An Approach for Modular Verification of Multi-Result Supercompilers
(Work in Progress)

Dimitur Nikolaev Krustev

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

Abstract. Multi-result supercompilation is a recent and promising generalization of classical supercompilation. One of its goals is to permit easy construction of different supercompilers from a reusable top-level algorithm and independently implemented components for driving, folding, etc. The problem of preservation of semantics by multi-result supercompilers has not yet been studied in detail. So, while the implementation of a new multi-result supercompiler is simplified by the high degree of modularity, its verification is not. To alleviate this burden, we search for a set of sufficient conditions on the basic building blocks (such as driving or folding), which – if met – can be plugged into general theorems, to ensure supercompiler correctness. If the proposed approach proves successful, it will make multi-result supercompiler verification as easy and modular as the implementation itself.

1 Introduction

Multi-result supercompilation [10,11] is a recent generalization of classical supercompilation [18,19]. One of its key insights is to permit generalization to happen at any moment, and to consider and collect the different graphs of configurations arising from different choices about generalization. Recall that in classical supercompilation generalization is only applied when the whistle blows and folding is not possible. Paradoxically it turns out that in certain situations early generalization can lead to an optimal result, which cannot be obtained using the classical approach [8].

Another advantage of multi-result supercompilation (MRSC[1]) is that, from the beginning, it was designed in a modular way, as follows:

- a generic high-level algorithm, which is largely independent of the particular choice of object language;
- a small set of primitive operations, which encapsulate the language-specific parts of the supercompiler algorithm.

[1] The abbreviation *MRSC* is usually reserved for the original implementation in Scala, "The MRSC Toolkit". We take the liberty to use it also for multi-result supercompilation in general, for brevity.

This modularity allows the programmer to easily create supercompilers for different object languages, including highly specialized ones for particular DSLs [8]. A further refinement of modularity is present in a recent Agda formalization of MRSC [4], which is based on a more streamlined set of primitive operations. Moreover, the Agda formalization uses a "big-step" definition of MRSC + whistles based on inductive bars [3], which further simplify the main data structures and algorithms. The main goal of Grechanik et al. [4] is to formalize a more compact representation of the whole set of results produced by MRSC, to prove this representation correct w.r.t. the original representation (simple list of graphs), and to show that many useful operations (filtering the set of results, selecting an optimal result by certain criteria) can be directly and more efficiently performed on the compact representation. The formalization of object language semantics and verification of the preservation of this semantics by MRSC is beyond the scope of that paper.

As verification of semantics preservation by supercompilers is an interesting and practically useful topic in itself [12–14], the approach we describe here aims to fill this gap, and to propose a way for verifying semantics preservation of supercompilers based on big-step MRSC. We take the Agda formalization of Grechanik et al. [4] as a starting point (ported to Coq) and augment it with:

- a set of primitives for describing the semantics of the object language (Sec. 3)
- based on these primitives:
  - a formal semantics of trees of configurations produced by multi-result driving + generalization (Sec. 3);
  - a formal semantics of graphs of configurations produced by multi-result supercompilation (Sec. 4);
- a more precise representation of backward graph nodes, which result from folding (Sec. 4), and an MRSC algorithm adapted to this representation (Sec. 5).

We further propose an approach for the modular verification of semantics preservation for any supercompiler built using the proposed components. The main idea is to provide, as much as possible, general proofs of correctness for the high-level parts of the supercompiler, which do not depend on the object language. Implementers of particular supercompilers then only need to fill those parts of the correctness proof that are specific to the object language and the particular choice of supercompiler primitive operations (generalization, folding, whistle, ...). A key idea for simplifying and modularizing the overall correctness proof is the assumption that any graph created by multi-result supercompilation, if unfolded to a tree, can also be obtained by performing multi-result driving alone. This important assumption allows to decompose the correctness verification in several stages:

- verification of semantics preservation for driving + generalization in isolation. In other words, all trees obtained from driving must preserve language semantics (Sec. 3). As this step is mostly language-specific, it must be done separately for each supercompiler.

- general proof that graphs produced by multi-result supercompilation have the same meaning as the trees to which they can be unfolded (Sec. 4). This proof can be reused directly for any supercompiler based on the described algorithm.
- using some assumptions about the folding operation, general proof that any MRSC-produced graph unfolded into a tree can also be produced by driving alone (Sec. 5). This proof can also be reused for any particular supercompiler, only the assumptions about the folding operation must be verified separately in each case. The advantage is that the impact of folding is limited only to checking its specific conditions; other parts of the proof can ignore folding completely.

The whistle is used only for ensuring termination, and has no impact on semantics preservation at all.

The current implementation of the proposed approach is in Coq and we give most definitions and property statements directly in Coq[2]. Understanding the ideas behind most such fragments of Coq source should not require any deep knowledge of that language, as they use familiar constructs from functional programming languages and formal logic, only with slight variations in syntax. We also stress that the approach is not limited in any way to working only with Coq, or even to computer-assisted formal verification in general. Of course, the implementation should be directly portable to other dependently-typed languages such as Agda or Idris. Most importantly, the implementation of the main data structures and algorithms should be easy to port as well to any modern language with good support for functional programming (Haskell, ML, Scala, ...). In the latter case a realistic alternative to formal proofs is the use of the sufficient conditions on basic blocks in conjunction with a property-based testing tool like QuickCheck [2], which can still give high confidence in the correctness of the supercompiler.

2 Preliminaries

Before delving into the main components of the proposed MRSC formalization, let's quickly introduce some preliminary definitions, mostly necessary to make things work in a total dependently-typed language like Coq or Agda.

2.1 Modeling General Recursive Computations

In order to formally prove correctness results, we need first to formalize programming language semantics, in an executable form, in a total language. As any kind of interpreter (big-step, small-step, ...) for a Turing-complete language

[2] All proofs, some auxiliary definitions, and most lemmas are omitted. Interested readers can find all the gory details in the Coq sources accompanying the paper: `https://sites.google.com/site/dkrustev/Home/publications`

is a partial, potentially non-terminating function, we must select some roundabout way to represent such functions. While different partiality monads based on coinduction exist, they all have different advantages and drawbacks [1], and none of them is practical for all possible occasions. So we stick to a very basic representation: monotonic functions of type **nat** → **option** A, where we use the usual ordering for **nat** and an "information ordering" for **option** A (which is explicitly defined in Fig. 1). The **nat** argument serves as a finite limit to the amount of computation performed. If we can complete it within this limit, we return the resulting value wrapped in Some, otherwise we return None. The monotonicity condition simply states, that if we can return a value within some limit, we will always return the same value when given higher limits. Such a representation should be compatible with most kinds of existing partiality monads, which can be equipped with a *run* function of type **nat** → **option** A. We are interested in the extensional equivalence of such computations: if one of 2 computations, given some limit, can return a value, the other also has some (possibly different) limit, after which it will return the same value, and vice versa. This is captured in the definition of EvalEquiv. Note that if converting the definition of MRSC below

```
Definition FunNatOptMonotone {A} (f: nat → option A) : Prop :=
  ∀ n x, f n = Some x → ∀ m, n ≤ m → f m = Some x.
Inductive OptInfoLe {A} : option A → option A → Prop :=
  | OptInfoLeNone: ∀ x, OptInfoLe None x
  | OptInfoLeSome: ∀ x, OptInfoLe (Some x) (Some x).
Definition EvalEquiv {A} (f1 f2: nat → option A) : Prop :=
  ∀ x, (∃ n, f1 n = Some x) ↔ (∃ n, f2 n = Some x).
```

Fig. 1: Model of general recursive computations

to a Turing-complete functional language like Haskell or ML, it would probably be more practical to replace this encoding of potentially non-terminating computations with a representation of lazy values.

2.2 Bar Whistles

Following [4], we use inductive bars [3] as whistles (Fig. 2). Recall that the main job of the whistle is to ensure termination of the supercompiler. The advantage of inductive bars is that the supercompiler definition becomes structurally-recursive on the bar, making it obviously terminating. Different kinds of bars can be built in a compositional way, and we can also build an inductive bar from a decidable almost-full relation – almost-full relations being another constructive alternative to the well-quasi orders classically used in supercompilation [20]. We do not go into further detail here, because the construction of a suitable inductive bar is orthogonal to the correctness issues we study.

```
Inductive Bar {A: Type} (D: list A → Type) : list A → Type :=
  | BarNow: ∀ {h: list A} (bz: D h), Bar D h
  | BarLater: ∀ {h: list A} (bs: ∀ c, Bar D (c :: h)), Bar D h.
Record BarWhistle (A: Type) : Type := MkBarWhistle {
  dangerous: list A → Prop;
  dangerousCons: ∀ (c: A) (h: list A), dangerous h → dangerous (c :: h);
  dangerousDec: ∀ h, {dangerous h} + {¬ (dangerous h)};
  inductiveBar: Bar dangerous nil
}.
```

Fig. 2: Inductive bars for whistles

3 Driving, Trees of Configurations and Their Semantics

3.1 Trees of Configurations, Sets of Trees

Given some abstract type representing *configurations*, we can give a straightforward definition of trees of configurations:

```
Variable Cfg: Type.
CoInductive CfgTree: Type := CTNode: Cfg → FList CfgTree → CfgTree.
```

What is conspicuously missing are *contractions*. We assume – as suggested in [4] – that when present, the contraction of an edge is merged with the configuration below the edge. So we need to only deal with configurations, thus simplifying the formal definition of MRSC. As an obscure technical detail, we use an alternative definition of lists here (FList A) just to avoid some restrictions of Coq's productivity checker [15]. (Readers not particularly interested in such idiosyncrasies of Coq can safely pretend that FList is the same as **list**, drop the "fl" prefix in functions/predicates like flMap, FLExists, ..., and consider list2flist and flist2list as identity functions.)

Multi-result driving will typically produce an infinite list of infinite trees of configurations. It appears hard to explicitly enumerate this list in a total language, as it grows both in width and in height at the same time. As an alternative, we can re-use the trick of Grechanik et al [4] to make a compact representation of the whole set of trees produced by multi-result driving. The meaning of the encoding is probably easiest to grasp in terms of the process of multi-result driving itself, to which we shall come shortly.

```
CoInductive CfgTreeSet: Type :=
  | CTSBuild: Cfg → FList (FList CfgTreeSet) → CfgTreeSet.
```

The only important operation on such sets of trees is membership. Luckily it is definable as a coinductive relation. This definition is best illustrated by a picture (Fig. 3).

```
CoInductive TreeInSet: CfgTree → CfgTreeSet → Prop :=
  | TreeInBuild: ∀ c (ts: FList CfgTree) (tsss: FList (FList CfgTreeSet)),
      FLExists (fun tss ⇒ FLForall2 TreeInSet ts tss) tsss
      → TreeInSet (CTNode c ts) (CTSBuild c tsss).
```

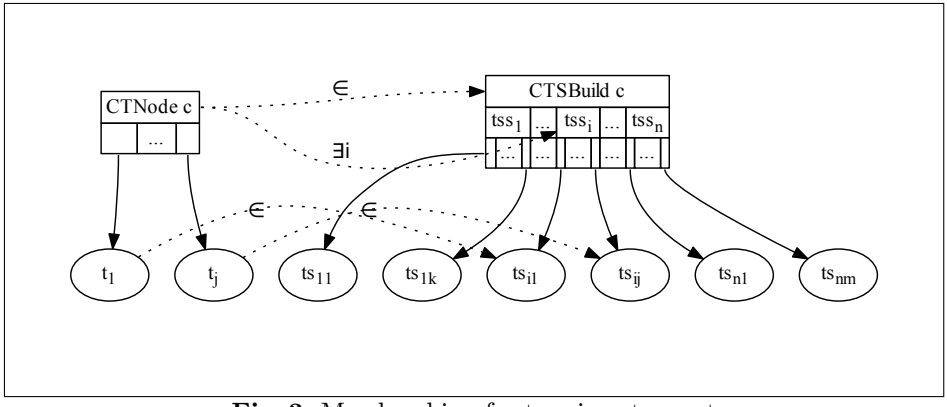

Fig. 3: Membership of a tree in a tree-set

3.2 Driving (+ Generalization)

We explicitly try to follow as closely as possible the Agda formalization of MRSC proposed by Grechanik et al [4] (modulo alpha-equivalence). We assume the same primitive – *mrscSteps* – for performing driving and generalization. Given a current configuration, it will produce a list of results, each of which is in turn a list of new configurations. Any such list of configurations is the result of either a driving step or a generalization step performed on the initial configuration. We can use this primitive to build the whole set of trees corresponding to a given initial configuration. buildCfgTrees is actually the full high-level algorithm for multi-result driving!

```
Variable mrscSteps: Cfg → list (list Cfg).
CoFixpoint buildCfgTrees (c: Cfg) : CfgTreeSet :=
  CTSBuild c (flMap (flMap buildCfgTrees)
    (list2flist (map list2flist (mrscSteps c)))).
```

3.3 Tree Semantics

We first introduce several abstract primitives, related to the semantics of the object language (Fig. 4). The most important one – *evalCfg* – represents a "reference" interpreter for (configurations of) the object language. As it is encoded using the chosen representation for general recursive functions, it must satisfy the corresponding monotonicity condition. The other 3 primitives (*evalNodeXXX*) supply the language-specific parts of the "tree interpreter". The generic algorithm of this tree interpreter – which formally defines the semantics of trees of configurations – is given in Fig. 5. It is easy to deduce the purpose of the 3 primitives from this definition itself: *evalNodeResult* computes (if possible) the final value for the current tree node, while *evalNodeInitEnv* and *evalNodeStep* serve to maintain the evaluation environment (assuming a fixed evaluation order from left to right for subtrees of the node). Note that, while *evalCfg* can be a

general recursive computation, the primitives for the tree interpreter do not have a "fuel" argument – they are expected to be total functions.

```
Variable Val: Type. Variable EvalEnv: Type.
Variable evalCfg: EvalEnv → Cfg → nat → option Val.
Hypothesis evalCfg_monotone: ∀ env c, FunNatOptMonotone (evalCfg env c).
Variable evalNodeInitEnv: EvalEnv → Cfg → EvalEnv.
Variable evalNodeStep: EvalEnv → Cfg → list (option Val) → option Val → EvalEnv.
Variable evalNodeResult: EvalEnv → Cfg → list (option Val) → option Val.
```

Fig. 4: Evaluation primitives

```
Fixpoint evalCfgTree (env: EvalEnv) (t: CfgTree) (n: nat) {struct n} : option Val :=
  match n with
  | 0 ⇒ None
  | S n ⇒ match t with
    | CTNode c ts ⇒
      let stepf (p: EvalEnv × list (option Val)) (t: CfgTree)
        : EvalEnv × list (option Val) :=
        let env := fst p in let ovs := snd p in
        let ov := evalCfgTree env t n in
        (evalNodeStep env c ovs ov, ov::ovs) in
      evalNodeResult env c
        (snd (fold_left stepf (flist2list ts) (evalNodeInitEnv env c, nil)))
    end
  end.
```

Fig. 5: Tree interpreter

The following requirement – *evalCfg_evalCfgTree_equiv* – is the cornerstone for establishing MRSC correctness: we assume that each tree produced by multi-result driving is semantically equivalent to the initial configuration. This assumption permits to easily establish another natural coherence property – that all trees resulting from multi-result driving are semantically equivalent.

```
Hypothesis evalCfg_evalCfgTree_equiv: ∀ env c t,
  TreeInSet t (buildCfgTrees c)
  → EvalEquiv (evalCfg env c) (evalCfgTree env t).
Lemma AllTreesEquiv: ∀ env c t1 t2, let ts := buildCfgTrees c in
  TreeInSet t1 ts → TreeInSet t2 ts →
  EvalEquiv (evalCfgTree env t1) (evalCfgTree env t2).
```

4 Graphs of Configurations, Graph Semantics

4.1 Graph definition

The definition of MRSC graphs of configurations (Fig. 6) is still similar to the Agda formalization of Grechanik et al. [4], the important difference being the treatment of backward nodes resulting from folding: they contain an index determining the upper node + a function that can convert the upper configuration to the lower one. Since a graph having a backward node as root has no sense (and is never created by MRSC), we capture this invariant by a dedicated data type – **TopCfgGraph**.

```
Inductive CfgGraph : Type :=
  | CGBack: nat → (Cfg → Cfg) → CfgGraph
  | CGForth: Cfg → list CfgGraph → CfgGraph.
Inductive TopCfgGraph : Type :=
  | TCGForth: Cfg → list CfgGraph → TopCfgGraph.
Definition top2graph (g: TopCfgGraph) : CfgGraph :=
  match g with
  | TCGForth c gs ⇒ CGForth c gs
  end.
```

Fig. 6: Graphs of configurations

4.2 Converting Graphs to Trees

We can define the unfolding of a graph of configurations into a tree of configurations (Fig. 7). The main work is done in a helper function graph2treeRec, which must maintain several recursion invariants. The parameter *topG* always keeps a reference to the root of the tree, and is used to give a meaning even to incorrect graphs, in which the index of a backward node is too big. In such cases we simply assume the index points to the root of the graph. *gs* contains – in reverse order – all nodes in the path to the root of the graph; it grows when passing through a forward node and shrinks back when passing through a backward node. The parameter *f* is the composition of all configuration transformations of backward nodes, through which we have already passed.

4.3 Graph Semantics

We define the semantics of graphs of configurations by defining a "graph interpreter" (Fig. 8). Again, we use a helper function evalGraphRec, whose parameters *topG*, *gs*, and *f* are used in the same way as in graph2treeRec, in essence performing graph unfolding "on the fly". For the interpretation of each node we reuse the same language-specific primitives we have used for the tree interpreter.

```
CoFixpoint graph2treeRec (topG: TopCfgGraph) (gs: list TopCfgGraph)
  (f: Cfg → Cfg) (g: CfgGraph) : CfgTree :=
  let topGraph2tree (gs: list TopCfgGraph) (f: Cfg → Cfg) (g: TopCfgGraph)
    : CfgTree :=
    match g with
    | TCGForth c gs1 ⇒
        CTNode (f c) (flMap (graph2treeRec topG (g::gs) f) (list2flist gs1))
    end in
  match g with
  | CGBack i f1 ⇒ match nthWithTail i gs with
    | Some (backG, gs1) ⇒ topGraph2tree gs1 (fun c ⇒ f (f1 c)) backG
    | None ⇒ topGraph2tree nil (fun c ⇒ f (f1 c)) topG
    end
  | CGForth c gs1 ⇒ topGraph2tree gs f (TCGForth c gs1)
  end.
Definition graph2tree (g: TopCfgGraph) : CfgTree :=
  graph2treeRec g nil (fun c ⇒ c) (top2graph g).
```

Fig. 7: Unfolding a graph into a tree

```
Fixpoint evalGraphRec (topG: TopCfgGraph) (gs: list TopCfgGraph)
  (f: Cfg → Cfg) (env: EvalEnv) (g: CfgGraph) (n: nat) {struct n} : option Val :=
  match n with
  | 0 ⇒ None
  | S n ⇒
    let evalTopGraph (gs: list TopCfgGraph) (f: Cfg → Cfg) (g: TopCfgGraph)
          : option Val :=
      match g with
      | TCGForth c subGs ⇒
        let stepf (p: EvalEnv × list (option Val)) (g1: CfgGraph)
          : EvalEnv × list (option Val) :=
          let env := fst p in let ovs := snd p in
          let ov := evalGraphRec topG (g::gs) f env g1 n in
          (evalNodeStep env (f c) ovs ov, ov::ovs) in
        evalNodeResult env (f c)
          (snd (fold_left stepf subGs (evalNodeInitEnv env (f c), nil)))
      end in
    match g with
    | CGBack i f1 ⇒
      let g_gs := match nthWithTail i gs with
                    Some p ⇒ p | None ⇒ (topG, nil) end in
      evalTopGraph (snd g_gs) (fun c ⇒ f (f1 c)) (fst g_gs)
    | CGForth c subGs ⇒ evalTopGraph gs f (TCGForth c subGs)
    end
  end.
Definition evalGraph (env: EvalEnv) (g: TopCfgGraph) (n: nat) : option Val :=
  evalGraphRec g nil (fun c ⇒ c) env (top2graph g) n.
```

Fig. 8: Graph interpreter

4.4 Graph semantics correctness

It is not hard to spot that the definition of evalGraphRec has many similarities to the definitions of evalCfgTree and graph2treeRec. Actually evalGraphRec can be seen as a manually fused composition of the other 2 functions. This fact is formally verified by the following theorem, which is one of the 2 key intermediate results used in establishing MRSC correctness:

```
Theorem evalGraph_evalCfgTree: ∀ env g,
  EvalEquiv (evalGraph env g) (evalCfgTree env (graph2tree g)).
```

5 Multi-result Supercompilation

5.1 Definition

We need 2 more primitives (besides *mrscSteps*) in order to define a generic multi-result supercompiler. As we have already explained, we assume a whistle in the form of an inductive bar (Fig. 9). The signature of the folding primitive – *tryFold* – is determined by our decision how to encode backward nodes: if folding is possible, *tryFold* must return:

- a valid index into the list of previous configurations (*tryFold_length*);
- a configuration-transforming function, which will turn the old configuration (higher in the tree) into the current one (*tryFold_funRelatesCfgs*).

The requirement that no folding should be possible with empty history – *tryFold_nil_None* – is quite natural. The last requirement about *tryFold* – *tryFold_funCommutes* – deserves more attention. It states that any configuration transformation, returned by folding, commutes (in a way precisely defined in Fig. 9) with *mrscSteps*. The reason for this assumption is that it permits us to prove that unfolding a mrsc-produced graph will always result into a tree that can also be obtained through driving alone. The latter property is key for enabling modular verification of the different supercompilers produced by the proposed approach. This requirement is further discussed in Sec. 6.

Apart from folding, the rest of the mrsc definition is very similar to the one proposed by Grechanik et al. [4]. The main algorithm is encoded by the recursive function mrscHelper. The top-level function mrsc may seem complicated at first, but it is only because it uses some Coq-specific idioms to convert the final list of results from type **CfgGraph** to type **TopCfgGraph**. If we ignored this conversion, the definition would become:

```
mrsc (c: Cfg) : list CfgGraph := mrscHelper (inductiveBar whistle) c.
```

5.2 Containment of Graphs in Driving Tree Sets

As already hinted, the following containment result is the second key theorem necessary for ensuring mrsc correctness:

```
Theorem graph2tree_mrsc_In_buildCfgTrees: ∀ c g,
  In g (mrsc c) → TreeInSet (graph2tree g) (buildCfgTrees c).
```

It opens the way to replace establishing the semantic correctness of graphs with verifying only the semantic correctness of trees.

```
Definition CommutesWithSteps (f: Cfg → Cfg) :=
  ∀ c, mrscSteps (f c) = map (map f) (mrscSteps c).
Variable tryFold: list Cfg → Cfg → option (nat × (Cfg → Cfg)).
Hypothesis tryFold_nil_None: ∀ c, tryFold nil c = None.
Hypothesis tryFold_length: ∀ h c i f, tryFold h c = Some (i, f) → i < length h.
Hypothesis tryFold_funRelatesCfgs: ∀ h c i f c1 h1,
  tryFold h c = Some (i, f) → nthWithTail i h = Some (c1, h1) → c = f c1.
Hypothesis tryFold_funCommutes: ∀ h c i f,
  tryFold h c = Some (i, f) → CommutesWithSteps f.
Variable whistle: BarWhistle Cfg.
```

Fig. 9: Remaining MRSC primitives

```
Fixpoint mrscHelper (h: list Cfg)
  (b: Bar (dangerous whistle) h) (c: Cfg) {struct b} : list CfgGraph :=
  match tryFold h c with
  | Some (n, f) ⇒ CGBack n f :: nil
  | None ⇒
    match dangerousDec whistle h with
    | left _ ⇒ nil
    | right Hdang ⇒
      match b in (Bar _ h) return (¬ dangerous whistle h → list CfgGraph)
      with
      | BarNow h' bz ⇒ fun (Hdang: ¬ dangerous whistle h') ⇒
          match Hdang bz with end
      | BarLater h' bs ⇒ fun (_: ¬ dangerous whistle h') ⇒
          map (CGForth c) (flat_map (fun css : list Cfg ⇒
              listsProd (map (mrscHelper (bs c)) css)) (mrscSteps c))
      end Hdang
    end
  end.
Definition mrsc (c: Cfg) : list TopCfgGraph :=
  let gs := mrscHelper (inductiveBar whistle) c in
  mapWithInPrf gs
    (fun g Hin ⇒ match g return _ = g → TopCfgGraph with
      | CGBack n f ⇒ fun Heq ⇒
        let Hin' := eq_rect g (fun g ⇒ In g gs) Hin _ Heq in
        match mrscHelper_nil_notBack _ _ _ Hin' with end
      | CGForth c gs ⇒ fun _ ⇒ TCGForth c gs
      end eq_refl).
```

Fig. 10: Big-step multi-result supercompilation

5.3 MRSC Correctness

The next theorem is the main result in this article. Its proof directly relies on the intermediate theorems evalGraph_evalCfgTree and graph2tree_mrsc_In_buildCfgTrees, and also on the key assumption *evalCfg_evalCfgTree_equiv*. The last assumption is the main task left to the user of the approach to verify individually in each particular case.

```
Theorem mrsc_correct: ∀ env g c, In g (mrsc c) →
  EvalEquiv (evalGraph env g) (evalCfg env c).
```

6 Current Status and Future Work

The main disadvantage of the current approach is that it has not been field-tested yet on specific supercompilers. We have actually started to build a simple supercompiler for SLL, a basic first-order functional language often used (under different names, and with small variations) in many works on supercompilation [9,17]. Although the verification of this supercompiler – using the proposed approach – is not complete, it has already pointed to some improvements. One of these improvements is already present – evaluation functions take an environment as input (and some return a modified environment). Environment-based evaluation (of trees/graphs of configurations) is useful in typical supercompilers for at least two reasons:

- Contractions in the tree/graph often take the form of patterns, which bind variables inside the corresponding subtree. A successful pattern match will supply values for these bound variables. Environment-based interpreters are a well-known and well-working approach to keep track of such new bindings during the evaluation of subexpressions.
- Generalization is often represented in the tree/graph with let-expressions (or something working in a similar way), whose evaluation by the tree interpreter also involves passing new bindings for the evaluation of sub-trees. Moreover, it is difficult to pass such bindings using a substitution operation, as we bind the let-bound variable not to a configuration, but to a computation, which may yield the value of the corresponding subtree. Environment-based evaluation appears easier to use in this case.

The introduction of evaluation environments as an abstract data type has made impossible to provide general proofs of monotonicity for the tree and for the graph interpreter. Such monotonicity properties can still be very useful, for example when the user proves equivalence between the tree interpreter and the reference interpreter of configurations. We plan to try to recover these general proofs in the future, by postulating some user-defined ordering of environments, and using it to formulate monotonicity requirements for the language-specific building blocks of the tree interpreter.

Another limitation made apparent by the SLL supercompiler involves the precise definition of configuration equality, which is used in several places (in the definition of membership inside our representation of tree sets; in the required properties of the folding primitive, etc.). Currently we use strict syntactic equality, but this may prove too restrictive in many practical cases. For example, if the configuration can bind variables, alpha-equivalence may be a much more useful notion of equality. We plan to fix this deficiency by introducing an abstract configuration equality relation, and basing other definitions on this user-supplied relation instead of the built-in syntactic equality.

We use a general assumption, that if we unfold a graph resulting from supercompilation into a tree, this tree must be among the trees generated by multi-result driving alone. The commutativity condition on configurations produced by folding is imposed exactly in order to make a general proof of this assumption. Both the general assumption and the folding requirement appear to be satisfied in typical cases (such as renaming-based folding, or folding only identical configurations). There are cases, however, where both the assumption and the condition do not hold. Consider folding based on substitution instead of renaming (take some unspecified functional language)[3]. In this case we can have a path fragment in the tree:

$$\ldots f(x) \longrightarrow \ldots \longrightarrow f(5)$$

where folding is possible as there is a substitution converting $f(x)$ into $f(5)$. Assume further that f is defined by pattern matching on its argument. In this case the tree unfolded from the graph cannot be produced by driving alone, because:

- driving $f(5)$ will proceed with a deterministic reduction, giving rise to a single subtree;
- the graph at node $f(x)$ will have 2 subgraphs corresponding to the 2 clauses in the definition of f (assuming Peano representation of natural numbers). Making a copy of this graph will give 2 subtrees at node $f(5)$ as well.

Note, however, that we can achieve similar effect with a suitable combination of generalization and renaming-based folding. The relevant path fragment in this case will be:

$$\ldots f(x) \longrightarrow \ldots \longrightarrow f(5) \longrightarrow \texttt{let}\ y = 5\ \texttt{in}\ f(y) \longrightarrow f(y)$$

Here folding by renaming from $f(x)$ to $f(y)$ is possible. So, ruling out folding by substitution does not lead to an important loss of generality. It remains to be seen if there are other useful kinds of folding ruled out by our assumptions, and, independently, if we can relax the properties required of the folding primitive, while keeping the overall separation of concerns achieved by the current approach.

Another possibility for future improvements concerns the requirement of semantics preservation for the tree interpreter (*evalCfg_evalCfgTree_equiv*). Recall, that this assumption must be verified separately for each supercompiler. This verification step will likely be the most complicated one for specific implementations based on the current approach. So it is interesting to try to find simpler sufficient conditions for the tree interpreter, which can replace this requirement.

Finally, note that we completely omit any formalization of residualization (converting the graph of configurations back into a program in the object language). To complete the proof of correctness of a specific supercompiler, the user must provide a separate proof for the correctness of residualization. Still,

[3] Example suggested by Ilya Klyuchnikov

the proposed approach may offer some help: as we have established the equivalence in the general case between the tree and the graph interpreter (for trees produced by unfolding a graph), it suffices for the user to make a specific proof of equivalence between the residualized program and the tree interpreter – which can be slightly simpler that equivalence w.r.t. the graph interpreter, as folding has no impact on the tree interpreter.

An interesting long-term goal would be to try to apply a similar approach for modular verification to other generalizations of classical supercompilation, such as distillation [5].

7 Related Work

Multi-result supercompilation was introduced by Klyuchnikov et al. [10] and more formally described in later work by the same authors [11], as a generalization of classical [18, 19] and non-deterministic supercompilation. Already in the second work on MRSC, there is a clear separation between the high-level method of multi-result supercompilation, which can be described in a completely language-neutral way, and the set of language-specific basic operations needed to obtain a complete working supercompiler. The recent Agda formalization of "big-step" MRSC [4] is based on an even simpler set of basic operations encapsulating the language-specific parts of the supercompiler. Our work is directly based on this Agda formalization, with some changes in the treatment of folding necessitated by our different goals. We do not use a compact representation for the set of graphs produced by MRSC, but reuse the same idea to represent the set of trees obtained by multi-result driving. It should be easy to merge the 2 formalizations for use cases that may need both an efficient way to represent and manipulate the result set of MRSC and a setting for verifying the correctness of these results.

A similar generic framework for implementing and verifying classical-style supercompilers has been proposed by the author [13]. The current work can be seen as extending that framework to cover the case of multi-result supercompilation. The current formalization is actually simpler, despite the fact that it covers a more general method. This is partly due to the inherent simplicity of MRSC itself, and also a result of incremental improvements based on experience with the previous framework. In particular, we hope the current approach will provide better treatment for generalization. Our stronger assumption of tree-interpreter semantics preservation permits us to have an unified general proof of both soundness and completeness of MRSC, while [13] deals only with soundness.

Taking a wider perspective – about supercompilation and related techniques in general – there are numerous works describing specific supercompilers, including correctness proofs; too many to attempt to enumerate them here. There are also some more general approaches about establishing supercompiler correctness, which are not tied to specific implementations. Sand's improvement theorem [16], for example, gives a general technique for proving semantics preservation of dif-

ferent program transformations, but only for the case of a functional language used as input.

There exist also a few (mostly) language-neutral descriptions of classical supercompilation as a general technique. Jones presents an abstract formulation of driving [6], with only a small number of assumptions about the object language. Still, some of these assumptions seem geared towards simple imperative or first-order tail-recursive functional languages. Also, termination and generalization are not treated there. Klimov [7] covers the complete supercompilation process, and proves a number of interesting high-level properties. To achieve these results, Klimov assumes a specific object language (first-order functional) and data domain. It seems feasible, however, to generalize this approach by abstracting from the details of the object language.

8 Conclusions

We have described the current state of a general approach for modular verification of arbitrary multi-result supercompilers. The main correctness property we are after is preservation of object language semantics by the supercompiler. One key observation that enables our modular approach is that MRSC can be defined in terms of a reusable top-level algorithm + a set of independent building blocks, which must be implemented anew for each specific supercompiler. We propose to apply a similar kind of modularity for the verification of correctness: general reusable theorems concerning the top-level algorithm, which rely on a set of smaller independent properties concerning the building blocks (driving, folding, ...). Only the latter set of properties must be verified from scratch in each case, the general theorems can be reused. Another, more specific idea concerning verification modularization is to consider the unfolding of MRSC-produced graphs of configurations back into trees of configurations. If we can show (as we do, under certain assumptions), that the unfolded graph will always belong to the set of trees produced by driving (+ generalization) alone, we can then ignore completely graphs and folding during the verification process. What is needed in this case is to only verify semantics preservation for the set of trees produced by multi-result driving.

We stress again, that although the current implementation of the approach is inside a proof assistant (Coq), and we speak of formal verification, the approach can be equally useful for verification by testing. Most modern languages already feature property-based testing tools mostly inspired by the Haskell QuickCheck library [2]. The set of correctness assumptions we have identified is perfectly suitable as a starting point of such property-based testing. So, even without doing formal computer-checked proofs, implementers of multi-result supercompilers can use the proposed approach to gain confidence in the correctness of their software.

We must still warn that we are reporting the current state of a work in progress. As we test the approach on specific supercompilers, we shall likely find

further opportunities for improving the framework and making it more easily applicable.

Acknowledgments The author would like to thank Sergei Romanenko and Ilya Klyuchnikov for the insightful discussions related to the topic of this article.

References

1. Chlipala, A.: Certified Programming with Dependent Types. MIT Press (2013), http://adam.chlipala.net/cpdt/
2. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Odersky, M., Wadler, P. (eds.) ICFP. pp. 268–279. ACM (2000)
3. Coquand, T.: About Brouwer's fan theorem. Revue Internationale de Philosophie 230, 483–489 (2004)
4. Grechanik, S.A., Klyuchnikov, I.G., Romanenko, S.A.: Staged multi-result supercompilation: filtering before producing. preprint 70, Keldysh Institute (2013)
5. Hamilton, G.W.: Distillation: extracting the essence of programs. In: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 61–70. ACM (2007)
6. Jones, N.D.: The essence of program transformation by partial evaluation and driving. In: Bjoner, D., Broy, M., Zamulin, A. (eds.) Perspectives of System Informatics'99. Lecture Notes in Computer Science, vol. 1755, pp. 62–79. Springer Berlin Heidelberg (2000)
7. Klimov, A.V.: A program specialization relation based on supercompilation and its properties. In: Turchin, V. (ed.) International Workshop on Metacomputation in Russia, META 2008 (2008)
8. Klimov, A.V., Klyuchnikov, I.G., Romanenko, S.A.: Automatic verification of counter systems via domain-specific multi-result supercompilation. In: Klimov, A.V., Romanenko, S.A. (eds.) Proceedings of the Third International Workshop on Metacomputation (META 2012). pp. 112–141 (2012)
9. Klyuchnikov, I.: The ideas and methods of supercompilation. Practice of Functional Programming (7) (2011), in Russian
10. Klyuchnikov, I., Romanenko, S.A.: Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In: Clarke, E.M., Virbitskaite, I., Voronkov, A. (eds.) Ershov Memorial Conference. Lecture Notes in Computer Science, vol. 7162, pp. 210–226. Springer (2011)
11. Klyuchnikov, I.G., Romanenko, S.A.: Formalizing and implementing multi-result supercompilation. In: Klimov, A.V., Romanenko, S.A. (eds.) Proceedings of the Third International Workshop on Metacomputation (META 2012). pp. 142–164 (2012)
12. Krustev, D.: A simple supercompiler formally verified in Coq. In: Nemytykh, A.P. (ed.) Proceedings of the Second International Workshop on Metacomputation in Russia (META 2010). pp. 102–127 (2010)
13. Krustev, D.: Towards a framework for building formally verified supercompilers in Coq. In: Loidl, H.W., Peña, R. (eds.) Trends in Functional Programming, Lecture Notes in Computer Science, vol. 7829, pp. 133–148. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-40447-4_9

14. Mendel-Gleason, G.: Types and verification for infinite state systems. PhD thesis, Dublin City University, Dublin, Ireland (2011)
15. Picard, C.: Représentation coinductive des graphes. These, Université Paul Sabatier - Toulouse III (Jun 2012), http://tel.archives-ouvertes.fr/tel-00862507
16. Sands, D.: Proving the correctness of recursion-based automatic program transformations. Theor. Comput. Sci. 167(1&2), 193–233 (1996)
17. Sørensen, M.H.: Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation. Master's thesis, Københavns Universitet, Datalogisk Institut (1994)
18. Sørensen, M.H., Glück, R.: Introduction to supercompilation. In: Hatcliff, J., Mogensen, T., Thiemann, P. (eds.) Partial Evaluation: Practice and Theory. Lecture Notes in Computer Science, vol. 1706, pp. 246–270. Springer-Verlag (1999)
19. Turchin, V.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (July 1986)
20. Vytiniotis, D., Coquand, T., Wahlstedt, D.: Stop when you are almost-full: Adventures in constructive termination. In: Beringer, L., Felty, A. (eds.) Interactive Theorem Proving. Lecture Notes in Computer Science, vol. 7406, pp. 250–265. Springer Berlin Heidelberg (2012)