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 sections 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.


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 ≠ b" := ¬(a = b).

2.2 Reflexive closure of a relation

Section Refl_closure.

Variable A: Set.
Variable r: A→A→Prop.

Definition of the reflexive closure of relation r.

Definition refl_closure := [x,y:A] x=y ∨ (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) → (refl_closure y z) → (r x z).

Lemma trans2_refl_closure:
(x,y,z:A) (refl_closure x y) → (r y z) → (r x z).

End Refl_closure.
2.3 Useful lemmas

Lemma lt_not_eq: \((x,y: \mathbb{N}) \rightarrow x \neq y\).
Lemma lt_ge_dec: \((x,y: \mathbb{N}) \{lt x y\} + \{le y x\}\).
Lemma not_in_cons:
(a: \mathbb{N})(succs:(\mathbb{N}\rightarrow\mathbb{N}))(s: \mathbb{N})
\neg(In s (cons a succs)) \rightarrow s \neq a \land \neg(In s succs).
Implicit Arguments On.
Lemma in_singleton: (A:Set)(x:A)(y:A) (In x (cons y (nil A))) \rightarrow 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 quantifi-
cations are automatically eliminated and simplified before being introduced in the hypotheses.

Recursive Tactic Definition IntroElim :=
Match Context With
| \vdash (id: (EX \ x | ?)) \ ? \ \rightarrow Intro id; Elim id; Clear id; IntroElim
| \vdash (id: ? \land ?) \ ? \ \rightarrow Intro id; Elim id; Clear id; IntroElim
| \vdash (id: ? \lor ?) \ ? \ \rightarrow Intro id; Elim id; Clear id; IntroElim
| \vdash ? \rightarrow ? \ \rightarrow Intro; IntroElim
| \ \rightarrow 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, corre-
spanding 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 the transition function from one loop iteration to the next; and \( stop \) is the stop criterion.

**Variable** \( A \): Set.
**Variable** \( trans \): \( A \rightarrow A \).
**Variable** \( stop \): \( A \rightarrow \text{bool} \).

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

**Variable** \( ltA \): \( A \rightarrow A \rightarrow \text{Prop} \).

**Hypothesis** \( \text{well\_founded\_ltA} \): (\( \text{well\_founded} \ A \ ltA \)).

**Hypothesis** \( \text{trans\_decreasing} \):
\[
(x:A) \ (\text{stop } x) = \text{false} \rightarrow (\text{ltA} \ (\text{trans } x) \ x).
\]

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

**Definition** \( \text{while\_trans} \):
\[
(x:A)((y:A)(\text{ltA} \ y \ x)\rightarrow A)\rightarrow A := \begin{cases}
[x:A] \ [f: (y:A)(\text{ltA} \ y \ x) \rightarrow A]
\ 
\text{Cases } (\text{sumbool\_of\_bool} \ (\text{stop} \ x)) \ \text{of}
\ 
\ (\text{left } \_ ) \Rightarrow x
\ | \ (\text{right } \text{pfalse} ) \Rightarrow (f \ (\text{trans} \ x) \ (\text{trans\_decreasing} \ x \ \text{pfalse}))
\end{cases}
\]

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

The transition step satisfies the extensionality properties required to show that the result of \( \text{while} \) is a fixpoint.

**Remark** \( \text{while\_trans\_extensional} \):
\[
(x:A) \ (f,g: (y:A)(\text{ltA} \ y \ x)\rightarrow A)
\ 
\ ((y:A; p:(\text{ltA} \ y \ x))(f \ y \ p) = (g \ y \ p)) \rightarrow
\ 
\ (\text{while\_trans} \ x \ f) = (\text{while\_trans} \ x \ g).
\]

\( \text{while} \) does compute a fixpoint of the \( trans \) function.

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

The \( stop \) condition holds for the result of \( \text{while} \).

**Lemma** \( \text{while\_satisfies\_stop} \):
\[
(x:A) \ (\text{stop} \ (\text{while } x)) = \text{true}.
\]

If \( transf \) preserves an invariant \( P \), and the start value of the while loop satisfies \( P \), then its final value satisfies \( P \) as well.
Variable $P: A \rightarrow \text{Prop}$.

Hypothesis $\text{trans\_preserves\_P}$: $(x:A) (\text{stop} x) = \text{false} \rightarrow (P x) \rightarrow (P (\text{trans} x))$.

Lemma $\text{while\_preserves\_invariant}$: $(x:A) (P x) \rightarrow (P (\text{while} x))$.

End While.

4 Module $\text{WFOder}$: well-founded orderings

The $\text{WFOder}$ 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 $\text{Wellfounded}$. Require $\text{Relations}$.

Section $\text{Pair\_Lexico}$.

Variable $A, B$: Set.

Variable $\text{gtA}$: $A \rightarrow A \rightarrow \text{Prop}$.

Variable $\text{gtB}$: $B \rightarrow B \rightarrow \text{Prop}$.

Hypothesis $\text{wfA}$: $(\text{well\_founded} A \text{gtA})$.

Hypothesis $\text{wfB}$: $(\text{well\_founded} B \text{gtB})$.

Inductive $\text{gt\_pair}$: $A \times B \rightarrow A \times B \rightarrow \text{Prop}$ :=

$\text{gt\_pair\_fst}$:

$(a:A)(b:B)(a':A)(b':B)$

$(\text{gtA} a a') \rightarrow (\text{gt\_pair} (a,b) (a',b'))$

$\text{gt\_pair\_snd}$:

$(a:A)(b:B)(b':B)$

$(\text{gtB} b b') \rightarrow (\text{gt\_pair} (a,b) (a',b'))$.

Remark $\text{acc\_gt}$:

$(a:A) (\text{Acc} A \text{gtA} a) \rightarrow$

$(b:B) (\text{Acc} B \text{gtB} b) \rightarrow$

$(\text{Acc} (A \times B) \text{gt\_pair} (a,b))$.

Lemma $\text{wf\_gt\_pair}$: $(\text{well\_founded} (A \times B) \text{gt\_pair})$.

Hypothesis $\text{transitive\_gtA}$: $(\text{transitive} A \text{gtA})$.

Hypothesis $\text{transitive\_gtB}$: $(\text{transitive} B \text{gtB})$.

Lemma $\text{transitive\_gt\_pair}$: $(\text{transitive} (A \times B) \text{gt\_pair})$.

End $\text{Pair\_Lexico}$.

5 Module $\text{Lattice}$: semi-lattices

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

Require $\text{Relations}$. Require $\text{Misc}$. Require $\text{WFOder}$.
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 $gt$: $T \rightarrow T \rightarrow Prop$.

Parameter $wf$: (well_founded $T \, gt$).

Parameter $trans$: (transitive $T \, gt$).

The “greater or equal” ordering derived from $gt$.

Definition $ge := (\text{refl\_closure} \, T \, gt)$.

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 $lub$: $T \rightarrow T \rightarrow T$.

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.


Definition $T := T_-$.

Same trick for the $gt$ ordering, which is defined via the inductive predicate $gt_-$.

Inductive $gt_-: T \rightarrow T \rightarrow Prop :=$

$gt\_intro: (gt_- \, Top \, Bot)$.

Definition $gt := gt_-$.

Remark acc_Top: (Acc $T \, gt \, Top$).
Remark \textit{acc}_{Bot}: (Acc \ T \ gt \ Bot).

Lemma \textit{wf}: (\textit{well}_{founded} \ T \ gt).

Lemma \textit{trans}: (\textit{transitive} \ T \ gt).

Lemma \textit{eq}_{dec}: (x,y:T) \{x=y\} + \{x\neq y\}.

Definition \textit{ge} := (\textit{refl}_\textit{closure} \ T \ gt).

Definition \textit{top} := Top.

Definition \textit{bot} := Bot.

Lemma \textit{ge}_{top} \ x: (x:T) (\textit{ge} \ top \ x).

Lemma \textit{ge}_{x} \ bot: (x:T) (\textit{ge} \ x \ bot).

Definition \textit{lub}[x,y:T] :=
\begin{align*}
\text{Cases } x \text{ of} \\
& \text{Bot } \Rightarrow y \\
& \text{Top } \Rightarrow \text{Top}
\end{align*}

Lemma \textit{lub}_\textit{commut}: (x,y:T) (\textit{lub} \ x \ y) = (\textit{lub} \ y \ x).

Lemma \textit{ge}_{lub}_\textit{left}: (x,y:T) (\textit{ge} \ (\textit{lub} \ x \ y) \ x).

End \textit{Boolean}.

5.3 Flat semi-lattice

The \textit{Flat} functor below defines the flat semi-lattice over any type for which equality is decidable.

Module Type \textit{TYPE}_{DEC\_EQ}.

Parameter \textit{T}: \textit{Set}.

Parameter \textit{eq}_{dec}: (x,y:T) \{x=y\} + \{x\neq y\}.

End \textit{TYPE}_{DEC\_EQ}.

Module \textit{Flat}[A: \textit{TYPE}_{DEC\_EQ}].

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

Inductive \textit{T}_: Set := \textit{Bot} : T_\_ | \textit{Inj}: A.T \rightarrow T_\_ | \textit{Top} : T_\_.

Definition \textit{T} := T_\_.

Definition \textit{top} := \textit{Top}.

Definition \textit{bot} := \textit{Bot}.

Definition \textit{inj} := \textit{Inj}.

Inductive \textit{gt}_\_: T \rightarrow T \rightarrow \textit{Prop} :=
\begin{align*}
& \textit{gt}_{\textit{inj} \_ \textit{bot}}: (x: A.T) (\textit{gt}_\_ \ (\textit{Inj} \ x) \ \textit{Bot}) \\
& | \textit{gt}_{\textit{top} \_ \textit{bot}}: (\textit{gt}_\_ \ \textit{Top} \ \textit{Bot}) \\
& | \textit{gt}_{\textit{top} \_ \textit{inj}}: (x: A.T) (\textit{gt}_\_ \ \textit{Top} \ (\textit{Inj} \ x)).
\end{align*}
Definition \( \text{gt} := \text{gt}_\_ \).

Remark \( \text{acc}_\text{Top} : (\text{Acc} \ T \ \text{gt} \ \text{Top}) \).

Remark \( \text{acc}_\text{Inj} : (x : A.T) (\text{Acc} \ T \ \text{gt} \ (\text{Inj} \ x)) \).

Remark \( \text{acc}_\text{Bot} : (\text{Acc} \ T \ \text{gt} \ \text{Bot}) \).

Lemma \( \text{wf} : (\text{well} \text{founded} \ T \ \text{gt}) \).

Lemma \( \text{trans} : (\text{transitive} \ T \ \text{gt}) \).

Definition \( \text{ge} := (\text{refl} \text{ closure} \ T \ \text{gt}) \).

Lemma \( \text{eq} \text{ dec} : (x,y:T) \{x=y\} + \{x\neq y\} \).

Lemma \( \text{ge}_\text{top} \_x : (x:T) (\text{ge} \ \text{top} \ x) \).

Lemma \( \text{ge}_x \_\text{bot} : (x:T) (\text{ge} \ x \ \text{bot}) \).

Definition \( \text{lub} \ [x,y:T] := \)

\[
\begin{align*}
\text{Cases} \ x \ \text{of} \\
\text{Bot} & \Rightarrow y \\
| \ Top & \Rightarrow Top \\
| (\text{Inj} \ a) & \Rightarrow \\
\text{Cases} \ y \ \text{of} \\
\text{Bot} & \Rightarrow (\text{Inj} \ a) \\
| \ Top & \Rightarrow Top \\
| (\text{Inj} \ b) & \Rightarrow \\
\text{Cases} \ (A.\text{eq} \_\text{dec} \ a \ b) \ \text{of} \\
(\text{left} \_ & \Rightarrow (\text{Inj} \ a) \\
| (\text{right} \_ & \Rightarrow \text{Top} \\
end \\
end.
\end{align*}
\]

Lemma \( \text{lub} \_\text{com}\text{mut} : (x,y:T) (\text{lub} \ x \ y) = (\text{lub} \ y \ x) \).

Lemma \( \text{ge}_\text{lub} \_\text{left} : (x,y:T) (\text{ge} \ (\text{lub} \ x \ y) \ x) \).

End Flat.

5.4 Product semi-lattice

The \( \text{Product} \) functor below defines the product of two semi-lattices, ordered lexicographically.

Module \( \text{Product}[A,B: \text{SEMILATTICE}] \).

Definition \( T := A.T \times B.T \).

Definition \( \text{gt} := (\text{gt}_\text{pair} A.T \ B.T \ A.\text{gt} \ B.\text{gt}) \).

Lemma \( \text{wf} : (\text{well} \text{founded} \ T \ \text{gt}) \).

Definition \( \text{top} := (A.\text{top}, B.\text{top}) \).
6 MODULE MAP: FINITE INTEGER MAPS

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

Definition ge := (refl_closure T gt).

Lemma ge_pointwise:
  (A.ge a a') → (B.ge b b') → (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 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 → 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 → (map A).
Axiom map_get: nat → (map A) → A.
Axiom map_set: nat → A → (map A) → (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 axiomatize the properties we are interested in.

Axiom map_get_init:
\[(a : A)(n : \text{nat}) \Rightarrow (\text{map_get } n (\text{map_init } a)) = a.\]

Axiom map_get_set_same:
\[(m : (\text{map } A))(n : \text{nat})(a : A) \Rightarrow (\text{map_get } n (\text{map_set } n a m)) = a.\]

Axiom map_get_set_other:
\[(m : (\text{map } A))(n, n' : \text{nat})(a : A) \Rightarrow n \neq n' \Rightarrow (\text{map_get } n' (\text{map_set } n a m)) = (\text{map_get } n' m).\]

End Map.

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

Notation "\(a.(b) \leftarrow c\)" := \((\text{map_set }? b c a)\).

Notation "\(a.(b)\)" := \((\text{map_get }? b a)\).

6.2 Ordering on maps

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

Section Map_Ordering.

Variable \(A : \text{Set}\).

Variable \(gtA : A \rightarrow A \rightarrow \text{Prop}\).

Hypothesis \(wf_{gtA} : (\text{well-founded } A \text{ } gtA)\).

\(\text{map_get_lexico } n p q\) holds if the tuple \((p.(0), \ldots, p.(n-1))\) is lexicographically greater than the tuple \((q.(0), \ldots, q.(n-1))\).

Inductive \(\text{map_get_lexico } \text{nat} \rightarrow (\text{map } A) \rightarrow (\text{map } A) \rightarrow \text{Prop} \) :=

\(\text{map_get_lexico_more}\):
\[(n : \text{nat}) \Rightarrow \lambda (p, q : (\text{map } A)).
\text{gtA } p.(n) q.(n) \Rightarrow
(\text{map_get_lexico } (S \text{ } n) p q)\]

\(\text{map_get_lexico_same}\):
\[(n : \text{nat}) \Rightarrow \lambda (p, q : (\text{map } A)).
\text{p.(n) = q.(n)} \Rightarrow
(\text{map_get_lexico } n p q) \Rightarrow
(\text{map_get_lexico } (S \text{ } n) p q)\).

\(\text{map_get_lexico } n\) is a well-founded ordering.

Remark \(\text{well-founded_map_get_lexico } \text{ind}\):
\[(n : \text{nat}) \Rightarrow
(\text{well-founded } (\text{map } A) (\text{map_get_lexico } n)) \Rightarrow
(\text{well-founded } (\text{map } A) (\text{map_get_lexico } (S \text{ } n))).\]

Lemma \(\text{well-founded_map_get_lexico}\):
(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 \to (map A) \to (map A) \to Prop :=
map_gt_intro:
\( \lambda (p,q: (map A)) \)
\( ((n:nat) (lt n range) \to (geA p.(n) q.(n))) \to \)
\( (n:nat) (lt n range) \to (gtA p.(n) q.(n)) \to \)
\( (map_gt range p q) \).

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

Lemma map_gt_inclusion:
\( \lambda (p,q: (map A)) \)
\( (map_gt range p q) \to (map_gt_lexico range p q) \).

Lemma well_founded_map_gt:
\( \lambda (range:nat) \) (well-founded (map A) (map_gt range)).

The pointwise ordering is transitive.

Hypothesis transitive_gtA: (transitive A gtA).

Lemma transitive_map_gt:
\( \lambda (range:nat) \) (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:
\( \lambda (range:nat) \) (p: (map A)) (n:nat) (a:A)
\( \lambda (lt n range) \to \)
\( (gtA a p.(n)) \to \)
\( (map_gt range (p.(n) \leftarrow a) p) \).

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

Hypothesis eqA_dec: \( (a,b: A) \{a=b\} + \{a\neq b\} \).

Lemma map_eq_dec:
\( \lambda (range:nat) \) (p,q: (map A))
\( \{ (n:nat) (lt n range) \to p.(n) = q.(n) \} \)
\( + \{ (EX n | (lt n range) \land p.(n) \neq q.(n)) \} \).

The reflexive closure of map_gt is indeed the pointwise “greater or equal” relation.

Lemma ge_map_pointwise_1:
\( \lambda (range:nat) \) (p,q: (map A))
\( (refl_closure (map A) (map_gt range) p q) \to \)
(n:nat) (lt n range) → (geA p.(n) q.(n)).

Lemma ge_map_pointwise_2:
(r:range:nat)(p,q: (map A))
((n:nat) (lt n range) → (geA p.(n) q.(n))) →
((n:nat) (lt n range) → p.(n) = q.(n))
\lor (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 ≤ i < range.

Section Map_apply.

Variable A, B: Set.
Variable f: nat → A → B.

Fixpoint map_apply_rec [p:(map A); q:(map B); n:nat] : (map B) :=
Cases n of
O ⇒ q
| (S m) ⇒ (map_apply_rec p q.(m) ← (f m p.(m))) m
end.

Definition map_apply [p:(map A); n:nat] : (map B) :=
Cases n of
O ⇒ (map_init B (f O p.(O)))
| (S m) ⇒ (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) → (map_apply_rec p q range).(n) = (f n p.(n)))
\land ((le range n) → (map_apply_rec p q range).(n) = q.(n)).

Lemma map_apply_transf:
(range:nat) (p: (map A))
(n:nat) (lt n range) → (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_n. Require Relations.
Require Le. Require Lt. Require Peano_dec.

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 \( X(s) \geq (\text{transf } n X(n)) \) 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`:

\[
(\text{last\_node}: \text{nat})
(\text{successors}: (\text{nat} \rightarrow (\text{list nat})))
(\text{successors\_in\_graph}: (\text{n},\text{s}:\text{nat})(\text{In } s \ (\text{successors } n)) \rightarrow (\text{le } s \ \text{last\_node}))
(\text{transf}: \text{nat} \rightarrow \text{L.T} \rightarrow \text{L.T})
(\text{entry\_points}: (\text{list nat} \times \text{L.T}))
(\text{map } \text{L.T}).
\]

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

Parameter `fixpoint_solution`:

\[
(\text{last\_node}: \text{nat})
(\text{successors}: \text{nat} \rightarrow (\text{list nat}))
(\text{successors\_in\_graph}: (\text{n},\text{s}:\text{nat})(\text{In } s \ (\text{successors } n)) \rightarrow (\text{le } s \ \text{last\_node}))
(\text{transf}: \text{nat} \rightarrow \text{L.T} \rightarrow \text{L.T})
(\text{entry\_points}: (\text{list nat} \times \text{L.T})){\text{\text{\text{)}}}
(\text{n},\text{s}:\text{nat})
(\text{le } n \ \text{last\_node}) \rightarrow
(\text{In } s \ (\text{successors } n)) \rightarrow
(\text{L.ge} \ (\text{fixpoint } \text{last\_node } \text{successors } \text{successors\_in\_graph } \text{transf } \text{entry\_points}).(s))
\]
(\text{transf } n \hspace{1em} (\text{fixpoint } last\_node \hspace{0.5em} \text{successors } \hspace{0.5em} \text{successors\_in\_graph } \hspace{0.5em} \text{transf } \hspace{0.5em} \text{entry\_points} \hspace{0.5em} .(n))))).

The following theorem shows that the solution returned by \text{fixpoint} satisfies the inequation \(X(n) \geq v\) for every \((\text{node}, \text{value})\) pair \((n, v)\) in \text{entry\_points}.

\textbf{Parameter} \text{fixpoint\_entry}:

\begin{align*}
&\text{(last\_node}: \text{nat}) \hspace{1em} \\
&(\text{successors}: \text{nat} \to (\text{list nat})) \hspace{1em} \\
&(\text{successors\_in\_graph} : (n,s:\text{nat})(\text{In } s \hspace{0.5em} (\text{successors } n)) \to (\text{le } s \hspace{0.5em} \text{last\_node})) \hspace{1em} \\
&(\text{transf} : \text{nat} \to L.T \to L.T) \hspace{1em} \\
&(\text{entry\_points} : (\text{list nat} \times L.T)) \hspace{1em} \\
&(\text{entry\_points\_in\_graph} : (n:\text{nat}; \ s:L.T)(\text{In } (n,s) \hspace{0.5em} \text{entry\_points}) \to (\text{le } n \hspace{0.5em} \text{last\_node})) \hspace{1em} \\
&(p:\text{nat}; \ l:L.T) \hspace{1em} \\
&(\text{In } (p,l) \hspace{0.5em} \text{entry\_points}) \to (L.g.e \hspace{0.5em} (\text{fixpoint } last\_node \hspace{0.5em} \text{successors } \hspace{0.5em} \text{successors\_in\_graph } \hspace{0.5em} \text{transf } \hspace{0.5em} \text{entry\_points}).(p) \ l). \\
\end{align*}

End \text{DATAFLOW\_SOLVER}.

### 7.2 \text{Kildall’s algorithm}

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

\textbf{Module} Dataflow\_Solver[\text{LAT}: \text{SEMI LATTICE}];

\textit{DATAFLOW\_SOLVER} with Module \text{L} := \text{LAT}.

Module \text{L} := \text{LAT}.

\textbf{Section} \text{Kildall}.

Parameters defining the flow graph.

\textbf{Variable} last\_node: \text{nat}.

\textbf{Variable} successors: \text{nat} \to (\text{list nat}).

\textbf{Hypothesis} successors\_in\_graph:

\((n:\text{nat}) \hspace{0.5em} (s:\text{nat}) \hspace{0.5em} (\text{In } s \hspace{0.5em} (\text{successors } n)) \to (\text{le } s \hspace{0.5em} \text{last\_node})\).

Parameters defining the analysis.

\textbf{Variable} transf: \text{nat} \to L.T \to L.T.

\textbf{Variable} entry\_points: (\text{list } \text{(nat} \times L.T)).

\textbf{Hypothesis} entry\_points\_in\_graph:

\((n:\text{nat}) \hspace{0.5em} (s:L.T) \hspace{0.5em} (\text{In } (n,s) \hspace{0.5em} \text{entry\_points}) \to (\text{le } n \hspace{0.5em} \text{last\_node})\).

### 7.2.1 Definition of the algorithm

Here is the algorithm in pseudocode:

\begin{verbatim}
in <- map_init bottom wrk <- [0, 1, ..., last_node]...
\end{verbatim}
foreach \((pt, st)\) in entry_points do 
  \(\text{in} \leftarrow (\text{in.}(pt) \leftarrow (\text{lub} \text{in.}(pt) \text{st}))\)
done 
while \(\text{wrk}\) not empty do 
  let \(n = \text{head} \text{wrk}\) 
  \(\text{wrk} \leftarrow \text{tail} \text{wrk}\) 
  let \(\text{out} = (\text{transf} n \text{in.}(n))\) 
  foreach \(s\) in \((\text{successors} n)\) do 
    let \(i = \text{lub} \text{in.}(s) \text{out}\) 
    if \(i \neq \text{in.}(s)\) then 
      \(\text{in} \leftarrow (\text{in.}(s) \leftarrow i);\) 
      \(\text{wrk} \leftarrow s :: \text{wrk}\) 
    endif 
  endfor 
endwhile 
return \(\text{in}\)

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

**Definition** state : Set := \((\text{map} \ L.T) \times (\text{list} \ nat)\). 

Definition of the start state for the fixpoint computation.

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

**Fixpoint** start_state_wrk \([n: \text{nat}]\) : \((\text{list} \ nat)\) := 
  Cases \(n\) of 
  \(O \Rightarrow (\text{cons} \ O \ \text{nil} \ \text{nat})\) 
  | \((S \ m) \Rightarrow (\text{cons} \ (S \ m) \ \text{start_state_wrk} \ m)\) 
  end.

**Definition** start_state : state := 
\((\text{start_state_in} \text{entry_points}, \text{start_state_wrk} \ \text{last_node})\). 

Definition of the “foreach \(s\) in \((\text{successors} n)\)” part of the algorithm.

**Definition** propagate_succ \([ks: \text{state}; \text{out}:\text{L.T}; n: \text{nat}]\) := 
  let oldl = (\text{Fst} \ ks).(n) \text{in} 
  let newl = (\text{L.lub} \ \text{oldl} \ \text{out}) \text{in} 
  Cases \(\text{L.eq_dec} \ \text{oldl} \ \text{newl}\) of 
  \(\text{left} \_ \Rightarrow \text{ks}\) 
  | \(\text{right} \_ \Rightarrow (((\text{Fst} \ ks).(n) \leftarrow \text{newl}), (\text{cons} \ n \ (\text{Snd} \ ks)))\) 
  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)) →
Lemma \textit{step\_increases}:

\begin{align*}
  \text{let } \text{ks}' = \text{propagate\_succ\_list } \text{ks} \text{ out } \text{nl} \text{ in} \\
  \text{(} \text{L, ge } \text{Fst ks'}.(s) \text{ out }) \\
  \land (\text{s: nat}) \\
  ((\text{In s succs}) \rightarrow (\text{L, ge } \text{Fst ks'}.(s) \text{ out })) \\
  \land (\neg (\text{In s succs}) \rightarrow (\text{Fst ks'}.(s) = (\text{Fst ks}).(s))).
\end{align*}

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

Definition \textit{while} :=

\begin{align*}
  \text{step step} \\
  \text{gt\_state wf\_gt\_state} \\
  \text{step\_increases} \\
  \text{start\_state}).
\end{align*}

Definition \textit{fixpoint} := (Fst while).

\subsection{Correctness invariant}

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

Definition \textit{good\_state} \([\text{ks: state}] :=

\begin{align*}
  \text{(n: nat)} \\
  (\text{le n last\_node}) \\
  \neg (\text{In n (Snd ks)}) \\
  (\text{s: nat}) \\
  (\text{In s (successors n)}) \\
  (\text{L, ge } (\text{Fst ks}).(s) \text{ out } (\text{transf n (Fst ks)}.(n))).
\end{align*}

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

Lemma \textit{start\_state\_good}:

\begin{align*}
  \text{(good\_state start\_state).}
\end{align*}

Lemma \textit{propagate\_good\_state}:

\begin{align*}
  \text{(ks: state) (out: L.T) (n: nat)} \\
  \text{let } \text{ks}' = \text{propagate\_succ ks out n} \text{ in} \\
  (\text{L, ge } (\text{Fst ks'}).(n) \text{ out}) \\
  \land ((\text{s: nat}) \text{ n } \neq \text{ s } \rightarrow ((\text{Fst ks'}).(s)) = ((\text{Fst ks}).(s))).
\end{align*}

Lemma \textit{propagate\_list\_good\_state}:

\begin{align*}
  \text{(succs: list nat) (ks: state) (out: L.T)} \\
  \text{let } \text{ks}' = \text{propagate\_succ\_list ks out succs} \text{ in} \\
  (\text{s: nat}) \\
  ((\text{In s succs}) \rightarrow (\text{L, ge } (\text{Fst ks'}.(s) \text{ out })) \\
  \land (\neg (\text{In s succs}) \rightarrow (\text{Fst ks'}.(s) = (\text{Fst ks}).(s))).
\end{align*}
Lemma \texttt{propagate\_increasing}:
\[
(ks: \text{state}) (out:L.T) (n: \text{nat})
\text{let } ks' = (\text{propagate\_succ } ks \text{ out } n) \text{ in}
(s:\text{nat}) (\text{In } s \ (\text{Snd } ks)) \rightarrow (\text{In } s \ (\text{Snd } ks')).
\]

Lemma \texttt{propagate\_list\_increasing}:
\[
(\text{succs}: \text{(list nat)}) (ks: \text{state}) (out:L.T)
\text{let } ks' = (\text{propagate\_succe\_list } ks \text{ out } \text{succs}) \text{ in}
(s:\text{nat}) (\text{In } s \ (\text{Snd } ks')) \rightarrow (\text{In } s \ (\text{Snd } ks')).
\]

Lemma \texttt{propagate\_records\_changes}:
\[
(ks: \text{state}) (out:L.T) (n: \text{nat})
\text{let } ks' = (\text{propagate\_succ } ks \text{ out } n) \text{ in}
(s:\text{nat}) (\text{In } s \ (\text{Snd } ks')) \lor (\text{Fst } ks').(s) = (\text{Fst } ks).(s).
\]

Lemma \texttt{propagate\_list\_records\_changes}:
\[
(\text{succs}: \text{(list nat)}) (ks: \text{state}) (out:L.T)
\text{let } ks' = (\text{propagate\_succe\_list } ks \text{ out } \text{succs}) \text{ in}
(s:\text{nat}) (\text{In } s \ (\text{Snd } ks')) \lor (\text{Fst } ks').(s) = (\text{Fst } ks).(s).
\]

Lemma \texttt{propagate\_ge}:
\[
(ks: \text{state}) (out:L.T) (n: \text{nat})
\text{let } ks' = (\text{propagate\_succ } ks \text{ out } n) \text{ in}
(s:\text{nat}) (L.g e (\text{Fst } ks').(s) (\text{Fst } ks).(s)).
\]

Lemma \texttt{propagate\_list\_ge}:
\[
(\text{succs}: \text{(list nat)}) (ks: \text{state}) (out:L.T)
\text{let } ks' = (\text{propagate\_succe\_list } ks \text{ out } \text{succs}) \text{ in}
(s:\text{nat}) (L.g e (\text{Fst } ks').(s) (\text{Fst } ks).(s)).
\]

Lemma \texttt{step\_state\_good}:
\[
(ks: \text{state})
\text{stop } ks = \text{false } \rightarrow
\text{good\_state } ks \rightarrow
\text{good\_state } (\text{step } ks).
\]

\textbf{7.2.3 Correctness theorems.}

Using the invariant on intermediate states, we prove that the result of the function \texttt{fixpoint} indeed satisfies the dataflow inequations \texttt{fixpoint}.(s) \geq (\text{transf } n \texttt{fixpoint}.(n)) for every edge \(n \rightarrow s\) in the flow graph.

Theorem \texttt{fixpoint\_solution}:
\[
(n: \text{nat})(s: \text{nat})
\text{le } n \text{ last\_node } \rightarrow
\text{In } s \ (\text{successors } n) \rightarrow
(L.g e \texttt{fixpoint}.(s) (\text{transf } n \texttt{fixpoint}.(n))).
\]

Moreover, for each \((n,v)\) in \texttt{entry\_points}, we have that \texttt{fixpoint}.(n) \geq 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 $X(s) \geq (\text{transf } n \ X(n))$ for every edge $n \rightarrow s$. We now develop a solver for backward dataflow problems, of the form $X(n) \geq (\text{transf } s \ X(s))$ for every edge $n \rightarrow s$. This “backward” solver is derived from the “forward” solver by inverting the flow graph, replacing the successor edges $n \rightarrow s$ by predecessor edges $s \rightarrow n$.

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
\[(n, s: \text{nat}) \quad (\text{In } n \text{ pred}'(s)) \leftrightarrow (\text{In } n \text{ pred}(s)) \lor (n = \text{from} \land (\text{In } s \text{ tolist})).\]

**Fixpoint** `make_predecessors_aux [pred: (map (list nat)); pc: \text{nat}]: (map (list nat)) :=`

```plaintext
Cases pc of
  O \Rightarrow \text{pred}
| (S n) \Rightarrow
  (make_predecessors_aux (add_successors pred n (successors n)) n)
end.
```

**Lemma** `make_predecessors_aux_correct`:

```plaintext
(let pred' = (make_predecessors_aux pred pc) in
 (n, s: \text{nat}) \quad (\text{In } n \text{ pred}'(s)) \leftrightarrow (\text{In } n \text{ pred}(s)) \lor ((\text{lt } n \text{ pc}) \land (\text{In } s \text{ (successors n))}).
```

**Definition** `predecessors : \text{nat} \rightarrow (\text{list nat}) :=`

```plaintext
let pred = (make_predecessors_aux (map_init (list nat) (nil nat)) (S last_node)) in
[n: \text{nat}] \text{pred}(n).
```

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

**Lemma** `predecessors_correct`:

```plaintext
(n, s: \text{nat}) \quad (\text{le } n \text{ last_node}) \Rightarrow
(\text{In } s \text{ (successors } n)) \leftrightarrow (\text{In } n \text{ (predecessors } s)).
```

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

**Lemma** `predecessors_in_graph`:

```plaintext
(s: \text{nat}) \quad (n: \text{nat}) \quad (\text{In } n \text{ (predecessors } s)) \Rightarrow (\text{le } n \text{ last_node}).
```

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

**Definition** `fixpoint :=`

```plaintext
(\text{DS} \text{.fixpoint } \text{last_node} \text{ predecessors predecessors_in_graph transf entry_points}).
```

**Theorem** `fixpoint_solution`:

```plaintext
(n, s: \text{nat}) \quad (\text{le } n \text{ last_node}) \Rightarrow
(\text{In } s \text{ (successors } n)) \Rightarrow
(\text{L} \ge \text{fixpoint}.(n)
  (\text{transf } s \text{ fixpoint}.(s))).
```

**Theorem** `fixpoint_entry`:

```plaintext
(p: \text{nat}; l: \text{L}T)
(\text{In } (p, l) \text{ entry_points}) \Rightarrow
(\text{L} \ge \text{fixpoint}.(p) \ l).
```
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 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 $\text{max\_num\_regs}$. 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 $\text{max\_num\_regs}: \text{nat}$.

Thus, a (pseudo-) register is an integer between 0 (included) and $\text{max\_num\_regs}$ (excluded). We represent them as a dependent record of an integer and a proof that this integer is less than $\text{max\_num\_regs}$.

Record $\text{reg}: \text{Set} := \text{make\_reg} \{ \text{reg\_no}: \text{nat}; \text{reg\_lt}: (\text{lt reg\_no max\_num\_regs}) \}$.

For some proofs, we need that the type $\text{reg}$ is inhabited. Hence, we require that $\text{max\_num\_regs}$ is not 0, and define register number 0.

Axiom $\text{regs\_not\_empty}: \text{max\_num\_regs} \neq 0$.

Definition $\text{some\_reg} :=$

$(\text{make\_reg} \ 0$

$(\text{req\_O\_lt max\_num\_regs} \ (\text{sym\_not\_eq nat max\_num\_regs} \ O \ \text{regs\_not\_empty})))$.

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 $\text{max\_num\_regs}$ differ.

Axiom $\text{reg\_eq}: (r1, r2: \text{reg}) \ (\text{reg\_no r1}) = (\text{reg\_no r2}) \rightarrow r1 = r2$.

Consequently, register equality is decidable.

Lemma $\text{reg\_not\_eq}: (r1, r2: \text{reg}) \ r1 \neq r2 \rightarrow (\text{reg\_no r1}) \neq (\text{reg\_no r2})$.

Lemma $\text{reg\_eq\_dec}: (r1, r2: \text{reg}) \{ r1 = r2 \} + \{ r1 \neq r2 \}$. 
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 \texttt{Map}, and actually are built on top of them, but have additional properties due to the finiteness of the type \texttt{reg}.

\textbf{Definition} \texttt{regmap} \( [A{:} Set] := (\text{map} \; A) \).

\textbf{Section} \texttt{RegMap}.

\textbf{Variable} \( A{:} Set \).

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

\textbf{Definition} \texttt{regmap_init} \( : \; A \to (\text{regmap} \; A) := (\text{map_init} \; A) \).

\textbf{Definition} \texttt{regmap_get} \( : \; \texttt{reg} \to (\text{regmap} \; A) \to A := [r{:}\text{reg}; m{:}(\text{regmap} \; A)] (\text{map_get} \; A (\text{reg_no} \; r) \; m) \).

\textbf{Definition} \texttt{regmap_set} \( : \; \texttt{reg} \to A \to (\text{regmap} \; A) \to (\text{regmap} \; A) := [r{:}\text{reg}; v{:}A; m{:}(\text{regmap} \; A)] (\text{map_set} \; A (\text{reg_no} \; r) \; v \; m) \).

\textbf{Lemma} \texttt{regmap_get_init}:
\( (a{:}A)(r{:}\text{reg}) (\text{regmap_get} \; r \; (\text{regmap_init} \; a)) = a \).

\textbf{Lemma} \texttt{regmap_get_set_same}:
\( (m{:}(\text{regmap} \; A))(r{:}\text{reg})(a{:}A) (\text{regmap_get} \; r \; (\text{regmap_set} \; r \; a \; m)) = a \).

\textbf{Lemma} \texttt{regmap_get_set_other}:
\( (m{:}(\text{regmap} \; A))(r{:}\text{reg}; r’{:}\text{reg})(a{:}A)
\quad \forall r \neq r’ \to (\text{regmap_get} \; r’ \; (\text{regmap_set} \; r \; a \; m)) = (\text{regmap_get} \; r’ \; m) \).

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

\textbf{Axiom} \texttt{regmap_extensional}:
\( (m_1{:}m_2{:} (\text{regmap} \; A))
\quad (\forall r{:}\text{reg} (\text{regmap_get} \; r \; m_1) = (\text{regmap_get} \; r \; m_2)) \rightarrow m_1 = m_2 \).

\textbf{Lemma} \texttt{regmap_set_invariant}:
\( (m{:}(\text{regmap} \; A))(r{:}\text{reg})(a{:}A)
\quad (\text{regmap_get} \; r \; m) = a \rightarrow (\text{regmap_set} \; r \; a \; m) = m \).

\textbf{End} \texttt{RegMap}.

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

\textbf{Notation} “\( a \;.[ \; b \; ] \leftarrow c \)” := (\text{regmap_set} \; ? \; b \; c \; a).

\textbf{Notation} “\( a \;.[ \; b \; ]” := (\text{regmap_get} \; ? \; b \; a).
8.3 Register maps as a semi-lattice

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

Module Regmap [A: SEMILATTICE].

Definition $T := (\text{regmap } A.T)$.

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

Lemma $\text{wf}$: (well-founded $T gt$).

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

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

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

Definition $\text{top} := (\text{regmap\_init } A.T A.\text{top})$.

Definition $\text{bot} := (\text{regmap\_init } A.T A.\text{bot})$.

Lemma $\text{ge\_top\_x}$: $(x: T) (\text{ge } top x)$.

Lemma $\text{ge\_x\_bot}$: $(x: T) (\text{ge } x \ bot)$.

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

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

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

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

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

Lemma $\text{ge\_lub\_left}$:
$(p, q: T) (\text{ge } (\text{lub } p \ q) \ p)$.

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** \( \text{value} := \text{nat} \).

**Parameter** \( \text{bad_value} : \text{value} \).

**Parameter** \( \text{mach_add} : \text{value} \rightarrow \text{value} \rightarrow \text{value} \).

**Parameter** \( \text{mach_sub} : \text{value} \rightarrow \text{value} \rightarrow \text{value} \).

**Parameter** \( \text{mach_equal} : \text{value} \rightarrow \text{value} \rightarrow \text{bool} \).

**Parameter** \( \text{mach_notequal} : \text{value} \rightarrow \text{value} \rightarrow \text{bool} \).

**Parameter** \( \text{mach_lessequal} : \text{value} \rightarrow \text{value} \rightarrow \text{bool} \).

**Parameter** \( \text{mach_greaterthan} : \text{value} \rightarrow \text{value} \rightarrow \text{bool} \).

**Parameter** \( \text{mach_greaterequal} : \text{value} \rightarrow \text{value} \rightarrow \text{bool} \).

**Axiom** \( \text{mach_add_commut} : (x,y: \text{value}) \ (\text{mach_add } x \ y) = (\text{mach_add } y \ x) \).

**Axiom** \( \text{mach_add_x_O} : (x: \text{value}) \ (\text{mach_add } x \ O) = x \).

**Axiom** \( \text{mach_add_O_x} : (x: \text{value}) \ (\text{mach_add } O \ x) = x \).

**Axiom** \( \text{mach_sub_x_O} : (x: \text{value}) \ (\text{mach_sub } x \ O) = x \).

### 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.


#### 10.1 Instruction set, functions, and programs.

**Inductive** \( \text{operation} : \text{Set} := \)

- \( \text{Omove} : \text{operation} \)
- \( \text{Oconst} : \text{value} \rightarrow \text{operation} \)
- \( \text{Oneg} : \text{operation} \)
- \( \text{Oadd} : \text{operation} \)
- \( \text{Osub} : \text{operation} \)
- \( \text{Oaddimm} : \text{value} \rightarrow \text{operation} \)
- \( \text{Osubimm} : \text{value} \rightarrow \text{operation} \)

**Inductive** \( \text{addressing_mode} : \text{Set} := \)

- \( \text{Aindexed} : \text{value} \rightarrow \text{addressing_mode} \)
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:

- **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 \( \rightarrow \) (list reg) \( \rightarrow \) reg \( \rightarrow \) instruction
| Iload: addressing_mode \( \rightarrow \) (list reg) \( \rightarrow \) reg \( \rightarrow \) instruction
| Istore: addressing_mode \( \rightarrow \) (list reg) \( \rightarrow \) reg \( \rightarrow \) instruction
| Icall: reg \( \rightarrow \) (list reg) \( \rightarrow \) reg \( \rightarrow \) instruction
| Ibranch: nat \( \rightarrow \) instruction
| Icondbranch: condition \( \rightarrow \) (list reg) \( \rightarrow \) nat \( \rightarrow \) instruction
| Ireturn: reg \( \rightarrow \) 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 $I\text{return}$ or a $I\text{goto}$.

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** $\text{instr\_valid} [i: \text{instruction}; \text{last\_pc}: \text{nat}] :=$

```plaintext
Cases i of
  (Ibranch n) ⇒ (le n last\_pc)
| (Icondbranch cond regs n) ⇒ (le n last\_pc)
| _ ⇒ True
end.
```

**Definition** $\text{instr\_terminal} [i: \text{instruction}] :=$

```plaintext
Cases i of
  (Ibranch _) ⇒ True
| (Ireturn _) ⇒ True
| _ ⇒ False
end.
```

**Record** $\text{function} : \text{Set} := \text{make}\_\text{function}

{ $\text{fn\_instr}: (\text{map} \ \text{instruction});$
  $\text{fn\_params}: (\text{list} \ \text{reg});$
  $\text{fn\_last\_pc}: \text{nat};$
  $\text{fn\_valid}:$
    $(\text{pc}:\text{nat}) (le \text{pc fn\_last\_pc}) \to (\text{instr\_valid fn\_instr.(pc)} \text{ fn\_last\_pc});$
  $\text{fn\_last\_instr\_terminal}:$
    $(\text{instr\_terminal fn\_instr.(fn\_last\_pc))} \}.$

A program is a collection of functions, identified by an integer between 0 and $\text{prog\_num\_functions}$ (excluded), plus a distinguished function that acts as the entry point (the “main function”) of the program.

**Record** $\text{program} : \text{Set} := \text{make}\_\text{program}

{ $\text{prog\_function}: (\text{map} \ \text{function});$
  $\text{prog\_num\_functions}: \text{nat};$
  $\text{prog\_entrypoint}: \text{nat};$
  $\text{prog\_entrypoint\_valid}: (lt \text{prog\_entrypoint prog\_num\_functions}) \}.$

### 10.2 Dynamic semantics

**Section** Dynamic semantics.

**Variable** $\text{prog}: \text{program}.$

The type $\text{location}$ represents memory locations. The type $\text{regset}$ represent register sets, i.e. mappings from registers to values. The type $\text{store}$ represent memory stores, i.e. mappings from locations to values.

**Definition** $\text{location} := \text{nat}.$
Definition \( \text{regset} := (\text{regmap} \text{ value}) \).
Definition \( \text{store} := (\text{map} \text{ value}) \).

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 \( \text{Mach\_ops} \).

Definition \( \text{eval\_operation} [\text{op:operation}; \text{args:}\text{(list value)}] := \)
\[
\text{Cases} (\text{op}, \text{args}) \text{ of}
\]
\[
(\text{Omove}, (\text{cons} x \text{ nil})) \Rightarrow x
\]
\[
|(\text{Oconst} n), \text{nil}) \Rightarrow n
\]
\[
|(\text{Oneg}), (\text{cons} x \text{ nil})) \Rightarrow (\text{mach\_add} x \text{ y})
\]
\[
|(\text{Osub}, (\text{cons} x (\text{cons} y \text{ nil}))), (\text{mach\_sub} x \text{ y})
\]
\[
|(\text{Oaddimm} n), (\text{cons} x \text{ nil})) \Rightarrow (\text{mach\_add} x \text{ n})
\]
\[
|(\text{Subimm} n), (\text{cons} x \text{ nil})) \Rightarrow (\text{mach\_sub} x \text{ n})
\]
\[
| \_ \Rightarrow \text{bad\_value}
\]
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 \( \text{eval\_addressing} [\text{mode:addressing\_mode}; \text{args:}\text{(list value)}] := \)
\[
\text{Cases} (\text{mode}, \text{args}) \text{ of}
\]
\[
((\text{Aindexed} n), (\text{cons} x \text{ nil})) \Rightarrow (\text{mach\_add} x \text{ n})
\]
\[
| \_ \Rightarrow \text{bad\_value}
\]
end.

Evaluation of a boolean condition.

Parameter \( \text{bad\_bool}: \text{bool} \).

Definition \( \text{eval\_condition} [\text{cond:condition}; \text{args:}\text{(list value)}] := \)
\[
\text{Cases} (\text{cond}, \text{args}) \text{ of}
\]
\[
(\text{Cequal}, (\text{cons} x (\text{cons} y \text{ nil}))) \Rightarrow (\text{mach\_equal} x \text{ y})
\]
\[
|(\text{Cnotequal}, (\text{cons} x (\text{cons} y \text{ nil}))) \Rightarrow (\text{mach\_notequal} x \text{ y})
\]
\[
|(\text{Clessthan}, (\text{cons} x (\text{cons} y \text{ nil}))) \Rightarrow (\text{mach\_lessthan} x \text{ y})
\]
\[
|(\text{Clessequal}, (\text{cons} x (\text{cons} y \text{ nil}))) \Rightarrow (\text{mach\_lessequal} x \text{ y})
\]
\[
|(\text{Cgreaterthan}, (\text{cons} x (\text{cons} y \text{ nil}))) \Rightarrow (\text{mach\_greaterthan} x \text{ y})
\]
\[
|(\text{Cgreaterequal}, (\text{cons} x (\text{cons} y \text{ nil}))) \Rightarrow (\text{mach\_greaterequal} x \text{ y})
\]
\[
| \_ \Rightarrow \text{bad\_bool}
\]
end.

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

Definition \( \text{eval\_reg} [\text{rs: regset}; \text{r: reg}] := \text{rs}[\text{r}] \).
Definition \( \text{eval\_regs} [\text{rs: regset}; \text{rl:}\text{(list reg)}] := \)
\[
(\text{PolyList}\text{.map} (\text{eval\_reg} \text{rs}) \text{rl})
\]
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:
\((\text{exec instr } f \text{ pc } rs \text{ st } pc' \text{ rs' st'})\) 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 \(\text{exec instr}\) step can correspond to a whole function application.

To deal with \(Icall\) instructions, another predicate is defined in a mutually-recursive manner:
\((\text{exec function } f \text{ pc } rs \text{ st } retval \text{ st'})\) 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:

\[\begin{align*}
\text{function } &\to \text{nat }\to \text{regset }\to \text{store }\to \text{nat }\to \text{regset }\to \text{store }\to \text{Prop} := \\
\text{exec nop:} &\quad (f: \text{function}) (pc: \text{nat}) (rs: \text{regset}) (st: \text{store}) \\
&\quad (\text{fn_instr } f).(pc) = \text{Inop} \to \\
&\quad (\text{exec instr } f \text{ pc } rs \text{ st } (\text{S pc) rs st})
\end{align*}\]

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

\[\begin{align*}
\text{exec op:} &\quad (f: \text{function}) (pc: \text{nat}) (rs: \text{regset}) (st: \text{store}) \\
&\quad (\text{op: operation}) (\text{srcs: (list reg)}) (\text{dst: reg}) \\
&\quad (\text{fn_instr } f).(pc) = (\text{Iop op srcs dst}) \to \\
&\quad (\text{exec instr } f \text{ pc } rs \text{ st } \\
&\quad \text{(S pc) (rs.[dst] ← (eval_operation op (eval_regs rs srcs))) st})
\end{align*}\]

The \(Iload\) operation evaluates the location to be read using the \(\text{eval_addressing}\) function. It then reads the corresponding store location, and store its value in the result register. The store is unchanged.

\[\begin{align*}
\text{exec load:}
\end{align*}\]
The \textit{Iload} operation evaluates the location to be modified using the \textit{eval_addressing} function. It then updates this store location with the value of the \textit{src} register. The register set is unchanged.

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

The \textit{Ibranch} instruction sets the \textit{pc} component to the destination of the branch. The register set and the store are unaffected.

The \textit{Icondbranch} instruction evaluates its boolean condition. If it evaluates to \textit{true}, \textit{pc} is set to the destination of the branch. If it evaluates to \textit{false}, \textit{pc} advances to the next instruction. The register set and the store are unaffected.
(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: \textit{successors }f \ pc\textit{ }is the list of all PCs that can be reached by executing the instruction at pc in function f. Thus, \textit{successors }f\textit{ }defines the edges of the control-flow graph for function f, as required for data flow analysis (see module Kildall).

\textbf{Section Successors.}

\textbf{Variable }f:\ function.

\textbf{Definition }successors \[pc\] :=
\[
\text{Cases } (\text{le lt dec } pc (\text{fn_last_pc } f)) \text{ of }
\]
\[
(\text{left } \_ ) \Rightarrow
\]
\[
\text{Cases } (\text{fn_instr } f). (pc) \text{ of }
\]
\[
(Ibranch dst) \Rightarrow (\text{cons } dst (\text{nil } \text{nat} ))
\]
\[
| (Icondbranch cond args dst) \Rightarrow (\text{cons } dst (\text{cons } (S \text{ pc}) (\text{nil } \text{nat} )))
\]
\[
| (Ireturn arg) \Rightarrow (\text{nil } \text{nat})
\]
\[
| \_ \Rightarrow (\text{cons } (S \text{ pc}) (\text{nil } \text{nat}))
\]
\[
\text{end}
\]
\[
| (\text{right } \_ ) \Rightarrow (\text{nil } \text{nat})
\]
\[
\text{end}.
\]

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 \text{fn_last_pc } f.

\textbf{Lemma successors_in_graph:}
\[
(n, s: \text{nat}) (\text{In } s (\text{successors } n)) \rightarrow (\text{le } s (\text{fn_last_pc } f)).
\]

\textbf{End Successors.}

The \textit{successors }function is correct in the following sense: if the execution of an instruction at position \text{pc} takes us to position \text{pc}', then \text{pc}' belongs to \textit{successors }f \text{ pc}.

\textbf{Lemma successors_correct:}
\[
(p: \text{program})(f: \text{function})
\]
\[
(pc: \text{nat})(rs: \text{regset})(st: \text{store})(pc': \text{nat})(rs':\text{regset})(st':\text{store})
\]
\[
(\text{le } pc (\text{fn_last_pc } f)) \rightarrow
\]
\[
(\text{exec_instr } p f pc rs st pc' rs' st') \rightarrow
\]
\[
(\text{In } pc' (\text{successors } f \text{ pc})).
\]

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

\textbf{Lemma exec_instr_in_code:}
\[
(p: \text{program})(f: \text{function})
\]
\[
(pc: \text{nat})(rs: \text{regset})(st: \text{store})(pc': \text{nat})(rs':\text{regset})(st':\text{store})
\]
\[
(\text{le } pc (\text{fn_last_pc } f)) \rightarrow
\]
\[
(\text{exec_instr } p f pc rs st pc' rs' st') \rightarrow
\]
\[
(\text{le } pc' (\text{fn_last_pc } f)).
\]
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] :=
The following lemma characterizes the instructions of the transformed program.

**Lemma transformed_function:**

\[
(p: \text{program}) \ (nm: \text{nat})
\]\(\rightarrow\)
\[
(lt nm (\text{prog\_num\_functions}\ p)) \rightarrow
\]
\[
(\text{prog\_function}\ (\text{transf\_program}\ p))(nm) =
\]
\[
(\text{transf\_function}\ nm\ (\text{prog\_function}\ p)(nm)).
\]

End Transform\_program.

**11 Module Constprop: constant propagation**

The module Constprop formalizes constant propagation.

Require PolyList. Require Peano\_dec.

**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 \(T := \text{value}\).

Definition \(eq\_dec := eq\_nat\_dec\).

End Value.

Module \(A := \text{(Flat Value)}\).

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 \(D := \text{(Regmap A)}\).
Given a list of registers and an abstract register set \( rs \), \( \text{abstr\_regs} \) 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 \( \text{None} \).

**Fixpoint abstr\_regs** \([rs: D.T;\ args: (\text{list} \ reg)] : (\text{option} \ (\text{list} \ value)) :=

\[
\text{Cases args of}
\]

\[
\text{nil} \Rightarrow (\text{Some} \ (\text{list} \ value) \ (\text{nil} \ value))
\]

\[
| (\text{cons} \ \text{hd} \ tl) \Rightarrow
\]

\[
\text{Cases rs.}[[\text{hd}]] \ of
\]

\[
A.\text{Bot} \Rightarrow (\text{None} \ (\text{list} \ value))
\]

\[
| A.\text{Top} \Rightarrow (\text{None} \ (\text{list} \ value))
\]

\[
| (A.\text{Inj} \ v) \Rightarrow
\]

\[
\text{Cases (abstr\_regs rs tl) of}
\]

\[
\text{None} \Rightarrow (\text{None} \ (\text{list} \ value))
\]

\[
| (\text{Some} \ vl) \Rightarrow (\text{Some} \ (\text{list} \ value) \ (\text{cons} \ v \ vl))
\]

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, \( \text{transfer} \) 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 \( A.\text{Inj} \ v \).

- If the instruction is an operation with unknown arguments, or a load, or a call, set the result register to \( A.\text{Top} \) to reflect the fact that the result value is statically unpredictable.

- Otherwise, leave the abstract register set unchanged.

**Definition transfer** \([f:\text{function}; \ pc:\text{nat}; \ rs: D.T] :=

\[
\text{Cases (fn\_instr f).}(pc) \ of
\]

\[
\text{Inop} \Rightarrow
\]

\[
rs
\]

\[
| (Iop \ op \ args \ res) \Rightarrow
\]

\[
\text{Cases (abstr\_regs rs args) of}
\]

\[
(\text{Some} \ vl) \Rightarrow rs.[[res]] \leftarrow (A.\text{inj} \ (\text{eval\_operation} \ op \ vl))
\]

\[
| \text{None} \Rightarrow rs.[[res]] \leftarrow A.\text{top}
\]

end

\[
| (Iload \ mode \ args \ res) \Rightarrow
\]

\[
rs.[[res]] \leftarrow A.\text{top}
\]

\[
| (Istore \ mode \ args \ src) \Rightarrow
\]

\[
rs
\]

\[
| (Icall \ fn \ args \ res) \Rightarrow
\]

end
\[ rs \left[ res \right] \leftarrow A.top \]
\[ | \text{(Ibranch} \text{ dst}) \Rightarrow \]
\[ rs \]
\[ | \text{(Icondbranch} \text{ cond} \text{ args} \text{ dst}) \Rightarrow \]
\[ rs \]
\[ | \text{(Ireturn} \text{ arg}) \Rightarrow \]
\[ rs \]
\[ \text{end}. \]

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

\textbf{Module Solver := (Dataflow Solver D).}

\textbf{Definition analyze_function [f: function] :=}
\[ (\text{Solver}. \text{fixpoint} \]
\[ (\text{fn}\_\text{last}\_\text{pc} \ f) \]
\[ (\text{successors} \ f) \]
\[ (\text{successors}_\text{in}_\text{graph} \ f) \]
\[ (\text{transfer} \ f) \]
\[ (\text{cons} (O, D.top) \text{ Nil})). \]

\section{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.

\textbf{Definition value_match_approx [a: A.T; v: value] :=}
\[ \text{Cases } a \text{ of} \]
\[ A.Top \Rightarrow \text{True} \]
\[ | (A.Inj v') \Rightarrow v = v' \]
\[ | A.Bot \Rightarrow \text{False} \]
\[ \text{end}. \]

\textbf{Definition regset_match_approx [a: D.T; rs: regset] :=}
\[ (r: \text{reg}) \ (\text{value_match_approx} a \ [r] \ rs \ [r]). \]

Some easy properties of these “match approximation” relations follow.

\textbf{Lemma value_match_approx_increasing:}
\[ (a,b: A.T)(v: value) \]
\[ (A.ge a \ b) \rightarrow (\text{value_match_approx} b \ v) \rightarrow (\text{value_match_approx} a \ v). \]

\textbf{Lemma regset_match_approx_increasing:}
\[ (a,b: D.T)(rs: \text{regset}) \]
\[(D.\geq a \ b) \rightarrow (\text{regset}\_\text{match}\_\text{approx} \ b \ \text{rs}) \rightarrow (\text{regset}\_\text{match}\_\text{approx} \ a \ \text{rs})\].

**Lemma** \text{regset}\_\text{match}\_\text{approx}\_\text{update}:

\[
\begin{align*}
    \text{ra}: D.T) (\text{rs}: \text{regset}) (a: A.T) (v: \text{value}) (r: \text{reg}) \\
    (\text{value}\_\text{match}\_\text{approx} \ a \ v) \rightarrow \\
    (\text{regset}\_\text{match}\_\text{approx} \ \text{ra} \ \text{rs}) \rightarrow \\
    (\text{regset}\_\text{match}\_\text{approx} \ (\text{ra}\.r \leftarrow a) \ (\text{rs}\.r \leftarrow v))
\end{align*}
\]

**Lemma** \text{abstr}\_\text{regs}\_\text{match}\_\text{approx}:

\[
\begin{align*}
    \text{ra}: D.T) (\text{rs}: \text{regset}) (\text{regset}\_\text{match}\_\text{approx} \ \text{ra} \ \text{rs}) \rightarrow \\
    \text{(args}: (\text{list} \ \text{reg})) (\text{vl}: (\text{list} \ \text{value})) \\
    (\text{abstr}\_\text{regs} \ \text{ra} \ \text{args}) = (\text{Some} (\text{list} \ \text{value}) \ \text{vl}) \rightarrow \\
    (\text{eval}\_\text{regs} \ \text{rs} \ \text{args}) = \text{vl}
\end{align*}
\]

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

**Lemma** \text{transfer}\_\text{correct}:

\[
\begin{align*}
    \text{p}: \text{program}) (f: \text{function}) \\
    (\text{pc}: \text{nat}) (\text{rs}: \text{regset}) (\text{st}: \text{store}) (\text{pc}': \text{nat}) (\text{rs}': \text{regset}) (\text{st}': \text{store}) \\
    (\text{exec}\_\text{instr} \ p \ f \ \text{pc} \ \text{rs} \ \text{st} \ \text{pc}' \ \text{rs}' \ \text{st}') \rightarrow \\
    (a: D.T) \\
    (\text{regset}\_\text{match}\_\text{approx} \ a \ \text{rs}) \rightarrow \\
    (\text{regset}\_\text{match}\_\text{approx} \ (\text{transfer} \ f \ \text{pc} \ a) \ \text{rs}')
\end{align*}
\]

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** \text{analysis}\_\text{correct}:

\[
\begin{align*}
    \text{p}: \text{program}) (f: \text{function}) \\
    (\text{pc}: \text{nat}) (\text{rs}: \text{regset}) (\text{st}: \text{store}) (\text{pc}': \text{nat}) (\text{rs}': \text{regset}) (\text{st}': \text{store}) \\
    (\text{le} \ \text{pc} (\text{fn}_\text{last}_\text{pc} \ f)) \rightarrow \\
    (\text{exec}\_\text{instr} \ p \ f \ \text{pc} \ \text{rs} \ \text{st} \ \text{pc}' \ \text{rs}' \ \text{st}') \rightarrow \\
    (\text{regset}\_\text{match}\_\text{approx} \ (\text{analyze}\_\text{function} \ f))(\text{pc}) \ \text{rs}) \rightarrow \\
    (\text{regset}\_\text{match}\_\text{approx} \ (\text{analyze}\_\text{function} \ f))(\text{pc}') \ \text{rs}').
\end{align*}
\]

**Lemma** \text{analysis}\_\text{entry}\_\text{point}:

\[
\begin{align*}
    \text{f}: \text{function}) (\text{rs}: \text{regset}) (\text{regset}\_\text{match}\_\text{approx} \ (\text{analyze}\_\text{function} \ f))(\text{O}) \ \text{rs}.
\end{align*}
\]

### 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, \( r := \text{add}(r1,r2) \) 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** \( \text{transf}_{\text{op}} \)

\[
[ra: \text{map } \text{D.T}; \text{op}: \text{operation}; \text{args: } \text{list reg}] :=
\]

\[
\text{Cases } (\text{abstr}\_\text{regs} \text{ ra args}) \text{ of }
\]

\[
\begin{align*}
(\text{Some } vl) & \Rightarrow \\
& ((O\text{const} (\text{eval operation op } vl)), \text{Nil}) \\
\mid \text{None} & \Rightarrow \\
& \text{Cases } (\text{op}, \text{args}) \text{ of } \\
& (O\text{add}, (\text{cons } r1 (\text{cons } r2 \text{ nil}))) \Rightarrow \\
& \text{Cases } ra.[r1] \text{ of } \\
& (A.inj O) \Rightarrow (O\text{move}, (\text{cons } r2 \text{ Nil})) \\
& | (A.inj v) \Rightarrow ((O\text{addimm } v), (\text{cons } r2 \text{ Nil})) \\
& | - \Rightarrow \\
& \text{Cases } ra.[r2] \text{ of } \\
& (A.inj O) \Rightarrow (O\text{move}, (\text{cons } r1 \text{ Nil})) \\
& | (A.inj v) \Rightarrow ((O\text{addimm } v), (\text{cons } r1 \text{ Nil})) \\
& | - \Rightarrow (\text{op}, \text{args}) \\
& \text{end}
\]

\[
\text{end}
\]

| (O\text{sub}, (\text{cons } r1 (\text{cons } r2 \text{ nil}))) \Rightarrow \\
& \text{Cases } ra.[r2] \text{ of } \\
& (A.inj O) \Rightarrow (O\text{move}, (\text{cons } r1 \text{ Nil})) \\
& | (A.inj v) \Rightarrow ((O\text{subimm } v), (\text{cons } r1 \text{ Nil})) \\
& | - \Rightarrow (\text{op}, \text{args}) \\
& \text{end}
\]

| - \Rightarrow \\
& (\text{op}, \text{args})
\]

\text{end}

\]

**Definition** \( \text{transf}_{\text{instr}} \)

\[
[ra: \text{map } \text{D.T}; n: \text{nat}; i: \text{instruction}] :=
\]

\[
\text{Cases } i \text{ of } \\
(I\text{op } op \text{ args } res) \Rightarrow \\
\text{let } oa = (\text{transf}_{\text{op}} \text{ ra.}(n) \text{ op args}) \text{ in } (I\text{op } (F\text{st } oa) (S\text{nd } oa) \text{ res}) \\
| (I\text{condbranch } cond \text{ args } dst) \Rightarrow \\
\text{Cases } (\text{abstr}\_\text{regs} \text{ ra.}(n) \text{ args}) \text{ of }
\]
(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)) →
\((\text{fn}_{\text{instr}} (\text{transf}_{\text{function}} \text{nm} f)) (\text{pc}) = \\
(\text{transf}_{\text{instr}} (\text{analyze}_{\text{function}} f) ~ \text{pc} (\text{fn}_{\text{instr}} f) (\text{pc})).\)

**Lemma transformed.function:**
\( (p : \text{program}) (\text{nm} : \text{nat}) \\
(\text{lt} \text{nm} (\text{prog}_{\text{num}} \text{functions} p)) \rightarrow \\
(\text{prog}_{\text{function}} (\text{transf}_{\text{program}} p)) (\text{nm}) = \\
(\text{transf}_{\text{function}} \text{nm} (\text{prog}_{\text{function}} p) (\text{nm})).\)

The main semantic equivalence result follows:

**Lemma transf.function.correct:**
\( (p : \text{program}) (f : \text{function}) \\
(\text{pc} : \text{nat}) (\text{rs} : \text{regset}) (\text{st} : \text{store}) (\text{res} : \text{value}) (\text{st'} : \text{store}) \\
(\text{exec}_{\text{function}} p f \text{pc} \text{rs} \text{st} \text{res} \text{st'}) \rightarrow \\
(\text{nm} : \text{nat}) \\
(\text{le} \text{pc} (\text{fn}_{\text{last}} \text{pc} f)) \rightarrow \\
(\text{regset}_{\text{match}}_{\text{approx}} (\text{analyze}_{\text{function}} f) (\text{pc}) \text{rs}) \rightarrow \\
(\text{exec}_{\text{function}} (\text{transf}_{\text{program}} p) (\text{transf}_{\text{function}} \text{nm} f) \text{pc} \text{rs} \text{st} \text{res} \text{st'}).\)

This result is shown by simultaneous induction on the derivation of the \(\text{exec}_{\text{function}}\) predicate, and on the derivation of the \(\text{exec}_{\text{instr}}\) predicate, using the following proposition as the induction hypothesis for \(\text{exec}_{\text{instr}}\):

**Check**
\( (p : \text{program}) (f : \text{function}) \\
(\text{pc} : \text{nat}) (\text{rs} : \text{regset}) (\text{st} : \text{store}) \\
(\text{pc'} : \text{nat}) (\text{rs'} : \text{regset}) (\text{st'} : \text{store}) \\
(\text{exec}_{\text{instr}} p f \text{pc} \text{rs} \text{st} \text{pc'} \text{rs'} \text{st'}) \rightarrow \\
(\text{nm} : \text{nat}) \\
(\text{le} \text{pc} (\text{fn}_{\text{last}} \text{pc} f)) \rightarrow \\
(\text{regset}_{\text{match}}_{\text{approx}} (\text{analyze}_{\text{function}} f) (\text{pc}) \text{rs}) \rightarrow \\
(\text{exec}_{\text{instr}} (\text{transf}_{\text{program}} p) (\text{transf}_{\text{function}} \text{nm} f) \text{pc} \text{rs} \text{st} \text{pc'} \text{rs'} \text{st'}).\)

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 \((\text{pc}, \text{rs}, \text{st})\) to state \((\text{pc'}, \text{rs'}, \text{st'})\), the transformed program performs the same execution step, provided the register set “before” \((\text{rs})\) matches the results of the static analysis at point \(\text{pc}\). The latter hypothesis holds at every step, because it holds at the entrance to each function (lemma \text{analysis_entry_point}) and is preserved by every execution step (lemma \text{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:**
\( (p : \text{program}) (\text{st} : \text{store}) (\text{res} : \text{value}) (\text{st'} : \text{store}) \\
(\text{exec}_{\text{program}} p \text{st} \text{res} \text{st'}) \rightarrow \\
(\text{exec}_{\text{program}} (\text{transf}_{\text{program}} p) \text{st} \text{res} \text{st'}).\)
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.

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: \( \text{Boolean}.\Top \) denotes a live register, and \( \text{Boolean}.\Bot \) denotes a dead register.

Module \( D := (\text{Regmap Boolean}) \).

Fixpoint \( \text{set} \_\text{live} \) \[ \[ d : D.T; \text{regs} : (\text{list} \text{reg}) \] : D.T :=
Cases \( \text{regs} \) of
\( \text{nil} \Rightarrow d \)
\( (\text{cons} \ r \ \text{rs}) \Rightarrow \text{(set} \_\text{live} \ d \ \text{rs}).[r] \leftarrow \text{Boolean}.\Top \)
\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 \( \text{transfer} \) \[ f : \text{function}; \text{pc} : \text{nat}; \text{after} : D.T \] :=
Cases \( (\text{fn} \_\text{instr} \ f).(\text{pc}) \) of
\( \text{Inop} \Rightarrow \text{after} \)
\( (\text{Iop} \ \text{op} \ \text{args} \ \text{res}) \Rightarrow \cases { \text{after} \ \text{after} } \text{after} \)
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

**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:

(is_move_operation op args) = (Some reg r) →
  op = Omove ∧ args = (cons r Nil).

**Lemma** eval_move_operation:

(is_move_operation op args) = (Some reg arg) →
  (eval_operation op (eval_regs rs args)) = rs.[arg].

**Lemma** is_move_operation_rewritten:

(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**\(\text{list}\_\text{live}\_\text{regs}: D. T \rightarrow \text{(list}\ \text{reg}).\)

**Axiom**\(\text{list}\_\text{live}\_\text{regs}\_\text{correct}:\)

\[(\text{live}: D. T) (r: \text{reg}) \text{ live.}[r] = \text{Boolean}\_\text{Top} \rightarrow (\text{In } r \ (\text{list}\_\text{live}\_\text{regs}\ \text{live})).\]

We now define the construction of the interference graph.

**Fixpoint** \(\text{add}\_\text{interf}\_\text{live}\)

\[\text{[filter: reg into bool; res: reg; interf: graph; lregs: (list}\ \text{reg]} : \text{ graph :=}
\]

\[
\text{Cases lregs of}
\]

\[
\quad \text{nil } \Rightarrow \text{ interf}
\]

\[
\quad \mid (\text{cons } r \ lregs') \Rightarrow
\]

\[
\quad \quad \quad \text{ (add}\_\text{interf}\_\text{live}\ \text{filter}\ \text{res}
\]

\[
\quad \quad \quad \quad \quad \text{ (if } (\text{filter } r) \text{ then } (\text{add}\_\text{edge } r \ res \ \text{interf}) \text{ else interf})
\]

\[
\quad \quad \quad \quad \quad \ lregs')
\]

\[\textend.\]

**Definition** \(\text{filter}\_\text{add}\_\text{interf}\_\text{op} [\text{res: reg; r: reg}] :=\)

\[\text{Cases } (\text{reg}\_\text{eq}\_\text{dec} r \ \text{res}) \text{ of } (\text{left } \_ ) \Rightarrow \text{false } \mid (\text{right } \_ ) \Rightarrow \text{true } \text{end}.\]

**Definition** \(\text{add}\_\text{interf}\_\text{move} [\text{interf: graph; res: reg; live: D. T}] :=\)

\[(\text{add}\_\text{interf}\_\text{live}
\]

\[
\quad \text{(filter}\_\text{add}\_\text{interf}\_\text{op} \text{res)
\]

\[
\quad \text{res interf (list}\_\text{live}\_\text{regs}\ \text{live}).\]

**Definition** \(\text{filter}\_\text{add}\_\text{interf}\_\text{move} [\text{arg: reg; res: reg; r: reg}] :=\)

\[\text{Cases } (\text{reg}\_\text{eq}\_\text{dec} r \ \text{res}) \text{ of}
\]

\[
\quad \text{(left } \_ ) \Rightarrow \text{false
\]

\[
\quad \mid (\text{right } \_ ) \Rightarrow
\]

\[
\quad \quad \text{Cases } (\text{reg}\_\text{eq}\_\text{dec} r \ \text{arg}) \text{ of}
\]

\[
\quad \quad \quad \text{(left } \_ ) \Rightarrow \text{false
\]

\[
\quad \quad \mid (\text{right } \_ ) \Rightarrow \text{true
\]

\[\text{end}\]

**end.**

**Definition** \(\text{add}\_\text{interf}\_\text{move}[\text{interf: graph; arg: reg; res: reg; live: D. T}] :=\)
\begin{align*}
\text{add\_interf\_live} & \\
\text{filter\_add\_interf\_move arg res} & \\
\text{res interf (list\_live\_regs live)}. & \\
\end{align*}

**Definition** \text{add\_interf\_instr}

\[
[f: \text{function}; \text{live: } (\text{map } D. T); \text{interf: } \text{graph}; \text{pc: } \text{nat}] :=
\begin{cases}
\text{Inop} \Rightarrow \text{interf} \\
(Iop \ op \ args \ res) \Rightarrow \text{Cases live}.(\text{pc}).[\text{res}] \text{ of} \\
\quad \text{Boolean.Bot} \Rightarrow \text{interf} \\
\quad \text{Boolean.Top} \Rightarrow \text{Cases (is\_move\_operation \ op \ args) \ of} \\
\quad \quad (\text{Some arg}) \Rightarrow (\text{add\_interf\_move interf arg res live}.(\text{pc})) \\
\quad \quad \text{None} \Rightarrow (\text{add\_interf\_op interf res live}.(\text{pc}))
\end{cases}
\end{align*}

\begin{align*}
\text{add\_interf\_list} & \\
\text{interf: } \text{graph}; \text{live: } D. T; \text{rl: } (\text{list } \text{reg})] : \text{graph} :=
\begin{cases}
\text{nil} \Rightarrow \text{interf} \\
(\text{cons } r \ rs) \Rightarrow (\text{add\_interf\_list (add\_interf\_op interf } r \ \text{live}) \ \text{live rs})
\end{cases}
\end{align*}

**Definition** \text{interference\_graph}[f: \text{function}; \text{live: } (\text{map } D. T)] :=

\[
(\text{add\_interf\_instrs f live} \\
(\text{add\_interf\_list empty\_graph (transfer f O live}.(O)) (\text{fn\_params f})) \\
(S (\text{fn\_last\_pc f})))).
\]
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 \textit{add\_interf\_live\_correct}:

\begin{equation*}
\text{(filter: reg} \rightarrow \text{bool) (res: reg) (lregs: (list reg)) (interf: graph)} \\
\text{let interf'} = (\text{add\_interf\_live filter res interf lregs} \text{ in)} \\
\text{((r: reg)} \\
\text{(In r lregs) } \rightarrow \text{(filter r)} = \text{true } \rightarrow \text{(has\_edge r res interf')} = \text{true)} \\
\wedge ((r1, r2: reg) \\
\text{(has\_edge r1 r2 interf)} = \text{true } \rightarrow \text{(has\_edge r1 r2 interf')} = \text{true}.)
\end{equation*}

Lemma \textit{add\_interf\_op\_correct}:

\begin{equation*}
\text{(interf: graph) (res: reg) (live: D.T) (r: reg)} \\
\text{live.[r] = Boolean.Ttop } \rightarrow \text{r } \neq \text{res } \rightarrow \\
\text{(has\_edge r res (add\_interf\_op interf res live)) = true.}
\end{equation*}

Lemma \textit{add\_interf\_op\_increasing}:

\begin{equation*}
\text{(interf: graph) (res: reg) (live: D.T) (r1,r2: reg)} \\
\text{(has\_edge r1 r2 interf)} = \text{true } \rightarrow \\
\text{(has\_edge r1 r2 (add\_interf\_op interf res live)) = true.}
\end{equation*}

Lemma \textit{add\_interf\_move\_correct}:

\begin{equation*}
\text{(interf: graph) (arg,res: reg) (live: D.T) (r: reg)} \\
\text{live.[r] = Boolean.Ttop } \rightarrow \text{r } \neq \text{arg } \rightarrow \text{r } \neq \text{res } \rightarrow \\
\text{(has\_edge r res (add\_interf\_move interf arg res live)) = true.}
\end{equation*}

Lemma \textit{add\_interf\_move\_increasing}:

\begin{equation*}
\text{(interf: graph) (arg,res: reg) (live: D.T) (r1,r2: reg)} \\
\text{(has\_edge r1 r2 interf)} = \text{true } \rightarrow \\
\text{(has\_edge r1 r2 (add\_interf\_move interf arg res live)) = true.}
\end{equation*}

Lemma \textit{add\_interf\_instr\_increasing}:

\begin{equation*}
\text{(f: function) (live: (map D.T)) (interf: graph) (pc: nat) (r1,r2: reg)} \\
\text{(has\_edge r1 r2 interf)} = \text{true } \rightarrow \\
\text{(has\_edge r1 r2 (add\_interf\_instr f live interf pc)) = true.}
\end{equation*}

Lemma \textit{add\_interf\_list\_increasing}:

\begin{equation*}
\text{(rl: (list reg)) (interf: graph) (live: D.T) (r1,r2: reg)} \\
\text{(has\_edge r1 r2 interf)} = \text{true } \rightarrow \\
\text{(has\_edge r1 r2 (add\_interf\_list interf live rl)) = true.}
\end{equation*}

Lemma \textit{add\_interf\_list\_correct}:...
The following complicated predicate defines exactly what edges are must be in an interference graph \texttt{interf} in order for this graph to reflect correctly the liveness information at point \texttt{pc}.

\textbf{Definition} \texttt{correct\_interf\_instr}

\begin{verbatim}
[f: function; live: (map D.T); interf: graph; pc: nat] :=
Cases (fn\_instr f).(pc) of
  (top op args res) ⇒
    Cases (is\_move\_operation op args) of
      (Some arg) ⇒
        (r: reg)
        r ≠ res → r ≠ arg →
        (has\_edge r res interf) = true
      | None ⇒
        (r: reg)
        r ≠ res →
        (has\_edge r res interf) = true
    end
  | (Iload mode args res) ⇒
    (r: reg)
    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.
\end{verbatim}

\textbf{Lemma} \texttt{add\_interf\_instr\_correct}:

\begin{verbatim}
(f: function) (live: (map D.T)) (interf: graph) (pc: nat)
(correct\_interf\_instr f live (add\_interf\_instr f live interf pc) pc).
\end{verbatim}

\textbf{Lemma} \texttt{add\_interf\_instrs\_increasing}:

\begin{verbatim}
(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.
\end{verbatim}

\textbf{Lemma} \texttt{correct\_interf\_instr\_increasing}:

\begin{verbatim}
(f: function) (live: (map D.T)) (interf1, interf2: graph) (pc: nat)
\end{verbatim}
((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:**

\(\begin{align*}
(f: function) & \ (\text{live: (map D.T)}) \ (pc: nat) \ (interf: graph) \\
((p: nat) \ (le pc p) & \ → \ (le p (fn_last_pc f)) → \\
\quad \ (correct_interf_instr f live interf p)) →
\quad \ ((p: nat) \ (le p (fn_last_pc f)) → \\
\quad \quad \ (correct_interf_instr f live (add_interf_instrs f live interf pc) p)).
\end{align*}\)

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:**

\(\begin{align*}
(f: function) & \ (\text{live: (map D.T)}) \ (pc: nat) \\
(le pc (fn_last_pc f)) & → \\
\quad \ (correct_interf_instr f live (interference_graph f live) pc).
\end{align*}\)

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:**

\(\begin{align*}
(f: function) & \ (\text{live: (map D.T)}) \ (r1,r2: reg) \\
(In r1 (fn_params f)) & → \\
\quad \ (transfer f O live.(O)).[r2] = Boolean.Top → \\
\quad \quad \ r1 \not= r2 → \\
\quad \quad \ \ (has_edge r1 r2 (interference_graph f live)) = true.
\end{align*}\)

### 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  \texttt{agree\_live\_regs} \[\text{live}: D.T; \text{rs1,rs2}: \text{regset}] := \\
(r: \text{reg}) \text{live.}[r] = \text{Boolean.}\text{Top} \rightarrow \text{rs1.}[r] = \text{rs2.}[\text{assign } r].

What follows is a long list of lemmas expressing properties of the \texttt{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.

\textbf{Lemma} \texttt{agree\_increasing}: \\
\[(\text{live1,live2}: D.T) (\text{rs1,rs2}: \text{regset}) \\\n(D.\text{ge live1 live2}) \rightarrow \\\n(\text{agree\_live\_regs live1 rs1 rs2}) \rightarrow \\\n(\text{agree\_live\_regs live2 rs1 rs2}).\]

Two useful special cases of \texttt{agree\_increasing}.

\textbf{Lemma} \texttt{agree\_set\_live\_1}: \\
\[(\text{live}: D.T) (r: \text{reg}) (\text{rs1,rs2}: \text{regset}) \\\n(\text{agree\_live\_regs (live.}[r] \leftarrow \text{Boolean.}\text{Top}) \text{rs1 rs2}) \rightarrow \\\n(\text{agree\_live\_regs live rs1 rs2}).\]

\textbf{Lemma} \texttt{agree\_set\_live\_N}: \\
\[(\text{args}: (\text{list reg})) (\text{live}: D.T) (\text{rs1,rs2}: \text{regset}) \\\n(\text{agree\_live\_regs (set\_live live args) rs1 rs2}) \rightarrow \\\n(\text{agree\_live\_regs live rs1 rs2}).\]

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 \(n\).

\textbf{Lemma} \texttt{analysis\_correct}: \\
\[(f: \text{function}) (n, s: \text{nat}) \\\n(\text{le } n (\text{fn\_last\_pc } f)) \rightarrow (\text{In } s (\text{successors } f n)) \rightarrow \\\n(\text{rs1,rs2}: \text{regset}) \\\n(\text{agree\_live\_regs (analyze\_function } f).(n) \text{rs1 rs2}) \rightarrow \\\n(\text{agree\_live\_regs (transfer } f s (analyze\_function } f).(s) \text{rs1 rs2}).\]

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.

\textbf{Lemma} \texttt{agree\_eval\_regs}: \\
\[(\text{args}: (\text{list reg})) (\text{live}: D.T) (\text{rs1, rs2}: \text{regset}) \\\n(\text{agree\_live\_regs (set\_live live args) rs1 rs2}) \rightarrow \\\n(\text{eval\_regs rs1 args}) = (\text{eval\_regs rs2 (PolyList.map assign args)}).\]

If a register is dead, assigning it an arbitrary value in \texttt{rs1} and leaving \texttt{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.)

\textbf{Lemma} \texttt{agree\_assign\_dead\_reg}: \\
\[(\text{live}: D.T) (r: \text{reg}) \\\n(\text{rs1,rs2}: \text{regset}) (v: \text{value}) \\\n\text{live.}[r] = \text{Boolean.}\text{Bot} \rightarrow \\\n(\text{agree\_live\_regs live rs1 rs2}) \rightarrow \\\n(\text{agree\_live\_regs live } (\text{rs1.}[r] \leftarrow v) \text{rs2}).\]
Setting register \( r \) to the value \( v \) in \( rs1 \) and setting register \( \text{assign } r \) to the value \( v \) in \( rs2 \) preserves agreement, provided that all live registers except \( r \) interfere with \( r \).

**Lemma** \texttt{agree\_assign\_live\_reg}:

\[
\begin{align*}
&\text{(live: } D, T) \text{ (r: reg)} \\
&(rs1, rs2: \text{regset}) \text{ (v: value)} \\
&(s: \text{reg}) \\
&\quad\text{live}[s] = \text{Boolean.Top} \rightarrow r \neq s \rightarrow (\text{has_edge } r \ s \ \text{interf}) = \text{true}) \rightarrow \\
&(\text{agree\_live\_regs} \ (\text{live}.[r] \leftarrow \text{Boolean.Bot}) \ rs1 \ rs2) \rightarrow \\
&(\text{agree\_live\_regs} \ \text{live} (rs1.[r] \leftarrow v) (rs2.[\text{assign } r] \leftarrow v)).
\end{align*}
\]

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 \( \text{arg} \). (This corresponds to a register-register move instruction.) In this case, the condition can be weakened: it suffices that all live registers except \( \text{arg} \) and \( \text{res} \) interfere with \( \text{res} \).

**Lemma** \texttt{agree\_move\_live\_reg}:

\[
\begin{align*}
&\text{(live: } D, T) \ (\text{arg, res: reg}) \ (rs1, rs2: \text{regset}) \\
&(s: \text{reg}) \\
&\quad\text{live}[s] = \text{Boolean.Top} \rightarrow \text{res} \neq s \rightarrow \text{arg} \neq s \rightarrow \\
&(\text{has_edge } \text{res } s \ \text{interf}) = \text{true}) \rightarrow \\
&(\text{agree\_live\_regs}) \\
&\quad(\text{set\_live} \ (\text{live}.[\text{res}] \leftarrow \text{Boolean.Bot}) \ (\text{cons arg Nil})) \ rs1 \ rs2) \rightarrow \\
&(\text{agree\_live\_regs} \ \text{live} \ (rs1.[\text{res}] \leftarrow (rs1.[\text{arg}])) \\
&(rs2.[\text{assign res}] \leftarrow (rs2.[\text{assign arg}])).
\end{align*}
\]

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** \texttt{agree\_dummy\_move}:

\[
\begin{align*}
&\text{(live: } D, T) \ (\text{arg, res: reg}) \ (rs1, rs2: \text{regset}) \\
&(\text{assign arg}) = (\text{assign res}) \rightarrow \\
&(\text{agree\_live\_regs}) \\
&\quad(\text{set\_live} \ (\text{live}.[\text{res}] \leftarrow \text{Boolean.Bot}) \ (\text{cons arg Nil})) \ rs1 \ rs2) \rightarrow \\
&(\text{agree\_live\_regs} \ \text{live} \ (rs1.[\text{res}] \leftarrow (rs1.[\text{arg}])) \ rs2).
\end{align*}
\]

This is a generalization of \texttt{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** \texttt{agree\_set\_regs}:

\[
\begin{align*}
&\text{(live: } D, T) \ (vl: \text{list value}) \ (rl: \text{list reg}) \ (rs1, rs2: \text{regset}) \\
&\ (\text{[(r1, r2: reg}) \\
&\quad (\text{In r1 rl} \rightarrow \text{live}.[r2] = \text{Boolean.Top} \rightarrow r1 \neq r2 \rightarrow \\
&\quad (\text{has_edge } r1 \ r2 \ \text{interf}) = \text{true}) \rightarrow \\
&\quad(\text{agree\_live\_regs} \ \text{live} \ rs1 \ rs2) \rightarrow \\
&\quad(\text{agree\_live\_regs} \ \text{live} \ (\text{set\_regs} rs1 \ rl vl) (\text{set\_regs} rs2 \ (\text{PolyList.map assign rl}) vl))).
\end{align*}
\]

This lemma shows agreement between the initial register sets at function entrance in the original
and transformed program.

**Lemma** \texttt{agree\_entry\_regs}:

\[
\text{(live: D.T) (params: list reg) (vl: list value)}\]

\[
((r_1, r_2: \text{reg})
\quad \begin{align*}
& (\text{In } r_1 \text{ params}) \rightarrow \text{live.}[r_2] = \text{Boolean.Top} \rightarrow r_1 \neq r_2 \rightarrow \\
& (\text{has\_edge } r_1 \ r_2 \ \text{interf}) = \text{true}) \rightarrow \\
& \text{(agree\_live\_regs live)} \\
& \text{(set\_regs (regmap\_init value bad\_value) params vl)} \\
& \text{(set\_regs (regmap\_init value bad\_value) (PolyList.map assign params) vl)}).
\]

End \texttt{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 \texttt{Inop} instructions);
- “move” instructions whose source and destination registers are identical after register assignment are eliminated (also by turning them into \texttt{Inop}).

The \texttt{Inop} instructions will be suppressed later (see module \texttt{Uselesscode}).

**Definition** \texttt{transf\_instr}

\[
\text{Cases } i \text{ of } \\
\quad \text{Inop } \Rightarrow \text{Inop} \\
\quad | \text{Iop op args res } \Rightarrow \\
\quad \quad \text{Cases live.(pc).[res] of } \\
\quad \quad \quad \text{Boolean.Bot } \Rightarrow \text{Inop} \\
\quad \quad | \text{Boolean.Top } \Rightarrow \\
\quad \quad \quad \text{Cases (is\_move\_operation op args) of } \\
\quad \quad \quad \quad \text{(Some arg) } \Rightarrow \\
\quad \quad \quad \quad \quad \text{Cases (reg\_eq\_dec (assign arg) (assign res)) of } \\
\quad \quad \quad \quad \quad \quad \text{(left _) } \Rightarrow \text{Inop} \\
\quad \quad \quad \quad \quad \quad | \text{(right _) } \Rightarrow \text{Iop op (PolyList.map assign args) (assign res)} \\
\quad \quad \quad \quad \quad \quad \text{end} \\
\quad \quad \quad | \text{None } \Rightarrow \\
\quad \quad \quad \quad \text{(Iop op (PolyList.map assign args) (assign res))} \\
\quad \quad \quad \text{end} \\
\quad \quad \text{end} \\
\quad | \text{Iload mode args res } \Rightarrow \\
\quad \text{Cases live.(pc).[res] of }
\]

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
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 |
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.


### 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 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)) →
There exists a “last reachable” instruction: an instruction that is reachable, but all following instructions are not.

**Lemma last_reachable:**

\[(f : function)\]

\[(EX \; pc \mid (le \; pc \; (fn\_last\_pc \; f)))\]

\[\land \; (analyze\_function \; f).(pc) = Boolean.\ Top\]

\[\land \; ((p : nat) \; (lt \; pc \; p) \rightarrow (le \; p \; (fn\_last\_pc \; f))) \rightarrow\]

\[(analyze\_function \; f).(p) = Boolean.\ Bot).\]

This “last reachable” instruction must be a terminal instruction.

**Lemma last_reachable_terminal:**

\[(f : function) \; (pc : nat)\]

\[(le \; pc \; (fn\_last\_pc \; f)) \rightarrow\]

\[(analyze\_function \; f).(pc) = Boolean.\ Top \rightarrow\]

\[((p : nat) \; (lt \; pc \; p) \rightarrow (le \; p \; (fn\_last\_pc \; f))) \rightarrow\]

\[(analyze\_function \; f).(p) = Boolean.\ Bot) \rightarrow\]

\[(instr\_terminal \; (fn\_instr \; f).(pc)).\]

### 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 :=\)**

\[\text{Cases } p \text{ of}\]

\[O \Rightarrow O\]

\[| \; (S \; p') \Rightarrow\]

\[\text{if } keep.(p') \text{ then } (S \; (count\_true \; p')) \text{ else } (count\_true \; p')\]

**end.**

**Lemma count_true_increasing:**

\[(q, \; p : nat)\]

\[(le \; p \; q) \rightarrow (le \; (count\_true \; p) \; (count\_true \; q)).\]

**Lemma count_true_constant:**

\[(q, \; p : nat)\]

\[(le \; p \; q) \rightarrow\]

\[((r : nat) \; (le \; p \; r) \rightarrow (lt \; r \; q) \rightarrow keep.(r) = false) \rightarrow\]

\[(count\_true \; p) = (count\_true \; q).\]
Lemma \textit{count\_true\_continuous}:
\[
(p: \text{nat}) \ (v: \text{nat}) \\
(\text{lt} \ v \ (\text{count\_true} \ p)) \rightarrow \\
(\text{EX} \ q \ | \ (\text{lt} \ q \ p) \land \text{keep} \ (q) = \text{true} \land \ (\text{count\_true} \ q) = v).
\]

\textit{End Count\_true.}

We now define the transformation of a function \(f\).

\textbf{Section} \textit{Transform\_function}.

\textbf{Variable} \(f: \text{function}\).

\textbf{Definition} \textit{reach\_f} := (\text{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 \textit{Inop} instruction.

\textbf{Definition} \textit{keep\_instr} [\textit{reach}: (\text{map Boolean}.T); \textit{pc}: \text{nat}; \textit{i}: \text{instruction}] :=
\[
\text{Cases} \ \text{reach}.(\text{pc}) \ \text{of} \\
\quad \text{Boolean}.\text{Bot} \Rightarrow \text{false} \\
\quad | \text{Boolean}.\text{Top} \Rightarrow \\
\quad \quad \text{Cases} \ \text{i} \ \text{of} \\
\quad \quad \quad \text{Inop} \Rightarrow \text{false} \\
\quad \quad | \_ \Rightarrow \text{true} \\
\quad \text{end}
\]

\textbf{Definition} \textit{keep\_code} [\textit{reach}: (\text{map Boolean}.T)] :=
\[
(\text{map\_apply\_instruction\_bool} \ (\text{keep\_instr} \ \text{reach}) \ (\text{fn\_instr} \ f) \ (S \ (\text{fn\_last\_pc} \ f))).
\]

\textbf{Definition} \textit{keep\_f} := (\text{keep\_code} \ \text{reach\_f}).

\textbf{Lemma} \textit{keep\_code\_at}:
\[
(\text{reach}: \ (\text{map Boolean}.T)) \ (\text{pc}: \text{nat}) \\
(\le \ \text{pc} \ (\text{fn\_last\_pc} \ f)) \rightarrow \\
(\text{keep\_code} \ \text{reach}).(\text{pc}) = (\text{keep\_instr} \ \text{reach} \ \text{pc} \ (\text{fn\_instr} \ f).(\text{pc})).
\]

\textbf{Lemma} \textit{keep\_f\_at}:
\[
(\text{pc}: \text{nat}) \\
(\le \ \text{pc} \ (\text{fn\_last\_pc} \ f)) \rightarrow \\
\text{keep\_f}.(\text{pc}) = (\text{keep\_instr} \ \text{reach\_f} \ \text{pc} \ (\text{fn\_instr} \ f).(\text{pc})).
\]

\textbf{Lemma} \textit{code\_mapping\_entry}:
\[
(\text{keep}: \ (\text{map bool})) \ (p: \text{nat}) \\
(\le \ p \ (\text{fn\_last\_pc} \ f)) \rightarrow \\
(\text{code\_mapping\_entry} \ \text{keep}).(p) = (\text{count\_true} \ \text{keep} \ p).
\]

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 \textit{code\_mapping\_last}:
\begin{align*}
(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).
\end{align*}

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 \textit{code\_mapping\_in\_range}:
\begin{align*}
(p: nat) \\
(le \ p \ (fn\_last\_pc \ f)) \ → \\
reach\_f.(p) = Boolean.Top \ → \\
(le \ mapping\_f.(p) \ (pred \ new\_code\_size)).
\end{align*}

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 \textit{relocate\_instr} [mapping: \textit{map} nat; \textit{i}: \textit{instruction}] :=
\begin{align*}
\text{Cases } \textit{i} \text{ of} \\
(Ibranch \ dest) ⇒ (Ibranch \ mapping.(dest)) \\
| (Icondbranch \ cond \ args \ dest) ⇒ (Icondbranch \ cond \ args \ mapping.(dest)) \\
| _ ⇒ \textit{i}
\end{align*}

Fixpoint \textit{transf\_code\_aux}
\begin{align*}
[\textit{keep}: \textit{map} \textit{bool}; \textit{mapping}: \textit{map} \textit{nat}; \\
\textit{newcode}: \textit{map} \textit{instruction}; \textit{pc}: \textit{nat}; \textit{newpc}: \textit{nat}; \\
\textit{count}: \textit{nat}] : \textit{map} \textit{instruction} :=
\text{Cases } \textit{count} \text{ of} \\
O ⇒ \textit{newcode} \\
| (S \ \textit{count'}) ⇒ \\
\text{if } \textit{keep}.(pc) \text{ then} \\
(\textit{transf\_code\_aux} \ \textit{keep} \ \textit{mapping} \\
(\textit{newcode}.(\textit{newpc}) \leftarrow (\textit{relocate\_instr} \ \textit{mapping} \ (\textit{fn\_instr \ f}).(pc)))) \\
(S \ \textit{pc}) \ (S \ \textit{newpc} \ \textit{count'}) \\
\text{else} \\
(\textit{transf\_code\_aux} \ \textit{keep} \ \textit{mapping} \\
\textit{newcode} \\
(S \ \textit{pc}) \ \textit{newpc} \ \textit{count'})
\end{align*}

Definition \textit{transf\_code} [\textit{keep}: \textit{map} \textit{bool}; \textit{mapping}: \textit{map} \textit{nat}] :=


(\texttt{transf\_code\_aux \texttt{keep mapping} \texttt{(map\_init\ instruction\ Inop)}}
\quad \texttt{O\ O\ (S\ (fn\_last\_pc\ f))}).

\textbf{Definition} \texttt{transf\_f := (transf\_code\ \texttt{keep\_f\ mapping\_f)}}.

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

\textbf{Remark} \texttt{transf\_code\_aux\ characterization:}
\begin{align*}
(\text{\texttt{count: nat}}) \quad (\text{\texttt{newcode: (map\ instruction)}}) \quad (\text{\texttt{pc: nat}}) \\
(\texttt{plus\ pc\ count}) \quad = \quad (S\ (fn\_last\_pc\ f)) \rightarrow \\
\text{\texttt{let finalcode = (transf\_code\_aux\ keep\_f\ mapping\_f\ newcode}}} \\
\quad \text{\texttt{pc\ (count\_true\ keep\_f\ pc\ count)}} \quad \text{\texttt{in}} \\
\quad (((p: \texttt{nat}) \quad (lt\ p\ pc) \rightarrow \texttt{keep\_f\.(p) = true}) \\
\quad \texttt{newcode\.(mapping\_f\.(p)) = (relocate\_instr\ mapping\_f\ (fn\_instr\ f).(p))) \rightarrow} \\
\quad (((p: \texttt{nat}) \quad (lt\ p\ (plus\ pc\ count)) \rightarrow \texttt{keep\_f\.(p) = true}) \\
\quad \texttt{finalcode\.(mapping\_f\.(p)) = (relocate\_instr\ mapping\_f\ (fn\_instr\ f).(p)))}).
\end{align*}

\textbf{Lemma} \texttt{transf\_code\_at:}
\begin{align*}
(\text{\texttt{pc: nat}}) \\
(\texttt{le\ pc\ (fn\_last\_pc\ f)}) \rightarrow \\
\texttt{keep\_f\.(pc) = true} \\
\texttt{transf\_f\.(mapping\_f\.(pc)) = (relocate\_instr\ mapping\_f\ (fn\_instr\ f).(pc))}.
\end{align*}

\textbf{Lemma} \texttt{reverse\_transf\_code\_at:}
\begin{align*}
(\text{\texttt{newpc: nat}}) \\
(\texttt{lt\ newpc\ new\_code\_size}) \rightarrow \\
(\text{\texttt{EX\ pc\ |}} \\
\quad (\texttt{le\ pc\ (fn\_last\_pc\ f)}) \\
\quad \texttt{\wedge\ transf\_f\.(newpc) = (relocate\_instr\ mapping\_f\ (fn\_instr\ f).(pc))} \\
\quad \texttt{\wedge\ keep\_f\.(pc) = true}).
\end{align*}

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

\textbf{Lemma} \texttt{transf\_code\_valid:}
\begin{align*}
(\text{\texttt{pc: nat}}) \\
(\texttt{le\ pc\ (pred\ new\_code\_size)}) \rightarrow \\
(\text{\texttt{instr\_valid\ transf\_f\.(pc)\ (pred\ new\_code\_size)}}).
\end{align*}

\textbf{Lemma} \texttt{transf\_code\_last\_instr\ terminal:}
\begin{align*}
(\text{\texttt{instr\_terminal\ transf\_f\.(pred\ new\_code\_size)}}).
\end{align*}

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

\textbf{Definition} \texttt{transf\_function :=}
\begin{align*}
\texttt{let reach = (analyze\_function\ f)\ in} \\
\texttt{let keep = (keep\_code\ reach)\ in} \\
\texttt{let mapping = (code\_mapping\ keep)\ in} \\
\texttt{(make\_function} \\
\quad \texttt{(transf\_code\ keep\ mapping))}
\end{align*}
\begin{verbatim}
(fn_params f)
(pred (count_true keep (S (fn_last_pc f))))
transf_code_valid
transf_code_last_instr_terminal).
\end{verbatim}

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:

The main simulation result above is proved by mutual induction with the following property of the
\texttt{exec\_function} predicate.

\textbf{Lemma} \texttt{transf\_function\_correct}:
\begin{align*}
(p: \text{program}) & \quad (f: \text{function})
\quad (pc: \text{nat})
\quad (rs: \text{regset})
\quad (st: \text{store})
\quad (res: \text{value})
\quad (st': \text{store})
\quad (exec\_function\ p\ f\ pc\ rs\ st\ res\ st') \rightarrow \\
(le\ pc\ (fn\_last\_pc\ f)) & \rightarrow \\
(reach\_f\ f)(pc) = \text{Boolean}\_\text{Top} \rightarrow \\
\neg (\text{instr\_terminal}\ (fn\_instr\ f)(pc)) & \rightarrow \\
(mapping\_f\ f)(S\ pc) = \\
\text{Cases}\ (fn\_instr\ f)(pc)\ of \\
\quad \text{Inop} & \Rightarrow (mapping\_f\ f)(pc) \\
\quad \text{| } & \Rightarrow (S\ (mapping\_f\ f)(pc)) \\
\end{align*}
\texttt{end}

\textbf{Lemma} \texttt{transf\_program\_correct}:
\begin{align*}
(p: \text{program}) & \quad (st: \text{store})
\quad (res: \text{value})
\quad (st': \text{store})
\quad (exec\_program\ p\ st\ res\ st') \rightarrow \\
\quad (exec\_program\ (transf\_program\ p)\ st\ res\ st') \\
\end{align*}

\section{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 \texttt{exec\_instr} and \texttt{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 simulation 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. However, preliminary attempts 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.