A Simple Supercompiler Formally Verified in Coq

Dimitur Krustev

IGE+XAO Balkan, Bulgaria
dkrustev@ige-xao.com

Abstract. We study an approach for verifying the correctness of a simplified supercompiler in Coq. While existing supercompilers are not very big in size, they combine many different program transformations in intricate ways, so checking the correctness of their implementation poses challenges. The presented method relies on two important technical features to achieve a compact and modular formalization: first, a very limited object language; second, decomposing the supercompilation process into many sub-transformations, whose correctness can be checked independently. In particular, we give separate correctness proofs for two key parts of driving – normalization and positive information propagation – in the context of a non-Turing-complete expression sub-language. Though our supercompiler is currently limited, its formal correctness proof can give guidance for verifying more realistic implementations.

1 Introduction

Supercompilation [21,18] typically combines a small set of local source transformations with (function) unfolding/folding and generalization in an intricate way. Some general methods have been developed for verifying its correctness – both in the sense of semantics preservation [15] and concerning termination on all inputs [16]. Nonetheless, in view of recent advances in tools for formal computer-verified reasoning, it appears interesting to develop techniques for formal proofs of supercompiler correctness. If one is to pursue such a task, there are two options. The first is to follow as closely as possible the definition of a working supercompiler and to develop a proof of its correctness that can be verified automatically by an existing proof checker. This approach has already been shown to work on systems as complex as compilers for real-world programming languages [12] and operating system kernels [10], so it should be feasible, but probably a lot of work. Alternatively, we can start with simplifying the task as much as possible, so that formal proofs become much easier. This approach is useful even if one is not interested in computer-checked proofs, as it can shed new light on the interaction of the various ingredients of supercompilation.

To simplify the definition of supercompilation, we combine two methods. Firstly, we use a toy programming language operating on an equally simple value domain. Then we take a step forward by isolating as much as possible the different ingredients of supercompilation, with independent proofs of correctness

for each one, which are then combined in a modular way in a proof of correctness for the whole supercompiler. Traditionally supercompilation is presented as the combination of several processes – driving, "whistle", folding, generalization (see [18] for a good introduction of positive supercompilation). In the actual definition of the complete supercompiler algorithm this conceptual decomposition is typically blurred, and it appears non-trivial to exploit it directly in verification. We achieve a more significant separation of phases, with an individual proof of correctness for each of them:

- Driving (minus unfolding) is defined in the context of a non-Turing-complete language of simple expressions (whose denotational semantics is defined in Sect. 2), and is decomposed into two separate transformations:
  - Normalization (equivalent to simple deforestation [22] minus unfolding, Sect. 2.1);
  - Positive information propagation (Sect. 2.3). An interesting technical detail here is, that we use a variable-free language and a simple form of explicit substitutions [1] for propagating information (Sect. 2.2);
- We use a small imperative language, which embeds the expression sublanguage discussed above and whose programs contain a single while loop. We define its semantics in a big-step operational style (Sect. 3). Unfolding and folding are correspondingly replaced by a basic form of loop unrolling. The proof of correctness is cleanly split in two – correctness of one-step unrolling (Sect. 3.1), and correctness of repeated unrollings (Sect. 3.4);
- The treatment of the "whistle" brings no novelties beside the fact that the proof of termination is completely separated from the proof of (partial) correctness (Sect. 3.2). This is one of the few places where we "cheat" a bit – we take Kruskal's tree theorem as an assumption, as its formal proof is a big topic of its own;
- We ignore generalization for the time being, as it does not appear essential for our current definition of loop unrolling.

While the resulting supercompiler is too limited to be practically useful, it can still achieve interesting results on select small examples (Sect. 3.3). To give a glimpse of the separation of concerns achieved, here is a small example of a normalization (*normConv*) of a simple expression, contrasted with normalization plus positive information propagation (*norm*):

```
Eval compute in (let e := (IfNil Hd (Nil # Nil) (IfNil Hd Nil Tl))
  in (ntrm2trm (normConv e), ntrm2trm (norm e))).
=(IfNil Hd (Nil # Nil) (IfNil Hd Nil Tl), IfNil Hd (Nil # Nil) Tl)
```

Notice the removal of the redundant test in the second expression.

1.1 Notation

The text of this article is produced from a literate Coq script using the coqdoc tool [19]. All data type and function definitions, as well as the statements

of all lemmas/theorems, are given directly in Coq syntax. We use almost none of the more advanced or specific features of this proof assistant, so while our readers should be familiar with functional programming and first-order logic, they do not need prior experience with Coq. Coq contains a total functional programming sublanguage, similar in many respects to languages like Haskell and OCaml (modulo totality requirements). It permits well-founded inductive[1] data type definitions (keyword `Inductive` ...), non-recursive global definitions (`Definition`), structurally recursive global (`Fixpoint`) and local (*fix*) definitions, pattern matching (`match` ... `with` | ... ⇒ ... | ... `end`), lambda functions (`fun` ... ⇒ ...). Coq also embeds (a form of) intuitionistic logic[2], with the usual logical quantifiers and connectives (∀, ∃, →, *False*, *True*, ¬, ∧, ∨, ↔, =). Type information is specified as (x: T) for an object x having a type T. Computable types usually have sort `Set`, while logical propositions live in sort `Prop`. There is a computable type *bool*: `Set` with a number of operations on it, which should not be confused with the non-executable logical propositions and connectives living in `Prop`. We use other standard library types – natural numbers (*nat* with constructors O and S, and standard arithmetic operations), and lists (*list* X with constructors *nil* and ::, and standard list operations like *length* and ++ (append)).

We believe, that the formulation of definitions and lemma statements is more valuable in understanding this work than the detailed proofs themselves. Proofs can be quite lengthy and are usually expressed using a special tactic language – which can be difficult to follow outside of Coq. The complete proofs can always be checked inside the original Coq source. Therefore, we have omitted them here and only give brief informal hints for some of the more complicated ones. Most of the lengthier proofs are broken up into a series of lemmas, each one building on the previous ones, and culminating in a final theorem with a typically trivial proof. We do give the statements of all such lemmas, not only the main theorems. Furthermore, proofs of individual lemmas usually proceed straightforwardly by induction, and can be automated to a great extent using suitable heuristics for automatic proof search [4].

2 Simple Expression Language

We start with an extremely simple domain of values - binary trees (or, equivalently, Lisp-like S-expressions) with a single atom, *VNil*. As some of our built-in functions will be partial, we also include a second dedicated atom, *VBottom*, used to make all built-in functions total.

`Inductive` *Val*: `Set` :=
  | *VNil*: *Val* | *VCons*: *Val* → *Val* → *Val* | *VBottom*: *Val*.

[1] Coinductive definitions are also possible, but we do not use them here
[2] Classical reasoning is also possible in Coq, but not required here

The use of an untyped language is motivated by a hope for greater overall simplicity, although a move to a typed setting would certainly bring some benefits. Notice that the domain is lifted, as *VCons* is not strict w.r.t. *VBottom*: $VCons\ VBottom\ VBottom \neq VBottom$.

The expressions of our simple language are built of primitives for constructing (*Nil*, *Cons*) and deconstructing (*Sel*) binary trees, function composition and identity (*Cmp*, *Id*), and conditional expressions testing for null values. It is convenient to have also a bottom-building primitive, *Bottom*, but there is no way of testing for bottom:

```
Inductive Selector: Set := | HD | TL.

Inductive Trm: Set :=
  | Nil: Trm | Cons: Trm → Trm → Trm | Sel: Selector → Trm
  | Id: Trm | Cmp: Trm → Trm → Trm
  | IfNil: Trm → Trm → Trm → Trm | Bottom.
```

We can use Coq's `Notation` mechanism to add a small amount of syntax sugar (note that lower levels correspond to higher precedence).

```
Infix "$" := Cmp (at level 60, right associativity).
Notation Hd := (Sel HD). Notation Tl := (Sel TL).
Infix "#" := Cons (at level 62, right associativity).
```

A few things are notable in the choice of language. It is variable-free, all expressions denoting functions of type $Val \to Val$. It is the presence of pair constructor and selectors, as well as function composition, as primitives, that gives this language the ability to encode substitutions and to do away with variables. As the language is not Turing-complete, it is straightforward to give its semantics as a total function, *evalT*:

```
Definition evalSel (sel: Selector) (v: Val) : Val :=
  match v with
  | VCons v1 v2 ⇒ match sel with | HD ⇒ v1 | TL ⇒ v2 end
  | _ ⇒ VBottom
  end.

Definition evalSels (sels: list Selector) (v: Val) : Val :=
  fold_left (fun v sel ⇒ evalSel sel v) sels v.

Fixpoint evalT (t: Trm) (v: Val) {struct t} : Val :=
  match t with
  | Nil ⇒ VNil
  | Cons t1 t2 ⇒ VCons (evalT t1 v) (evalT t2 v)
  | Sel sel ⇒ evalSel sel v
  | Id ⇒ v
  | Cmp t1 t2 ⇒ evalT t1 (evalT t2 v)
  | IfNil t1 t2 t3 ⇒ match evalT t1 v with
```

```
    | VNil ⇒ evalT t2 v | VCons _ _ ⇒ evalT t3 v | VBottom ⇒ VBottom
    end
  | Bottom ⇒ VBottom
  end.
```

2.1 Normalization of Simple Expressions

The first step in our series of transformations will be to perform some standard normalizing simplifications to expressions. As the resulting expressions will always have a specific shape, we can define a special type for normal-form expressions:

```
Inductive NTrm: Set :=
  | NNil: NTrm | NCons: NTrm → NTrm → NTrm
  | NSelCmp: list Selector → NTrm
  | NIfNil: list Selector → NTrm → NTrm → NTrm
  | NBottom: NTrm.
```

The important difference is, that in normal forms function composition can only be applied to pair selectors, and that tests in conditional expressions are only of this special form of selector compositions. Notice that the selectors appear in reverse order in lists, and such lists of selectors can be directly interpreted as positions in the binary trees of values. Of course, normal forms can be injected back into the set of full-blown expressions:

```
Definition sels2trm (sels: list Selector): Trm := fold_left (fun t sel ⇒
  match t with | Id ⇒ Sel sel | _ ⇒ Cmp (Sel sel) t end) sels Id.

Fixpoint ntrm2trm (nt: NTrm) {struct nt} :Trm :=
  match nt with
  | NNil ⇒ Nil
  | NCons nt1 nt2 ⇒ Cons (ntrm2trm nt1) (ntrm2trm nt2)
  | NSelCmp sels ⇒ sels2trm sels
  | NIfNil sels nt1 nt2 ⇒ IfNil (sels2trm sels) (ntrm2trm nt1) (ntrm2trm nt2)
  | NBottom ⇒ Bottom
  end.
```

Using this injection, we can define a specialized evaluation function for normal terms by re-using the main evaluation function.

```
Definition evalNT (nt: NTrm) (v: Val) : Val := evalT (ntrm2trm nt) v.
```

Next we establish some basic properties involving *evalSels*, that will be useful in subsequent proofs.

```
Lemma evalT_sels2trm: ∀ sels: list Selector, ∀ v: Val,
  evalT (sels2trm sels) v = evalSels sels v.

Lemma evalSelsVBottom: ∀ sels: list Selector, evalSels sels VBottom = VBottom.
```

```
Lemma evalSelsAppend: ∀ sels1 sels2: list Selector, ∀ v: Val,
  evalSels (sels1 ++ sels2) v = evalSels sels2 (evalSels sels1 v).
```

The main normalization function, *normConv*, uses a number of auxiliary operations on normal-form expressions, dealing mostly with special cases of function composition. We list along the way some lemmas establishing characteristic properties of the functions defined. The simplest cases cover composition of a selector or a list of selectors with an expression.

```
Fixpoint normSelNCmp (sel: Selector) (nt: NTrm) {struct nt}: NTrm :=
  match nt with
  | NNil ⇒ NBottom
  | NCons nt1 nt2 ⇒ match sel with | HD ⇒ nt1 | TL ⇒ nt2 end
  | NSelCmp sels ⇒ NSelCmp (sels ++ (sel::nil))
  | NIfNil sels nt1 nt2 ⇒ NIfNil sels
      (normSelNCmp sel nt1) (normSelNCmp sel nt2)
  | NBottom ⇒ NBottom
  end.

Lemma normSelNCmpPreservesEval: ∀ (sel: Selector) (nt: NTrm) (v: Val),
  evalNT (normSelNCmp sel nt) v = evalSel sel (evalNT nt v).

Definition normSelsNCmp (sels: list Selector) (nt: NTrm) : NTrm :=
  fold_left (fun nt sel ⇒ normSelNCmp sel nt) sels nt.

Lemma normSelsNCmpPreservesEvalT: ∀ sels: list Selector, ∀ nt: NTrm,
  ∀ v: Val, evalT (ntrm2trm (normSelsNCmp sels nt)) v
  = evalSels sels (evalT (ntrm2trm nt) v).

Lemma normSelsNCmpPreservesEval: ∀ sels: list Selector, ∀ nt: NTrm, ∀ v: Val,
  evalNT (normSelsNCmp sels nt) v = evalSels sels (evalNT nt v).

Lemma normSelsNCmp_NSelCmp: ∀ (sels1 sels2: list Selector),
  normSelsNCmp sels1 (NSelCmp sels2) = NSelCmp (sels2 ++ sels1).
```

We also consider composition of selectors to the right of a normal-form expression *nt*.

```
Fixpoint normNCmpSels (nt: NTrm) (sels: list Selector) {struct nt}
  : NTrm := match nt with
  | NNil ⇒ NNil
  | NCons nt1 nt2 ⇒
      NCons (normNCmpSels nt1 sels) (normNCmpSels nt2 sels)
  | NSelCmp sels2 ⇒ NSelCmp (sels ++ sels2)
  | NIfNil sels2 nt1 nt2 ⇒ NIfNil (sels ++ sels2)
      (normNCmpSels nt1 sels) (normNCmpSels nt2 sels)
  | NBottom ⇒ NBottom
  end.
```

```
Lemma normNCmpSelsPreservesEval: ∀ sels: list Selector, ∀ nt: NTrm, ∀ v: Val,
  evalNT (normNCmpSels nt sels) v = evalNT nt (evalSels sels v).

Lemma normNCmpSels_app: ∀ (sels1 sels2: list Selector) (nt: NTrm),
  normNCmpSels nt (sels1 ++ sels2)
  = normNCmpSels (normNCmpSels nt sels2) sels1.
```

Next, we deal with building conditional expressions in normal form. If the normal form of the condition is a value-constructing primitive, we can statically reduce the whole if-expression. The other interesting case is when the condition is itself another if-expression – in this case we switch the order of the tests and duplicate the original outer *NIfNil* inside the branches of the new outer *NIfNil*.

```
Fixpoint normNIf (nt1 nt2 nt3: NTrm) {struct nt1} : NTrm :=
  match nt1 with
  | NNil ⇒ nt2
  | NCons _ _ ⇒ nt3
  | NSelCmp sels ⇒ NIfNil sels nt2 nt3
  | NIfNil sels nt1_1 nt1_2 ⇒ NIfNil sels
      (normNIf nt1_1 nt2 nt3) (normNIf nt1_2 nt2 nt3)
  | NBottom ⇒ NBottom
  end.

Lemma normNIfPreservesEvalT: ∀ nt1 nt2 nt3: NTrm, ∀ v: Val,
  evalT (ntrm2trm (normNIf nt1 nt2 nt3)) v
  = match evalT (ntrm2trm nt1) v with
    | VNil ⇒ evalT (ntrm2trm nt2) v
    | VCons _ _ ⇒ evalT (ntrm2trm nt3) v
    | VBottom ⇒ VBottom
    end.

Lemma normNIfPreservesEval: ∀ nt1 nt2 nt3: NTrm, ∀ v: Val,
  evalNT (normNIf nt1 nt2 nt3) v = match evalNT nt1 v with
    | VNil ⇒ evalNT nt2 v
    | VCons _ _ ⇒ evalNT nt3 v
    | VBottom ⇒ VBottom
    end.
```

The sequence of operations on normal-form expressions culminates in a function *normNCmp*, which permits to form the composition of two normal-form expressions without having function composition as primitive. The most interesting cases here involve the composition of *NIfNil* and *NSelCmp*/*NIfNil*:

```
Definition normNCmp : NTrm → NTrm → NTrm :=
  fix normNCmp_nt1 (nt1: NTrm): NTrm → NTrm :=
  fix normNCmp_nt2 (nt2: NTrm): NTrm :=
  match nt1 with
  | NNil ⇒ NNil
```

```
  | NCons nt1_1 nt1_2 ⇒
      NCons (normNCmp_nt1 nt1_1 nt2) (normNCmp_nt1 nt1_2 nt2)
  | NSelCmp sels ⇒ normSelsNCmp sels nt2
  | NIfNil sels nt1_1 nt1_2 ⇒ match nt2 with
    | NSelCmp sels2 ⇒ NIfNil (sels2 ++ sels)
        (normNCmpSels nt1_1 sels2) (normNCmpSels nt1_2 sels2)
    | NIfNil sels2 nt2_1 nt2_2 ⇒ NIfNil sels2
        (normNCmp_nt2 nt2_1) (normNCmp_nt2 nt2_2)
    | _ ⇒ normNIf (normSelsNCmp sels nt2)
        (normNCmp_nt1 nt1_1 nt2) (normNCmp_nt1 nt1_2 nt2)
    end
  | NBottom ⇒ NBottom
  end.
```

We can easily establish that the composition of 2 if-expressions can be replaced by pushing the first if-expression inside the branches of the second:

```
Lemma normNCmpIfIf: ∀ sels1 sels2: list Selector,
  ∀ nt1_1 nt1_2 nt2_1 nt2_2: NTrm, let nt1 := NIfNil sels1 nt1_1 nt1_2 in
  normNCmp nt1 (NIfNil sels2 nt2_1 nt2_2)
  = NIfNil sels2 (normNCmp nt1 nt2_1) (normNCmp nt1 nt2_2).
```

We also establish that *normNCmp* satisfies the defining property of function composition; this is the key lemma on which correctness of normalization relies:

```
Lemma normNCmpPreservesEval: ∀ nt1 nt2: NTrm, ∀ v: Val,
  evalNT (normNCmp nt1 nt2) v = evalNT nt1 (evalNT nt2 v).
```

The last lemma is a bit tricky to prove: as *normNCmp* is defined using nested lexicographic recursion, we must use nested induction in the proof and apply rewritings using the previously proved lemmas.

Finally, the stage is set for the conversion of arbitrary expressions into normal form:

```
Fixpoint normConv (t: Trm) {struct t} :NTrm :=
  match t with
  | Nil ⇒ NNil
  | Cons t1 t2 ⇒ NCons (normConv t1) (normConv t2)
  | Sel sel ⇒ NSelCmp (sel::nil)
  | Id ⇒ NSelCmp nil
  | Cmp t1 t2 ⇒ normNCmp (normConv t1) (normConv t2)
  | IfNil t1 t2 t3 ⇒ normNIf (normConv t1) (normConv t2) (normConv t3)
  | Bottom ⇒ NBottom
  end.
```

With all this in place, the main theorem establishing the correctness of normalization can be proved by straightforward induction on the expression structure:

```
Theorem normConvPreservesEval: ∀ (t: Trm) (v: Val),
  evalNT (normConv t) v = evalT t v.
```

We can see on an example, that *normConv* not only brings expressions into normal form, but also achieves some optimizations like deforestation:

```
Eval compute in (ntrm2trm (normConv ((IfNil Hd ((Tl $ Tl) # (Hd $ Tl)) Tl) $ (Nil # Id)))).

  = Tl # Hd : Trm
```

As we have seen in the introduction, however, normalization by itself does not eliminate redundant tests.

2.2 Emulating Substitutions

Before we tackle positive information propagation, we need to make a small detour and show how substitutions can be emulated inside our language, giving a simple form of explicit substitutions [1]. Let's first note that we can replace a set of values, denoted by variables, with a list structure built from pairs (e.g. [8]). Variables in this case can be replaced by positions in the list structure, represented by lists of pair selectors. For example, the expression *IfNil x1 x2 x3*, has three free variables. We can pack their values into a list – *x1 # x2 # x3* – and replace their references inside the expression with the corresponding list positions: *IfNil Hd* (*Hd* $ *Tl*) (*Tl* $ *Tl*), as the following clearly holds: (*IfNil Hd* (*Hd* $ *Tl*) (*Tl* $ *Tl*)) $ (*x1 # x2 # x3*) = *IfNil x1 x2 x3*. We can define an operation, *replaceAt*, which for a given tree position (represented with a list of selectors *pos*) and two normal-form expressions, generates a new expression, which has the result of the second expression pushed at position *pos* in the result of the first expression.

```
Fixpoint replaceAt (pos: list Selector) (t trep: NTrm) {struct pos}: NTrm :=
  match pos with
    | nil ⇒ trep
    | sel::sels ⇒ match sel with
      | HD ⇒ NCons (replaceAt sels (normSelNCmp HD t) trep)
                   (normSelNCmp TL t)
      | TL ⇒ NCons (normSelNCmp HD t)
                   (replaceAt sels (normSelNCmp TL t) trep)
    end
  end.
```

The action of this function is best illustrated with a couple of examples. If we have 2 values packed in a pair as input – say *x1 # x2* – we can fix the value of *x1* to *Nil # Nil* in the following way:

```
Eval compute in (ntrm2trm (replaceAt (HD::nil) (normConv Id) (normConv
(Nil # Nil)))).
```

```
= (Nil # Nil) # Tl : Trm
```

We can substitute not only constant values but also arbitrary expressions with *replaceAt* and *normNCmp*. If we consider again the expression *IfNil x1 x2 x3* with the given encoding of variables (*x1 # x2 # x3*), we can substitute *Tl $ Hd $ Tl # Hd $ Hd $ Tl* for *x2* thusly:

```
Eval compute in (let nt1 := normConv (IfNil Hd (Hd $ Tl) (Tl $ Tl)) in
  let nt2 := normConv (Tl $ Hd $ Tl # Hd $ Hd $ Tl) in
  ntrm2trm (normNCmp nt1 (replaceAt (TL::HD::nil) (normConv Id) nt2))).

  = IfNil Hd (Tl $ Hd $ Tl # Hd $ Hd $ Tl) (Tl $ Tl) : Trm
```

We now establish some properties of *replaceAt* that will prove useful later.

```
Lemma replaceAt_id: ∀ sels: list Selector, ∀ t trep: NTrm,
  normSelsNCmp sels (replaceAt sels t trep) = trep.

Lemma replaceAt_app: ∀ (sels1 sels2: list Selector) (nt ntrepl: NTrm),
  replaceAt (sels1 ++ sels2) nt ntrepl
  = replaceAt sels1 nt (replaceAt sels2 (normSelsNCmp sels1 nt) ntrepl).
```

For the next property, we need to compute the common prefix and the different suffixes of 2 lists. We shall need also to compute equivalence of selectors.

```
Definition Sel_eq_dec (sel1 sel2: Selector) : {sel1 = sel2} + {sel1 ≠ sel2}.
  decide equality.
Defined.
```

This is just a nice trick to let Coq deduce the equality predicate for us. The type $\{sel1 = sel2\} + \{sel1 \neq sel2\}$ is a sum type than not only gives the outcome of the test, but also contains a proof of the corresponding equality/inequality. For simplicity, we can cast this result to a simple *bool*, using the fact that Coq if expressions apply generically to any type with 2 constructors:

```
Definition eqSel sel1 sel2 := if Sel_eq_dec sel1 sel2 then true else false.

Lemma eqSel_reflx: ∀ sel, eqSel sel sel = true.

Fixpoint commonPrefix (X: Set) (eqX: X → X → bool) (l1 l2: list X)
  {struct l1} : (list X) × (list X) × (list X) := match l1, l2 with
  | nil, _ ⇒ (nil, nil, l2)
  | _, nil ⇒ (nil, l1, nil)
  | x::xs, y::ys ⇒ if eqX x y then
      let cp := commonPrefix X eqX xs ys in let pr := fst (fst cp) in
      let l1a := snd (fst cp) in let l2a := snd cp in (x::pr, l1a, l2a)
    else (nil, l1, l2)
  end.
```

```
Lemma commonPrefix_X_XappY: ∀ X: Set, ∀ eqX: X → X → bool,
  (∀ x: X, eqX x x = true) → ∀ xs ys: list X,
  commonPrefix X eqX xs (xs ++ ys) = (xs, nil, ys).

Lemma normSelsNCmp_ReplaceAt: ∀ (sels1 sels2: list Selector),
  ∀ (nt ntrepl: NTrm), normSelsNCmp sels1 (replaceAt sels2 nt ntrepl) =
  let cp := commonPrefix _ eqSel sels1 sels2 in let csels := fst (fst cp) in
  let usels1 := snd (fst cp) in let usels2 := snd cp in
  normSelsNCmp usels1 (replaceAt usels2 (normSelsNCmp csels nt) ntrepl).

Lemma replaceAt_NSelCmp: ∀ (sels1 sels2: list Selector) (nt: NTrm),
  replaceAt sels1 (NSelCmp sels2) nt
  = normSelsNCmp sels2 (replaceAt (sels2 ++ sels1) (NSelCmp nil) nt).
```

2.3 Positive Information Propagation

We can use object-level substitution, as implemented by *replaceAt* and *normNCmp*, to propagate information about the test result inside the branches of a conditional expressions. This transformation is one of the key differences that distinguish supercompilation from weaker optimizations like classical partial evaluation and deforestation [5,18]. The definition is greatly simplified by the fact that normal-form tests can only take the form of selector compositions.

```
Definition setNilAt (sels: list Selector): NTrm :=
  replaceAt sels (NSelCmp nil) NNil.

Definition setConsAt (sels: list Selector) : NTrm :=
  replaceAt sels (NSelCmp nil)
    (NCons (NSelCmp (sels ++ HD::nil)) (NSelCmp (sels ++ TL::nil))).
```

Once we have an expression encoding the substitution of the test result, what remains is to compose it with the corresponding if-branch, as in our case substitution composition is replaced by simple function composition.

```
Fixpoint propagateIfCond (nt: NTrm) {struct nt} : NTrm :=
  match nt with
  | NCons nt1 nt2 ⇒ NCons (propagateIfCond nt1) (propagateIfCond nt2)
  | NIfNil sels nt1 nt2 ⇒
    let nt1a := propagateIfCond nt1 in let nt2a := propagateIfCond nt2 in
    let nt1b := normNCmp nt1a (setNilAt sels) in
    let nt2b := normNCmp nt2a (setConsAt sels) in NIfNil sels nt1b nt2b
  | _ ⇒ nt
  end.
```

Establishing the correctness of *propagateIfCond* is once again decomposed into a sequence of lemmas.

```
Lemma setNilAtPreservesEvalAux: ∀ (sels1 sels2: list Selector),
```

```
  replaceAt sels1 (NSelCmp sels2) NNil
  = normNCmpSels (replaceAt sels1 (NSelCmp nil) NNil) sels2.

Lemma setConsAtPreservesEvalAux: ∀ (sels1 sels2: list Selector),
  replaceAt sels1 (NSelCmp sels2) (NCons (NSelCmp
    (sels2++sels1++HD::nil)) (NSelCmp (sels2++sels1++TL::nil)))
  = normNCmpSels (replaceAt sels1 (NSelCmp nil) (NCons
    (NSelCmp (sels1++HD::nil)) (NSelCmp (sels1++TL::nil)))) sels2.

Lemma setNilAtPreservesEvalAux2: ∀ (v: Val), ∀ (sels1 sels2: list Selector),
  evalSels sels1 (evalNT (setNilAt (sels1++sels2)) v)
  = evalNT (setNilAt sels2) (evalSels sels1 v).

Lemma setConsAtPreservesEvalAux2: ∀ (v: Val), ∀ (sels1 sels2: list Selector),
  evalSels sels1 (evalNT (setConsAt (sels1++sels2)) v)
  = evalNT (setConsAt sels2) (evalSels sels1 v).

Lemma setNilAtPreservesEval: ∀ sels: list Selector, ∀ v: Val,
  evalSels sels v = VNil → evalNT (setNilAt sels) v = v.

Lemma setConsAtPreservesEval: ∀ sels: list Selector, ∀ v v1 v2: Val,
  evalSels sels v = VCons v1 v2 → evalNT (setConsAt sels) v = v.

Lemma condPropagatorsPreserveEval: ∀ (sels: list Selector) (nt1 nt2: NTrm),
  ∀ (v: Val), evalNT (NIfNil sels (normNCmp nt1 (setNilAt sels))
    (normNCmp nt2 (setConsAt sels))) v = evalNT (NIfNil sels nt1 nt2) v.
```

The proofs of these lemmas involve some tricky rewrites, using the established properties of *replaceAt*. Details can be found in the actual Coq sources. The main theorem can now be proved easily by induction, using the last lemma *condPropagatorsPreserveEval*.

```
Theorem propagateIfCondPreservesEval: ∀ nt: NTrm, ∀ v: Val,
  evalNT (propagateIfCond nt) v = evalNT nt v.
```

We can combine the first two stages – normalization and positive information propagation – into a single function, and trivially establish its correctness.

```
Definition norm (t: Trm) := propagateIfCond (normConv t).

Theorem normPreservesEval: ∀ t v, evalNT (norm t) v = evalT t v.
```

Recalling the example from the introduction, we can see that *norm* also eliminates redundant tests, besides other reductions:

```
Eval compute in (ntrm2trm (norm (IfNil Hd (Nil # Nil) (IfNil Hd Nil Tl)))).

  = IfNil Hd (Nil # Nil) Tl : Trm
```

3 A Turing-complete Imperative Language

While our simple expression language has helped us to successfully study some key aspects of supercompilation, it is obvious that we cannot write many interesting programs in it. Not only it is far from being Turing-complete, but it even lacks full-blown primitive recursion. However, we can build upon this language to obtain a larger, Turing-complete one. For example, we can embed the language of simple expressions inside a small imperative language with assignments and while-loops (called here SWhile):

```
Inductive SWhileStmt: Set :=
  | Assign: Trm → SWhileStmt
  | Seq: SWhileStmt → SWhileStmt → SWhileStmt
  | While: Trm → SWhileStmt → SWhileStmt.
```

As a further simplification, we assume that the language has a single variable, similar to other research languages like I and LOOP [8,7]. This variable is implicitly used in assignments and while tests. As this language is Turing-complete, we cannot specify its evaluator directly as a total Coq function, like we did for the language of simple expressions. We can specify its semantics as a logical relation, which is encoded in Coq as a (dependent) inductive family living in `Prop`:

```
Inductive SWhileEvalRel: Val → SWhileStmt → Val → Prop :=
  | SWhileEvalAssign: ∀ e v, SWhileEvalRel v (Assign e) (evalT e v)
  | SWhileEvalSeq: ∀ st1 st2 v1 v2 v3,
      SWhileEvalRel v1 st1 v2 → SWhileEvalRel v2 st2 v3 →
      SWhileEvalRel v1 (Seq st1 st2) v3
  | SWhileEvalWhileNil: ∀ cond st v,
      evalT cond v = VNil → SWhileEvalRel v (While cond st) v
  | SWhileEvalWhileBottom: ∀ cond st v,
      evalT cond v = VBottom → SWhileEvalRel v (While cond st) VBottom
  | SWhileEvalWhileCons: ∀ cond st v1 v2 v3 vh vt,
      evalT cond v1 = VCons vh vt → SWhileEvalRel v1 st v2 →
      SWhileEvalRel v2 (While cond st) v3 →
      SWhileEvalRel v1 (While cond st) v3.
```

We can further simplify our task, by considering only programs containing a single while loop. This can be seen as an analog of Kleene's normal form (KNF) from recursion theory, and there are well-known proofs (not repeated here) that limiting ourselves to a single while loop implies no loss of generality [6]. The "Kleene normal form" analog for SWhile programs can be represented as a record of 4 simple expressions:

```
Record KNFProg : Set := MkKNFProg {
  initExp: Trm; condExp: Trm; bodyExp: Trm; finalExp: Trm }.
```

The meaning is obvious by the injection into the full syntax of SWhile programs:

```
Definition KNFtoProg knf :=
  Seq (Assign (initExp knf))
  (Seq (While (condExp knf) (Assign (bodyExp knf)))
    (Assign (finalExp knf))).
```

We can introduce a bit of syntactic sugar for SWhile constructs (at the expense of a conflict with the `Record` syntax).

```
Infix ";" := Seq (at level 65, right associativity).
Notation "'VAR' '←' e" := (Assign e) (at level 64).
Notation "'WHILE' cond 'DO' body 'DONE'" := (While cond body) (at level
0).
```

As a simple example, here is a program that reverses its input (assuming the usual Lisp encoding of lists as binary trees).

```
Definition revList_knf := MkKNFProg
  (Id # Nil) Hd (Tl $ Hd # Hd $ Hd # Tl) Tl.
Eval compute in (KNFtoProg revList_knf).
```

```
= VAR  <- Id # Nil;
  WHILE Hd DO VAR  <- Tl $ Hd # Hd $ Hd # Tl DONE;
  VAR  <- Tl : SWhileStmt
```

We see here one important drawback of the simplifications we introduced: our language is very difficult to program in, and very unreadable. To make the meaning of the code clearer, we can rewrite it by hand to a version of SWhile with many variables; in our case 2 suffice – *input* and *output*:

```
  output  <- Nil;
  WHILE input DO
    (input # output) <- (Tl $ input) # (Hd $ input # output) DONE;
```

While the abstract syntax of SWhile permits arbitrary expressions as while-loop conditions, many optimizing transformations that follow are valid only if the condition of the loop is strict, according to the following definition:

```
Definition strictTrm (t: Trm) := evalT t VBottom = VBottom.
```

We can easily see that strictness for our expression language amounts to a simple syntactic check on the normal form of the expression:

```
Lemma strictTrm_SyntaxCriterion: ∀ (t: Trm), strictTrm t ↔
  (match normConv t with | NNil | NCons _ _ ⇒ false | _ ⇒ true end) = true.
```

So it is obviously reasonable to consider only programs with strict loop conditions as otherwise the loop degenerates to either an infinite or an empty one.

While the relational specification of SWhile semantics is elegant, it is not executable (at least not inside Coq). We can build an approximation to an evaluation function in Coq itself, using a standard trick for modeling partial functions – we add an extra parameter limiting the recursion depth, and the definition of the evaluation function can be done by structural recursion on that new parameter. We do so only for the KNF special case.

```
Fixpoint evalKNFCore (d: nat) (cond e: Trm) (v: Val) {struct d}
  : option Val := match d with
  | O ⇒ None
  | S d' ⇒ match evalT cond v with
    | VNil ⇒ Some v
    | VBottom ⇒ Some VBottom
    | VCons _ _ ⇒ evalKNFCore d' cond e (evalT e v)
    end
  end.

Definition evalKNF (d: nat) (knf: KNFProg) (v: Val) : option Val :=
  match evalKNFCore d (condExp knf) (bodyExp knf)
   (evalT (initExp knf) v) with
  | None ⇒ None
  | Some v ⇒ Some (evalT (finalExp knf) v)
  end.
```

We can now execute the example program above on some input:

```
Definition listToVal vs := fold_right VCons VNil vs.
Eval vm_compute in (evalKNF 3 revList_knf
  (listToVal (VNil::(VCons VNil VNil)::nil))).

  = Some (VCons (VCons VNil VNil) (VCons VNil VNil)) : option Val
```

In order to verify that the executable interpreter is correct with respect to the relational semantics given above, we first establish, that the evaluation of the loop by *evalKNFCore* respects the semantics, and then we prove the correctness of the main evaluation function – *evalKNF*

```
Lemma evalKNFCore_SWhileEvalRel: ∀ cond e v1 v2,
  SWhileEvalRel v1 (While cond (Assign e)) v2 ↔
  ∃ d: nat, evalKNFCore d cond e v1 = Some v2.

Theorem evalKNF_SWhileEvalRel: ∀ knf v1 v2,
  SWhileEvalRel v1 (KNFtoProg knf) v2 ↔
  ∃ d: nat, evalKNF d knf v1 = Some v2.
```

3.1 Loop Unrolling

The principal additional optimization that we can perform on loop programs – on the top of the already existing optimizations for the expression sub-language – is loop unrolling. We can study different forms of while-loop unrolling; here we shall limit ourselves to one simple form of unrolling – trying to execute the body of the loop once before entering the loop itself, provided the condition of the loop holds. Of course, we cannot expect spectacular optimizations from this form of unrolling; in the very least, it leaves the loop itself unmodified. It is sufficient, however, to demonstrate the power of positive information propagation in some simple cases. Later in the paper we discuss possibilities for more powerful forms of loop unrolling.

```
Definition unrollToInit knf := let nrm t := ntrm2trm (norm t) in
  let newInit := nrm ((IfNil (condExp knf) Id (bodyExp knf)) $ (initExp knf))
  in MkKNFProg newInit (condExp knf) (bodyExp knf) (finalExp knf).
```

We can verify that unrolling the loop once respects the semantics. It turns easier to use *evalKNFCore* and *evalKNF* as semantics specifications; it is OK as we have already verified that they are faithful to the original specification by a logical relation.

```
Lemma normPreservesEval': ∀ t v, evalT (ntrm2trm (norm t)) v = evalT t v.

Lemma evalKNFCore_Bottom: ∀ d cond e v, strictTrm cond →
  evalKNFCore d cond e VBottom = Some v → v = VBottom.

Lemma evalKNFCore_unrollToInit_fw: ∀ d knf v1 v2, strictTrm (condExp knf)
  → evalKNFCore d (condExp knf) (bodyExp knf) v1 = Some v2 →
  ∃ d2: nat, evalKNFCore d2 (condExp knf) (bodyExp knf) (evalT
    (IfNil (condExp knf) Id (bodyExp knf)) v1) = Some v2.

Lemma evalKNF_unrollToInit_fw: ∀ d knf v1 v2, strictTrm (condExp knf)
  → evalKNF d knf v1 = Some v2 →
  ∃ d2: nat, evalKNF d2 (unrollToInit knf) v1 = Some v2.

Lemma evalKNFCore_unrollToInit_bw: ∀ d knf v1 v2, strictTrm (condExp knf)
  → evalKNFCore d (condExp knf) (bodyExp knf) (evalT
    (IfNil (condExp knf) Id (bodyExp knf)) v1) = Some v2 →
  ∃ d2: nat, evalKNFCore d2 (condExp knf) (bodyExp knf) v1 = Some v2.

Lemma evalKNF_unrollToInit_bw: ∀ d knf v1 v2, strictTrm (condExp knf)
  → evalKNF d (unrollToInit knf) v1 = Some v2 →
  ∃ d2: nat, evalKNF d2 knf v1 = Some v2.

Theorem evalKNF_unrollToInit: ∀ knf v1 v2, strictTrm (condExp knf) →
  ((∃ d: nat, evalKNF d knf v1 = Some v2) ↔
   (∃ d2: nat, evalKNF d2 (unrollToInit knf) v1 = Some v2)).
```

To deal with repeated unrollings and to lay the background for termination verification of the whole supercompiler, we need streams (infinite sequences). A simple function-based definition suffices for our purposes.

```
Definition Stream A := nat → A .
```

We define a couple of basic operations on streams – the well-known *map* and *unfold* from the functional programming repertoire.

```
Definition streamMap A B (f: A → B) (s: Stream A) : Stream B :=
  fun n ⇒ f (s n).

Definition streamUnfold X (seed: X) (f: X → X) : Stream X :=
  fix streamUnfold' (n: nat) {struct n} : X := match n with
  | O ⇒ seed | S n' ⇒ f (streamUnfold' n') end.
```

3.2 Homeomorphic Embedding for Ensuring Termination

The so-called "whistle" of our supercompiler uses the now-standard approach of relying on homeomorphic embedding and the Kruskal's tree theorem [17] to ensure termination of the process. To formulate this theorem in its general form, we introduce a type of arbitrary first-order terms. The Coq `Section` mechanism allows to specify only once parameters common for a whole set of definitions – in our case the types for term variables and function symbols, as well as the fact that function symbols have decidable equality. (Variables of first-order terms typically also have decidable equality, but it is not needed in the current development.)

```
Section FOTerms.

Variable V: Set. Variable F: Set.
Variable F_eq_dec: ∀ f g: F, {f = g} + {f ≠ g}.
```

We adopt a slightly non-standard definition of first-order terms, which is however easier to work with in Coq:

```
Inductive FOTerm : Set :=
  | FOVar: V → FOTerm
  | FOFun0: option F → FOTerm
  | FOFun2: option F → FOTerm → FOTerm → FOTerm.

Definition optionF_eq_dec (f1 f2: option F): {f1 = f2} + {f1 ≠ f2}.
  decide equality.
Defined.
```

Even with this definition of first-order terms, defining an executable version of homeomorphic embedding in Coq is a little tricky – we need two nested structural recursions, like in the case of *normNCmp*.

```
Definition homemb (t1 t2: FOTerm) : bool :=
```

```
  (fix homemb1 (t1: FOTerm): FOTerm → bool :=
  (fix homemb2 (t2: FOTerm): bool :=
  match t1 with
  | FOVar _ ⇒ match t2 with | FOVar _ ⇒ true | _ ⇒ false end
  | FOFun0 f1 ⇒ match t2 with
    | FOFun0 f2 ⇒ if optionF_eq_dec f1 f2 then true else false
    | FOFun2 _ t21 t22 ⇒ orb (homemb2 t21) (homemb2 t22)
    | _ ⇒ false
    end
  | FOFun2 f1 t11 t12 ⇒ match t2 with
    | FOFun2 f2 t21 t22 ⇒ orb (if optionF_eq_dec f1 f2
        then andb (homemb1 t11 t21) (homemb1 t12 t22)
        else false) (orb (homemb2 t21) (homemb2 t22))
    | _ ⇒ false
    end
  end
  )) t1 t2.
```

We can now give a formulation of Kruskal's theorem. It is beyond the scope of the current work to give a formal proof of this result, so we just take it as an assumption.

```
Theorem Kruskal: ∀ s: Stream FOTerm,
  ∃ i: nat, ∃ j: nat, i < j ∧ homemb (s i) (s j) = true.
Admitted.

End FOTerms.
```

We mark some arguments as implicit so that they are inferred by the Coq typechecker.

```
Implicit Arguments FOVar [V F]. Implicit Arguments FOFun0 [V F].
Implicit Arguments FOFun2 [V F]. Implicit Arguments homemb [V F].
```

To use Kruskal's theorem for online termination, we need a few additional ingredients. Firstly, a function that actually computes (the index of) the first of the two terms in a sequence, that are related by homeomorphic embedding. For simplicity, we limit the search to a finite initial fragment of the sequence and prove separately that there is always such initial fragment that will produce results.

```
Definition isNthEmbeddedBelow V F fn_eq_dec (n m: nat)
  (s: Stream (FOTerm V F)) : bool :=
  existsb (fun i ⇒ homemb fn_eq_dec (s n) (s i)) (seq (S n) (m - n)).
Implicit Arguments isNthEmbeddedBelow [V F].

Definition firstEmbedded V F fn_eq_dec (n: nat) (s: Stream (FOTerm V F))
: option nat :=
```

```
  find (fun i ⇒ isNthEmbeddedBelow fn_eq_dec i n s) (seq 0 n).
Implicit Arguments firstEmbedded [V F].
```

We use list functions from the standard library, like *existsb*, *find*, *seq*, with hopefully obvious meanings. Some of their useful properties are missing from the library, and we have to prove them first:

```
Lemma find_Some: ∀ X (f: X → bool) x xs,
  In x xs → f x = true → ∃ y, find f xs = Some y.

Lemma In_seq: ∀ n m l, In n (seq m l) ↔ m ≤ n < m + l.
```

With these properties in addition to Kruskal's theorem, we easily establish that *firstEmbedded* is total.

```
Theorem firstEmbedded_total: ∀ V F F_eq_dec (s: Stream (FOTerm V F)),
  ∃ n, ∃ m, firstEmbedded F_eq_dec n s = Some m.
```

Another helper function we need is an injection from simple expressions into first-order terms. We first define an enumeration of the constructors of expressions, together with their decidable equality predicate. Then the definition of the injection is straightforward.

```
Inductive TrmCons: Set := | TCNil | TCCons | TCSelHd
  | TCSelTl | TCId | TCCmp | TCIfNil | TCBottom.

Definition TrmCons_eq_dec (t1 t2: TrmCons) : {t1 = t2} + {t1 ≠ t2}.
  decide equality.
Defined.

Fixpoint TrmToFOTerm (e: Trm) : FOTerm Empty_set TrmCons :=
  match e with
  | Nil ⇒ FOFun0 (Some TCNil)
  | Cons e1 e2 ⇒ FOFun2 (Some TCCons)
      (TrmToFOTerm e1) (TrmToFOTerm e2)
  | Sel sel ⇒ if sel then FOFun0 (Some TCSelHd)
       else FOFun0 (Some TCSelTl)
  | Id ⇒ FOFun0 (Some TCId)
  | Cmp e1 e2 ⇒ FOFun2 (Some TCCmp)
      (TrmToFOTerm e1) (TrmToFOTerm e2)
  | IfNil e1 e2 e3 ⇒ FOFun2 (Some TCIfNil) (TrmToFOTerm e1)
      (FOFun2 (Some TCCons) (TrmToFOTerm e2) (TrmToFOTerm e3))
  | Bottom ⇒ FOFun0 (Some TCBottom)
  end.
```

3.3 Simple Supercompiler using Loop Unrolling

Now we can assemble all previously defined components into a finished basic supercompiler. It first builds a stream of iterated unrollings of the program in

KNF. Then it looks for pairs of initializer expressions related by homeomorphic embedding in an initial fragment of the stream (the length of this fragment being specified by an input parameter – n). We use only initializer expressions when checking for termination, because they are the only KNF part changed by the simple loop unrolling used here. To aid the experimentations on practical examples, there is also an input option, *alwaysSome*, which can be used to force a result even if no homeomorphic embedding is found in the specified initial stream segment.

```
Definition sscpCore (alwaysSome: bool) unroll knf2trm n (knf: KNFProg) :=
  let knfs := streamUnfold knf unroll in
  let ts := streamMap (fun knf ⇒ TrmToFOTerm (knf2trm knf)) knfs in
  match firstEmbedded TrmCons_eq_dec n ts with
  | None ⇒ if alwaysSome then Some (knfs n) else None
  | Some m ⇒ Some (knfs m)
  end.

Definition sscp (alwaysSome: bool) n knf :=
  sscpCore alwaysSome unrollToInit initExp n knf.
```

Alternatively, we define a cut-down version, which uses *normConv* instead of *norm* during loop unrolling. In essence it is a simple deforestation analog of the simple supercompiler above:

```
Definition unrollToInit' knf :=
  let nrm t := ntrm2trm (normConv t) in
  let newInit := nrm ((IfNil (condExp knf) Id (bodyExp knf)) $ (initExp knf))
  in MkKNFProg newInit (condExp knf) (bodyExp knf) (finalExp knf).

Definition sscp' (alwaysSome: bool) n knf :=
  sscpCore alwaysSome unrollToInit' initExp n knf.
```

Now we can see both methods at work, demonstrating the usefulness of even this limited form of supercompilation. Consider again the usual Lisp-like encoding of booleans and lists in the domain of binary trees. The task of checking if an input list of booleans contains at least one false value can be performed by the following program:

```
Definition listHasWFalse_knf :=
  let WFalse := Nil in let WTrue := Nil # Nil in MkKNFProg
  (Id # WFalse) Hd (IfNil (Hd $ Hd) (Nil # WTrue) ((Tl $ Hd) # Tl)) Tl.

Eval compute in (KNFtoProg listHasWFalse_knf).
```

```
= VAR  <- Id # Nil;
  WHILE Hd DO
    VAR  <- IfNil (Hd $ Hd) (Nil # Nil # Nil) (Tl $ Hd # Tl)
  DONE;
  VAR  <- Tl : SWhileStmt
```

A few explanations are in order. We extend the computation state with a flag to hold the final result – at position *Tl* – while keeping the original input list at position *Hd*. Then we loop while the list is not empty, and test its head. If it is *VNil*, we make the list empty to force an exit of the loop, and set the result to true, otherwise we remove the list head and continue.

Next, we introduce a specialized version of this program, which, if the input list is not empty, adds a negated copy of the head of the list. The idea is clearly that this specialized version should return true on all non-empty lists, and false only on the empty list.

```
Definition modifyKNFinput knf modifierExp := MkKNFProg
  ((initExp knf) $ modifierExp) (condExp knf) (bodyExp knf) (finalExp knf).

Definition listHasWFalse_knf_negdupHd :=
  let WFalse := Nil in let WTrue := Nil # Nil in
  let negate x := IfNil x WTrue WFalse in
  modifyKNFinput listHasWFalse_knf (IfNil Id Id (negate Hd # Id)).

Eval vm_compute in (match sscp false 3 listHasWFalse_knf_negdupHd with
  | Some knf ⇒ Some (KNFtoProg knf) | None ⇒ None end).

  = Some (VAR <- IfNil Id (Nil # Nil)
            (IfNil Hd (Nil # Nil # Nil) (Nil # Nil # Nil));
          WHILE Hd
          DO VAR  <- IfNil (Hd $ Hd)
            (Nil # Nil # Nil) (Tl $ Hd # Tl)
          DONE; VAR  <- Tl) : option SWhileStmt
```

While the resulting program may not look simplified at first, if we remove by hand the second if-expression with equal branches, we can see that the loop will never be entered. The final correct result will be computed directly by the initializer expression. The combination of deforestation, positive information propagation and simple loop unrolling has resulted in an almost optimal specialized program in this case.

```
  = Some (VAR <- IfNil Id (Nil # Nil) (Nil # Nil # Nil);
          WHILE Hd
          DO VAR  <- IfNil (Hd $ Hd)
            (Nil # Nil # Nil) (Tl $ Hd # Tl)
          DONE; VAR  <- Tl) : option SWhileStmt
```

In contrast, if we remove just positive information propagation from the mix, the end result is much less satisfactory:

```
Eval vm_compute in (match sscp' false 2 listHasWFalse_knf_negdupHd with
  | Some knf ⇒ Some (KNFtoProg knf) | None ⇒ None end).

  = Some (VAR  <- IfNil Id
           (IfNil Id (IfNil Id Id (IfNil Hd (Nil # Nil) Nil # Id) # Nil)
```

```
    (IfNil Id (IfNil Hd (Nil # Nil # Nil) (IfNil Id Tl Id # Nil))
       (IfNil Hd (IfNil Id Tl Id # Nil) (Nil # Nil # Nil))))
  (IfNil Id (IfNil Hd (Nil # Nil # Nil) (IfNil Id Tl Id # Nil))
     (IfNil Hd (IfNil Id Tl Id # Nil) (Nil # Nil # Nil)));
WHILE Hd
DO VAR  <- IfNil (Hd $ Hd) (Nil # Nil # Nil) (Tl $ Hd # Tl) DONE;
VAR  <- Tl) : option SWhileStmt
```

3.4 Proof of Correctness of the Full Supercompiler

We consider two aspects of supercompiler correctness - totality and preservation of semantics. Totality of the supercompiler function is a direct consequence of totality of *firstEmbedded* (`Theorem` *firstEmbedded_total*).

`Lemma` *sscpCore_total*: ∀ *b unroll knf2trm knf*, ∃ *n*, ∃ *knf1*,
sscpCore b unroll knf2trm n knf = *Some knf1*.

`Theorem` *sscp_total*: ∀ *b knf*, ∃ *n*, ∃ *knf1*, *sscp b n knf* = *Some knf1*.

Preservation of semantics, on the other hand, is established through a sequence of lemmas, essentially relying only on correctness of one-step loop unrolling (*evalKNF_unrollToInit*). We can say that we have achieved one of the main goals of this study - maximum modularity in proving different aspects of supercompiler correctness.

`Lemma` *condExp_unrollToInitStream*: ∀ *knf n*,
condExp (*streamUnfold knf unrollToInit n*) = *condExp knf*.

`Lemma` *unrollToInitStream_evalKNF_fw*: ∀ *knf v1 v2 n d1*,
strictTrm (*condExp knf*) → *evalKNF d1 knf v1* = *Some v2* →
∃ *d2*, *evalKNF d2* (*streamUnfold knf unrollToInit n*) *v1* = *Some v2*.

`Lemma` *unrollToInitStream_evalKNF_bw*: ∀ *knf v1 v2 n d1*,
strictTrm (*condExp knf*) → *evalKNF d1* (*streamUnfold knf unrollToInit n*)
v1 = *Some v2* → ∃ *d2*, *evalKNF d2 knf v1* = *Some v2*.

`Lemma` *sscpCore_correct_fw*: ∀ *b knf knf1 n d1 v1 v2*, *strictTrm* (*condExp knf*)
→ *sscpCore b unrollToInit initExp n knf* = *Some knf1* →
evalKNF d1 knf v1 = *Some v2* → ∃ *d2*, *evalKNF d2 knf1 v1* = *Some v2*.

`Lemma` *sscpCore_correct_bw*: ∀ *b knf knf1 n d1 v1 v2*, *strictTrm* (*condExp knf*)
→ *sscpCore b unrollToInit initExp n knf* = *Some knf1* →
evalKNF d1 knf1 v1 = *Some v2* → ∃ *d2*, *evalKNF d2 knf v1* = *Some v2*.

`Lemma` *sscpCore_correct*: ∀ *b knf knf1 n v1 v2*, *strictTrm* (*condExp knf*)
→ *sscpCore b unrollToInit initExp n knf* = *Some knf1* →
((∃ *d1*, *evalKNF d1 knf v1* = *Some v2*) ↔

(∃ *d2*, *evalKNF d2 knf1 v1* = *Some v2*)).

`Theorem` *sscp_correct*: ∀ *b knf knf1 n v1 v2*,
strictTrm (*condExp knf*) → *sscp b n knf* = *Some knf1* →
((∃ *d1*, *evalKNF d1 knf v1* = *Some v2*) ↔
(∃ *d2*, *evalKNF d2 knf1 v1* = *Some v2*)).

4 Related Work

Since Turchin's ground-breaking work on supercompilation of Refal has gained popularity [21], a number of supercompilers have appeared for different languages ([5,17,18,3,13], to mention just a few). The supercompiler described in this work is most closely related to the formulation of positive supercompilation by Sørensen at al [18]. In contrast to other treatments of supercompilation, which typically use either substitutions or environments on the meta-level in order to propagate information in conditional branches, we use a form of object-level explicit substitutions. Explicit substitutions (introduced by Abadi et al [1]) are by no means a new technique, and have been used, with varying details, in many other contexts. In the context of supercompilation, they were previously applied by the author in his PhD thesis [11], but not with the aim to simplify formal proofs of correctness.

Studies have been published on general frameworks for proving semantics preservation and termination of supercompilers and similar program transformers - such as [15,16]. To the best of our knowledge, there has been no previous work on formally verifying the correctness of a supercompiler implementation. At the same time, as numerous formal proof assistants grow mature, we see more and more computer-checked proofs of correctness for practical systems. We have already mentioned two impressive examples – the Compcert compiler from a large subset of C to a real assembler language [12], and the seL4 operating system microkernel [10].

Many treatments of supercompilation (and related studies like optimal self-application) use either a small subset of an existing language (like Core Haskell [13] or FlatCurry [3]), or a tiny language operating on Lisp-like well-founded binary trees. Languages like I/LOOP [8,7] and S-Graph/TSG [5,2] were an important source of inspiration for the development of SWhile. Non-Turing-complete languages have long been a subject of research in computability and computational complexity theory (e.g. [9]). Our language of simple expressions can, in particular, be seen as a generalization – from the domain of natural numbers to the domain of binary trees – of the language of "simple programs" of Tsichritzis [20]. The explicit use of a non-Turing-complete language to formulate parts of the driving process in supercompilation appears to be a new result of this work.

The standard formulation of positive supercompilation distinguishes 4 phases - driving (including unfolding), "whistle", folding, and generalization [18]. The separation of unfolding from the rest of the driving transformations, and their splitting in two parts (normalization + positive information propagation) is a

new result, although its roots can be traced to the author's previous work [11]. As for the definition of normalization itself, similar transformations have been used in many different contexts (e.g. [7]), unrelated to supercompilation.

5 Conclusions and Future Work

We have achieved a full formal verification, in Coq, of a greatly simplified supercompiler for a basic imperative language operating on binary-tree data. To the best of our knowledge, this is the first attempt of a computer-checked proof of correctness of a supercompiler or a similar transformer. An advantage of our method is that it leads to a small-size formalization and verification source – about 1100 non-empty, non-comment lines of Coq code[3], of which about 45% are definitions, and the rest are proofs. As a comparison, the formal verification of the Compcert compiler amounts to about 42000 lines of Coq code [12].

As another advantage, our verification is organized in a very modular way, thanks to a new refactoring of the supercompilation process into smaller pieces that can be checked independently. We believe that this modularity makes the approach more re-usable in different contexts.

There are a couple of important directions for future improvements. Firstly, our object language is so simple that it is hard to read and very hard to program in. While this is normal for a toy language crafted for research purposes, the effort to improve its usability may be worthwhile. Making the language easier to program in is the simpler task – we can always add arbitrary amount of syntax sugar in a preprocessing phase, or even a compiler from a higher-level language to SWhile. The more challenging task is to make the program resulting from supercompilation easier to understand. Some form of post-processing transformations may help, but probably will not be sufficient by themselves.

Secondly, our supercompiler is crippled in its current form, due to its very limited definition of loop unrolling. We have tried some other, seemingly more powerful forms of unrolling, omitted from this text. Unfortunately the preliminary experimentation on simple examples does not show good results. Further research is needed to find a more powerful form of loop unrolling, if we want to pass the "KMP test" [5,18]. Another interesting option to pursue is to switch from an imperative language with a single while loop to a functional one with a single recursive function (like the language F of N. D. Jones [8]). This switch might also be beneficial for the readability of the resulting programs. A challenge for this approach would be to keep a clean separation between unfolding and the other parts of driving, but it appears feasible.

Beside the improvements of the method suggested above, some practical applications might be interesting to study. For example, we could re-use parts of the current development as new proof tactics in Coq, using the mechanism of proof by reflection [4]. Another idea is to re-use the decomposition of supercompilation in smaller parts, that has helped the current development, for repeating

[3] as reported by the coqwc tool

the verification in another proof system, possibly one based on supercompilation itself.

6 Acknowledgments

The author would like to express his gratitude to the four anonymous referees, to Prof. Iwan Tabakow and to Prof. Andrei Klimov, whose comments and suggestions helped improve the article. This study would not be possible without the great work of the team developing Coq [19]. The author found both the introductions of B. C. Pierce et al [14] and A. Chlipala [4] invaluable in learning Coq and automated theorem proving in general.

References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991.
2. S. M. Abramov. *Metavychisleniya i ih primenenie (Metacomputation and its applications)*. Nauka, Moscow, 1995.
3. Elvira Albert, Michael Hanus, and Germán Vidal. A practical partial evaluation scheme for multi-paradigm declarative languages. *Journal of Functional and Logic Programming*, 2002, 2002.
4. Adam Chlipala. Certified programming with dependent types (book draft). `http://adam.chlipala.net/cpdt/`, 2010.
5. Robert Glück and Andrei V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Static Analysis. Proceedings*, volume 724 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 1993.
6. David Harel. On folk theorems. *Commun. ACM*, 23(7):379–389, 1980.
7. Lars Hartmann, Neil D. Jones, and Jakob Grue Simonsen. Interpretive overhead and optimal specialisation. or: Life without the pending list (workshop version). In V.F. Turchin, editor, *International Workshop on Metacomputation in Russia, Meta2008*, pages 1–12, 2008.
8. Neil D. Jones. *Computability and Complexity from a Programming Perspective*. Foundations of Computing. MIT Press, Boston, London, 1 edition, 1997.
9. Neil D. Jones. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming*, 11(1):55–94, january 2001. Special Issue on Functional Programming and Computational Complexity.
10. Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: seL4: formally verifying a high-performance microkernel. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 91–96. ACM, 2009.
11. Dimitur N. Krustev. Software test generation using program skeletons. PhD thesis Technical Report PRE 23/01, Wroclaw Polytechnic, Wroclaw, Poland, 2001.
12. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
13. Neil Mitchell and Colin Runciman. A supercompiler for core Haskell. In Olaf Chitil et al., editor, *IFL 2007*, volume 5083 of *LNCS*, pages 147–164. Springer-Verlag, May 2008.

14. Benjamin C. Pierce, Chris Casinghino, and Michael Greenberg. Software foundations (book draft). `http://www.cis.upenn.edu/~bcpierce/sf/`, 2010.
15. David Sands. Proving the correctness of recursion-based automatic program transformations. *Theor. Comput. Sci.*, 167(1&2):193–233, 1996.
16. Morten Heine Sørensen. Convergence of program transformers in the metric space of trees. In J. Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 315–337. Springer-Verlag, 1998.
17. Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.
18. Morten Heine Sørensen and Robert Glück. Introduction to supercompilation. In John Hatcliff, Torben Mogensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 246–270. Springer-Verlag, 1999.
19. Coq Development Team. The Coq proof assistant reference manual, version 8.2, February 2009.
20. Dennis Tsichritzis. The equivalence problem of simple programs. *J. ACM*, 17(4):729–738, 1970.
21. V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.
22. Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.