[*r*] ← *Boolean.Top*) *rs1 rs2*) →
(*agree_live_regs live rs1 rs2*).

Lemma *agree_set_live_N*:
(*args*: (*list reg*)) (*live*: *D.T*) (*rs1*,*rs2*: *regset*)
(*agree_live_regs* (*set_live live args*) *rs1 rs2*) →
(*agree_live_regs live rs1 rs2*).

If two register sets agree on the registers live "after" an instruction n, they also agree on the registers live "before" every successor s of n.

Lemma *analysis_correct*:
(*f*: *function*) (*n*, *s*: *nat*)
(*le n* (*fn_last_pc f*)) → (*In s* (*successors f n*)) →
(*rs1*,*rs2*: *regset*)
(*agree_live_regs* (*analyze_function f*).(*n*) *rs1 rs2*) →
(*agree_live_regs* (*transfer f s* (*analyze_function f*).(*s*)) *rs1 rs2*).

Evaluating a list of registers in two register sets produces the same result if the register sets agree on at least the registers being evaluated.

Lemma *agree_eval_regs*:
(*args*: (*list reg*)) (*live*: *D.T*) (*rs1*, *rs2*: *regset*)
(*agree_live_regs* (*set_live live args*) *rs1 rs2*) →
(*eval_regs rs1 args*) = (*eval_regs rs2* (*PolyList.map assign args*)).

If a register is dead, assigning it an arbitrary value in *rs1* and leaving *rs2* unchanged preserves agreement. (This corresponds to an operation over a dead register in the original program that is turned into a no-op in the transformed program.)

Lemma *agree_assign_dead_reg*:
(*live*: *D.T*) (*r*: *reg*)
(*rs1*,*rs2*: *regset*) (*v*: *value*)
live.[*r*] = *Boolean.Bot* →
(*agree_live_regs live rs1 rs2*) →
(*agree_live_regs live* (*rs1*.[*r*] ← *v*) *rs2*).

Setting register r to the value v in *rs1* and setting register *assign* r to the value v in *rs2* preserves agreement, provided that all live registers except r interfere with r.

Lemma *agree_assign_live_reg*:
(*live*: *D.T*) (*r*: *reg*)
(*rs1*,*rs2*: *regset*) (*v*: *value*)
((*s*: *reg*)
live.[*s*] = *Boolean.Top* → *r* ≠ *s* → (*has_edge r s interf*) = *true*) →
(*agree_live_regs* (*live*.[*r*] ← *Boolean.Bot*) *rs1 rs2*) →
(*agree_live_regs live* (*rs1*.[*r*] ← *v*) (*rs2*.[*assign r*] ← *v*)).

This is a special case of the previous lemma where the value v stored in the registers is not arbitrary, but is the value of another register *arg*. (This corresponds to a register-register move instruction.) In this case, the condition can be weakened: it suffices that all live registers except *arg* and *res* interfere with *res*.

Lemma *agree_move_live_reg*:
(*live*: *D.T*) (*arg*,*res*: *reg*) (*rs1*,*rs2*: *regset*)
((*s*: *reg*)
live.[*s*] = *Boolean.Top* → *res* ≠ *s* → *arg* ≠ *s* →
(*has_edge res s interf*) = *true*) →
(*agree_live_regs*
(*set_live* (*live*.[*res*] ← *Boolean.Bot*) (*cons arg Nil*)) *rs1 rs2*) →
(*agree_live_regs live*
(*rs1*.[*res*] ← (*rs1*.[*arg*]))
(*rs2*.[*assign res*] ← (*rs2*.[*assign arg*]))).

This is another special case for "move" instructions, this time capturing the case where the "move" instruction is eliminated in the transformed program because the source and destination registers are assigned the same register.

Lemma *agree_dummy_move*:
(*live*: *D.T*) (*arg*,*res*: *reg*) (*rs1*,*rs2*: *regset*)
(*assign arg*) = (*assign res*) →
(*agree_live_regs*
(*set_live* (*live*.[*res*] ← *Boolean.Bot*) (*cons arg Nil*)) *rs1 rs2*) →
(*agree_live_regs live* (*rs1*.[*res*]←(*rs1*.[*arg*])) *rs2*).

This is a generalization of *agree_assign_live_reg* corresponding to storing a list of values into a list of registers. This preserves agreement provided every register in the list interferes with every live register.

Lemma *agree_set_regs*:
(*live*: *D.T*) (*vl*: (*list value*)) (*rl*: (*list reg*)) (*rs1*,*rs2*: *regset*)
((*r1*,*r2*: *reg*)
(*In r1 rl*) → *live*.[*r2*] = *Boolean.Top* → *r1* ≠ *r2* →
(*has_edge r1 r2 interf*) = *true*) →
(*agree_live_regs live rs1 rs2*) →
(*agree_live_regs live*
(*set_regs rs1 rl vl*) (*set_regs rs2* (*PolyList.map assign rl*) *vl*)).

This lemma shows agreement between the initial register sets at function entrance in the original

and transformed program.

```
Lemma agree_entry_regs:
  (live: D.T) (params: (list reg)) (vl: (list value))
  ((r1,r2: reg)
     (In r1 params) → live.[r2] = Boolean.Top → r1 ≠ r2 →
     (has_edge r1 r2 interf) = true) →
  (agree_live_regs live
     (set_regs (regmap_init value bad_value) params vl)
     (set_regs (regmap_init value bad_value) (PolyList.map assign params) vl)).

End Agree_live_regs.
```

13.5 Code transformations

Once the liveness information and the register assignment have been computed, we transform every function as follows:

- all references to registers are rewritten according to the register assignment;
- instructions that have no side-effect and whose result register is dead afterwards are eliminated (by turning them into *Inop* instructions);
- "move" instructions whose source and destination registers are identical after register assignment are eliminated (also by turning them into *Inop*).

The *Inop* instructions will be suppressed later (see module *Uselesscode*).

```
Definition transf_instr
              [live: (map D.T); assign: reg→reg; pc: nat; i: instruction] :=
  Cases i of
    Inop ⇒ Inop
  | (Iop op args res) ⇒
      Cases live.(pc).[res] of
        Boolean.Bot ⇒ Inop
      | Boolean.Top ⇒
          Cases (is_move_operation op args) of
            (Some arg) ⇒
              Cases (reg_eq_dec (assign arg) (assign res)) of
                (left _) ⇒ Inop
              | (right _) ⇒ (Iop op (PolyList.map assign args) (assign res))
              end
          | None ⇒
              (Iop op (PolyList.map assign args) (assign res))
          end
      end
  | (Iload mode args res) ⇒
      Cases live.(pc).[res] of
```

```
        Boolean.Bot ⇒ Inop
      | Boolean.Top ⇒ (Iload mode (PolyList.map assign args) (assign res))
      end
  | (Istore mode args src) ⇒
      (Istore mode (PolyList.map assign args) (assign src))
  | (Icall fn args res) ⇒
      (Icall (assign fn) (PolyList.map assign args) (assign res))
  | (Ibranch label) ⇒ (Ibranch label)
  | (Icondbranch cond args label) ⇒
      (Icondbranch cond (PolyList.map assign args) label)
  | (Ireturn arg) ⇒
      (Ireturn (assign arg))
  end.
```

Lemma *transf_instr_valid*:
(*ra*: (*map D.T*)) (*assign*: *reg*→*reg*) (*last_pc*: *nat*) (*n*: *nat*)(*i*: *instruction*)
(*instr_valid i last_pc*) →
(*instr_valid* (*transf_instr ra assign n i*) *last_pc*).

Lemma *transf_instr_terminal*:
(*ra*: (*map D.T*)) (*assign*: *reg*→*reg*) (*n*: *nat*) (*i*: *instruction*)
(*instr_terminal i*) →
(*instr_terminal* (*transf_instr ra assign n i*)).

Putting all the pieces together, transforming a function involves performing liveness allocation, computing the interference graph, coloring it, and rewriting the instructions and the parameters of the function accordingly.

```
Definition transf_function [n:nat; f:function] :=
  let live = (analyze_function f) in
  let assign = (graph_coloring (interference_graph f live)) in
  (make_function
    (map_apply instruction instruction
                (transf_instr live assign) (fn_instr f) (S (fn_last_pc f)))
    (PolyList.map assign (fn_params f))
    (fn_last_pc f)
    (Instr.transf_is_valid
       (transf_instr live assign) (transf_instr_valid live assign) f)
    (Instr.transf_last_instr_terminal
       (transf_instr live assign) (transf_instr_terminal live assign) f)).
```

```
Definition transf_program [p: program] :=
  (Instr.transf_program transf_function p).
```

Lemma *transformed_instr*:
(*f*: *function*) (*nm*: *nat*) (*pc*: *nat*)
(*le pc* (*fn_last_pc f*)) →
(*fn_instr* (*transf_function nm f*)).(*pc*) =
(*transf_instr*

(*analyze_function f*)
(*graph_coloring* (*interference_graph f* (*analyze_function f*)))
pc (*fn_instr f*).(*pc*)).

Lemma *transformed_function*:
(*p*: *program*) (*nm*: *nat*)
(*lt nm* (*prog_num_functions p*)) →
(*prog_function* (*transf_program p*)).(*nm*) =
(*transf_function nm* (*prog_function p*).(*nm*)).

13.6 Transformation preserves semantics

We now show that transformed programs behave like the original programs. The core result is the following simulation property:
Check
(*p*: *program*)(*f*: *function*)
(*pc*: *nat*) (*rs*: *regset*) (*st*: *store*)
(*pc'*: *nat*) (*rs'*: *regset*) (*st'*: *store*)
(*exec_instr p f pc rs st pc' rs' st'*) →
(*nm*: *nat*) (*rs1*: *regset*)
(*le pc* (*fn_last_pc f*)) →
(*agree_live_regs* (*interference_graph f* (*analyze_function f*))
(*transfer f pc* (*analyze_function f*).(*pc*)) *rs rs1*) →
(*EX rs1'* |
(*exec_instr* (*transf_program p*) (*transf_function nm f*)
pc rs1 st pc' rs1' st')
∧ (*agree_live_regs* (*interference_graph f* (*analyze_function f*))
(*analyze_function f*).(*pc*) *rs' rs1'*)).

What this means is that if we have two register sets *rs* and *rs1* that agree on the registers live before the instruction at *pc*, and the original program makes a transition from (*pc*, *rs*, *st*) to (*pc'*, *rs'*, *st'*), then there exists a register set *rs1'* such that the transformed program makes a transition from (*pc*, *rs1*, *st*) to (*pc'*, *rs1'*, *st'*), and moreover *rs'* and *rs1'* agree on the registers live after the instruction at *pc*.

We first show this property for the two most difficult cases: the instruction at *pc* is an *Iop* or an *Iload*.

Lemma *transf_op_correct*:
(*p*: *program*)(*f*: *function*) (*pc*: *nat*) (*rs*: *regset*) (*st*: *store*)
(*op*: *operation*) (*args*: (*list reg*)) (*res*: *reg*)
(*fn_instr f*).(*pc*) = (*Iop op args res*) →
(*nm*:*nat*; *rs1*:*regset*)
(*le pc* (*fn_last_pc f*)) →
(*agree_live_regs* (*interference_graph f* (*analyze_function f*))
(*transfer f pc* (*analyze_function f*).(*pc*)) *rs rs1*) →
(*EX rs1'*: *regset* |

```
      (exec_instr (transf_program p) (transf_function nm f)
                  pc rs1 st (S pc) rs1' st)
  ∧ (agree_live_regs
       (interference_graph f (analyze_function f))
       (analyze_function f).(pc)
       (rs.[res] ←(eval_operation op (eval_regs rs args)))
       rs1')).
```

Lemma *transf_load_correct*:

```
(p: program) (f:function) (pc:nat) (rs:regset) (st:store)
(mode:addressing_mode) (addrs:(list reg)) (res:reg)
(fn_instr f).(pc)=(Iload mode addrs res) →
(nm:nat)(rs1:regset)
(le pc (fn_last_pc f)) →
(agree_live_regs (interference_graph f (analyze_function f))
                 (transfer f pc (analyze_function f).(pc)) rs rs1) →
(EX rs1': regset|
    (exec_instr (transf_program p) (transf_function nm f)
                pc rs1 st (S pc) rs1' st)
 ∧ (agree_live_regs
      (interference_graph f (analyze_function f))
      (analyze_function f).(pc)
      (rs.[res] ← (st.(eval_addressing mode (eval_regs rs addrs)))) rs1')).
```

We now show the general case, by mutual induction with the following correctness property of the *exec_function* predicate.

Lemma *transf_function_correct*:

```
(p: program)(f: function)
(pc: nat) (rs: regset) (st: store) (res: value) (st': store)
(exec_function p f pc rs st res st') →
(nm: nat) (rs1: regset)
(le pc (fn_last_pc f)) →
(agree_live_regs (interference_graph f (analyze_function f))
                 (transfer f pc (analyze_function f).(pc)) rs rs1) →
(exec_function (transf_program p) (transf_function nm f) pc rs1 st res st').
```

It follows that the transformed program behaves identically to the original program.

Lemma *transf_program_correct*:

```
(p: program) (st: store) (res: value) (st': store)
(exec_program p st res st') →
(exec_program (transf_program p) st res st').
```

14 Module *Uselesscode*: dead code elimination

This module cleans up the optimized code produced by constant propagation and register allocation. It removes unreachable instructions and no-op instructions. While this sounds easy, the

representation of function code as instructions for a virtual machine makes these transformations non-trivial. In particular, suppressing instructions requires changing the PC of the remaining instructions, and recomputing branch targets. That this is non-trivial explains why previous optimization passes, e.g. the *Allocation* module, do not suppress useless instructions but turn them into *Inop* instructions: it is more convenient to do the actual removal of the useless instructions in a separate pass.

Require *PolyList*. Require *Le*. Require *Lt*. Require *Plus*.
Require *Misc*. Require *Map*. Require *Lattice*. Require *Mach_ops*.
Require *Instr*. Require *Reg*. Require *Kildall*.

14.1 Reachability analysis

Determining reachable instructions is a trivial forward data flow analysis. To each instruction, we associate a boolean ("reachable" or "unreachable"), with the constraints that the entry point of a function is reachable, and the successors of a reachable instruction are reachable. This corresponds to forward data flow analysis with the identity function as transfer function.

Module *Solver* := (*Dataflow_Solver Boolean*).

Definition *transfer* [*pc*: *nat*; *reachable*: *Boolean.T*] := *reachable*.

Definition *analyze_function* [*f*: *function*] : (*map Boolean.T*) :=
  (*Solver.fixpoint*
    (*fn_last_pc f*)
    (*successors f*)
    (*successors_in_graph f*)
    transfer
    (*cons* (*O*, *Boolean.Top*) *Nil*)).

The result of the analysis satisfies the two properties mentioned above: the successors of a reachable instruction are reachable, and the entry point (PC 0) is reachable.

Lemma *successor_reachable*:
  (*f*: *function*) (*n*,*s*: *nat*)
  (*le n* (*fn_last_pc f*)) →
  (*In s* (*successors f n*)) →
  (*analyze_function f*).(*n*) = *Boolean.Top* →
  (*analyze_function f*).(*s*) = *Boolean.Top*.

Lemma *entry_point_reachable*:
  (*f*: *function*)
  (*analyze_function f*).(*O*) = *Boolean.Top*.

Corollary: if the instruction at PC n is reachable, but not the instruction at PC n+1, then the instruction at n must be a terminal instruction (*Ibranch* or *Ireturn*).

Lemma *discontinuity_terminal*:
  (*f*: *function*) (*n*: *nat*)
  (*le n* (*fn_last_pc f*)) →

```
(analyze_function f).(n) = Boolean.Top →
(analyze_function f).(S n) = Boolean.Bot →
(instr_terminal (fn_instr f).(n)).
```

There exists a "last reachable" instruction: an instruction that is reachable, but all following instructions are not.

```
Lemma last_reachable:
  (f: function)
  (EX pc | (le pc (fn_last_pc f))
        ∧ (analyze_function f).(pc) = Boolean.Top
        ∧ ((p: nat) (lt pc p) → (le p (fn_last_pc f)) →
                    (analyze_function f).(p) = Boolean.Bot)).
```

This "last reachable" instruction must be a terminal instruction.

```
Lemma last_reachable_terminal:
  (f: function) (pc: nat)
  (le pc (fn_last_pc f)) →
  (analyze_function f).(pc) = Boolean.Top →
  ((p: nat) (lt pc p) → (le p (fn_last_pc f)) →
                        (analyze_function f).(p) = Boolean.Bot) →
  (instr_terminal (fn_instr f).(pc)).
```

14.2 Code transformation

We first define an auxiliary function that will be useful in defining and proving properties of the code transformation. Given a mapping *keep* from integers to booleans, (*count_true keep n*) returns the number of integers $0 \leq p < n$ such that *keep*.(*p*) is *true*.

```
Section Count_true.

Variable keep: (map bool).

Fixpoint count_true [p: nat] : nat :=
  Cases p of
    O ⇒ O
  | (S p') ⇒
      if keep.(p') then (S (count_true p')) else (count_true p')
  end.

Lemma count_true_increasing:
  (q, p: nat)
  (le p q) → (le (count_true p) (count_true q)).

Lemma count_true_constant:
  (q, p: nat)
  (le p q) →
  ((r: nat) (le p r) → (lt r q) → keep.(r) = false) →
  (count_true p) = (count_true q).
```

```
Lemma count_true_continuous:
  (p: nat) (v: nat)
  (lt v (count_true p)) →
  (EX q | (lt q p) ∧ keep.(q) = true ∧ (count_true q) = v).

End Count_true.
```

We now define the transformation of a function f.

```
Section Transform_function.

Variable f: function.

Definition reach_f := (analyze_function f).
```

We first compute a mapping from PC to booleans giving, for each instruction, whether to keep it (*true*) or remove it (*false*). An instruction is to be removed if it is unreachable, or it is a *Inop* instruction.

```
Definition keep_instr [reach: (map Boolean.T); pc: nat; i: instruction] :=
  Cases reach.(pc) of
    Boolean.Bot ⇒ false
  | Boolean.Top ⇒
      Cases i of
        Inop ⇒ false
      | _ ⇒ true
      end
  end.

Definition keep_code [reach: (map Boolean.T)] :=
  (map_apply instruction bool
    (keep_instr reach) (fn_instr f) (S (fn_last_pc f))).

Definition keep_f := (keep_code reach_f).

Lemma keep_code_at:
  (reach: (map Boolean.T)) (pc: nat)
  (le pc (fn_last_pc f)) →
  (keep_code reach).(pc) = (keep_instr reach pc (fn_instr f).(pc)).

Lemma keep_f_at:
  (pc: nat)
  (le pc (fn_last_pc f)) →
  keep_f.(pc) = (keep_instr reach_f pc (fn_instr f).(pc)).

Lemma code_mapping_entry:
  (keep: (map bool)) (p: nat)
  (le p (fn_last_pc f)) →
  (code_mapping keep).(p) = (count_true keep p).
```

There exists a "last kept" instruction with the following characteristics: it is kept; all following instructions are unreachable; it is terminal; and the size of the transformed code is one more than

the new PC of this instruction. Unsurprisingly, this "last kept" instruction is exactly the "last reachable" instruction that we constructed in the previous section.

```
Lemma code_mapping_last:
  (EX pc | (le pc (fn_last_pc f))
         ∧ keep_f.(pc) = true
         ∧ new_code_size = (S (count_true keep_f pc))
         ∧ (instr_terminal (fn_instr f).(pc))
         ∧ ((p: nat) (lt pc p) → (le p (fn_last_pc f)) →
                        reach_f.(p) = Boolean.Bot)).
```

As a consequence, the new PC for every reachable instruction is less than or equal to the new PC of the last kept instruction. This shows that the branch instructions in the transformed code will be valid.

```
Lemma code_mapping_in_range:
  (p: nat)
  (le p (fn_last_pc f)) →
  reach_f.(p) = Boolean.Top →
  (le mapping_f.(p) (pred new_code_size)).
```

We now build the code of the transformed function. It is obtained by eliminating instructions that should not be kept, and adjusting the targets of branch instructions according to the PC mapping previously computed.

```
Definition relocate_instr [mapping: (map nat); i: instruction] :=
  Cases i of
    (Ibranch dest) ⇒ (Ibranch mapping.(dest))
  | (Icondbranch cond args dest) ⇒ (Icondbranch cond args mapping.(dest))
  | _ ⇒ i
  end.

Fixpoint transf_code_aux
     [keep: (map bool); mapping: (map nat);
      newcode: (map instruction); pc: nat; newpc: nat;
      count: nat] : (map instruction) :=
  Cases count of
    O ⇒ newcode
  | (S count') ⇒
      if keep.(pc) then
        (transf_code_aux keep mapping
          (newcode.(newpc) ← (relocate_instr mapping (fn_instr f).(pc)))
          (S pc) (S newpc) count')
      else
        (transf_code_aux keep mapping
          newcode
          (S pc) newpc count')
  end.

Definition transf_code [keep: (map bool); mapping: (map nat)] :=
```

```
(transf_code_aux keep mapping (map_init instruction Inop)
                 O O (S (fn_last_pc f))).
```

Definition *transf_f* := (*transf_code keep_f mapping_f*).

The following technical lemmas establish a correspondence between instructions of the original code and those of the transformed code.

Remark *transf_code_aux_characterization*:

```
(count: nat) (newcode: (map instruction)) (pc: nat)
(plus pc count) = (S (fn_last_pc f)) →
let finalcode = (transf_code_aux keep_f mapping_f newcode
                                 pc (count_true keep_f pc) count) in
((p: nat) (lt p pc) → keep_f.(p) = true →
    newcode.(mapping_f.(p)) = (relocate_instr mapping_f (fn_instr f).(p))) →
((p: nat) (lt p (plus pc count)) → keep_f.(p) = true →
    finalcode.(mapping_f.(p)) = (relocate_instr mapping_f (fn_instr f).(p))).
```

Lemma *transf_code_at*:

```
(pc: nat)
(le pc (fn_last_pc f)) →
keep_f.(pc) = true →
transf_f.(mapping_f.(pc)) = (relocate_instr mapping_f (fn_instr f).(pc)).
```

Lemma *reverse_transf_code_at*:

```
(newpc: nat)
(lt newpc new_code_size) →
(EX pc |
    (le pc (fn_last_pc f))
∧ transf_f.(newpc) = (relocate_instr mapping_f (fn_instr f).(pc))
∧ keep_f.(pc) = true).
```

We now show that the transformed code is well-formed: each transformed instruction is valid, and the last transformed instruction is terminal.

Lemma *transf_code_valid*:

```
(pc: nat)
  (le pc (pred new_code_size)) →
  (instr_valid transf_f.(pc) (pred new_code_size)).
```

Lemma *transf_code_last_instr_terminal*:

```
(instr_terminal transf_f.(pred new_code_size)).
```

We can at last define the transformation of functions and of programs.

Definition *transf_function* :=

```
let reach = (analyze_function f) in
let keep = (keep_code reach) in
let mapping = (code_mapping keep) in
(make_function
    (transf_code keep mapping)
```

(*fn_params f*)
(*pred* (*count_true keep* (*S* (*fn_last_pc f*))))
transf_code_valid
transf_code_last_instr_terminal).

End *Transform_function*.

Definition *transf_program* [*p*: *program*] :=
(*transf_program* ([*n*: *nat*; *f*: *function*] (*transf_function f*)) *p*).

14.3 Transformation preserves semantics

We now show that the transformed program behaves like the original program. The key result is the following simulation property: if the original program executes one instruction from state (*pc*, *rs*, *st*) to state (*pc'*, *rs'*, *st'*), and this instruction is reachable, then either:

- this instruction was a no-op: $rs' = rs$ and $st' = st$, and *pc* and *pc'* are mapped to the same PC in the transformed code
- or the transformed program executes the corresponding instruction from state ((*mapping_f f*).(*pc*), *rs*, *st*) to state ((*mapping_f f*).(*pc'*), *rs'*, *st'*).

Check
(*p*: *program*) (*f*: *function*)
(*pc*: *nat*) (*rs*: *regset*) (*st*: *store*)
(*pc'*: *nat*) (*rs'*: *regset*) (*st'*: *store*)
(*exec_instr p f pc rs st pc' rs' st'*) →
(*le pc* (*fn_last_pc f*)) →
(*reach_f f*).(*pc*) = *Boolean.Top* →
((*mapping_f f*).(*pc*) = (*mapping_f f*).(*pc'*) ∧ *rs* = *rs'* ∧ *st* = *st'*)
∨ (*exec_instr* (*transf_program p*) (*transf_function f*)
(*mapping_f f*).(*pc*) *rs st* (*mapping_f f*).(*pc'*) *rs' st'*).

Some useful technical lemmas first.

Lemma *transformed_instr*:
(*f*: *function*) (*pc*: *nat*)
(*le pc* (*fn_last_pc f*)) →
(*keep_f f*).(*pc*) = *true* →
(*fn_instr* (*transf_function f*)).((*mapping_f f*).(*pc*)) =
(*relocate_instr* (*mapping_f f*) (*fn_instr f*).(*pc*)).

Lemma *transformed_function*:
(*p*: *program*) (*nm*: *nat*)
(*lt nm* (*prog_num_functions p*)) →
(*prog_function* (*transf_program p*)).(*nm*) =
(*transf_function* (*prog_function p*).(*nm*)).

Lemma *mapping_f_succ*:

```
(f: function) (pc: nat)
(le pc (fn_last_pc f)) →
(reach_f f).(pc) = Boolean.Top →
¬(instr_terminal (fn_instr f).(pc)) →
(mapping_f f).(S pc) =
  Cases (fn_instr f).(pc) of
    Inop ⇒ (mapping_f f).(pc)
  | _ ⇒ (S (mapping_f f).(pc))
  end.
```

The main simulation result above is proved by mutual induction with the following property of the *exec_function* predicate.

```
Lemma transf_function_correct:
  (p: program)(f: function)
  (pc: nat) (rs: regset) (st: store) (res: value) (st': store)
  (exec_function p f pc rs st res st') →
  (le pc (fn_last_pc f)) →
  (reach_f f).(pc) = Boolean.Top →
  (exec_function (transf_program p) (transf_function f)
                 (mapping_f f).(pc) rs st res st').
```

```
Lemma transf_program_correct:
  (p: program) (st: store) (res: value) (st': store)
  (exec_program p st res st') →
  (exec_program (transf_program p) st res st').
```

15 Concluding remarks

The Coq development presented here is a first attempt, so it is perhaps too early to draw conclusions. However, some lessons can already be drawn from it.

The static analyses are perhaps the simplest and least problematic parts of the optimization passes considered. The general framework for dataflow analysis is not trivial, but needs only be proved once, and is easy to re-use in each optimization pass. Despite a few rough edges, the new module system of Coq 7.4 helped a lot to structure this part of the development and allow its re-use. For more complex optimization passes such as common subexpression elimination, the only foreseeable difficulty is in constructing the domain (the semi-lattice) for the analysis: on the one hand, it must have the right mathematical properties, mostly well-foundedness; on the other hand, it must be algorithmically efficient, i.e. support efficiently the basic operations needed by the transfer function and by the subsequent code transformation.

The code transformations following static analysis are generally easy, but can get tricky if instructions need to be deleted or inserted. This is probably a consequence of a bad design of the intermediate language (see below). Moreover, unlike the static analyses, the transformations are quite ad-hoc and specific to each optimization.

The proofs of semantic equivalence are not as difficult as originally expected. Proving the core simulation lemma takes only one to two pages of (densely packed) Coq proof script. The mutual induction principle between *exec_instr* and *exec_function* produces exactly the right proof

obligations: one per kind of instruction (*Inop*, *Iop*, etc) and one for the "step" rule (chaining the execution of one instruction with the execution of the remainder of the function). However, the simluation proofs require a fairly large amount of technical lemmas on auxiliary predicates. Finding the correct predicates, discovering the required lemmas, and proving them is the most time-consuming part of the development.

The intermediate language and its semantics have some weaknesses. First, the linear presentation of the code (a non-branch instruction at *pc* always continue with instruction at $pc + 1$) makes it painful to actually remove instructions, and even more painful to insert new instructions. In retrospect, it would have been preferable to represent the code of a function as a real control-flow graph, with each instruction carrying an explicit list of successor instructions. Adapting the present development to this alternate representation does not look too difficult.

Another weakness is the use of *bad_value* as the initial value of registers that are not function parameters, and as the result of ill-formed operations (e.g. *Oadd* applied to three registers). This decision causes programs that are essentially ill-formed to evaluate correctly and deterministically (although the result is not fully specified because the actual bit-pattern of *bad_value* is unspecified). Therefore, program transformations must also preserve the behavior of these ill-formed programs. This constraint is unimportant for the code transformations presented here, but could become impossible to ensure for other transformations. There are two possible solutions to this problem. The first is to change the dynamic semantics: the contents of registers could be of type (*option value*) rather than *value*, so that accesses to uninitialized registers can be prevented by the reduction rules. The second solution is to strengthen the static well-formedness conditions on function code to include some well-typing conditions similar to those of Java bytecode verification, thus guaranteeing the absence of references to uninitialized registers, just like bytecode verification does.

This work can be continued in two directions. The first is to formalize and prove correct other optimizations, such as common subexpression elimination. The second is to move towards the generation of actual assembly code: register spilling and reloading; introduction of calling conventions (arguments and results being explicitly passed in fixed registers rather than implicitly moved by the *Icall* and *Ireturn* pseudo-instructions); introduction of a stack to hold return addresses and spilled registers; and development of a machine-level semantics for instructions. This semantics would be essentially identical to that shown in the *Instr* module as regards the *Iop*, *Iload*, *Istore*, *Ibranch* and *Icondbranch* families of instructions, but requires significant changes to the *Icall* and *Ireturn* instructions.

The computational parts of this development (the definitions of Coq functions) was carefully written in a style that should extract to reasonably efficient Caml code. In particular, some attention was paid to let-bind complex intermediate results rather than recompute them all over the place. Howver, preliminary attemps at extracting Caml code failed early on the *map* type constructor: I was unable to explain Coq that this constructor should translate to a user-provided Caml type constructor.