Certifying supercompilation for Martin-Löf's type theory

Ilya Klyuchnikov and Sergei Romanenko

Keldysh Institute of Applied Mathematics

Abstract. The paper describes the design and implementation of a *certifying* supercompiler TT Lite SC, which takes an input program and produces a residual program and a proof of the fact that the residual program is equivalent to the input one. As far as we can judge from the literature, this is the first implementation of a certifying supercompiler for a non-trivial higher-order functional language. The proofs generated by TT Lite SC can be verified by a type checker which is independent from TT Lite SC and is not based on supercompilation. This is essential in cases where the reliability of results obtained by supercompilation is of fundamental importance. Currently, the proofs can be either verified by the type-checker built into TT Lite, or converted into Agda programs and checked by the Agda system. The main technical contribution is a simple but intricate interplay of supercompilation and type theory.

1 Introduction

Supercompilation [1,2] is a program manipulation technique that was originally introduced by V. Turchin in terms of the programming language Refal (a first-order applicative functional language) [3], for which reason the first supercompilers were designed and developed for the language Refal [4].

Roughly speaking, the existing supercompilers can be divided into two large groups: "optimizing" supercompilers that try to make programs more efficient, and "analyzing" supercompilers that are meant for revealing and proving some hidden properties of programs, in order to make programs more suitable for subsequent analysis and/or verification.

The main idea behind the program analysis by supercompilation is that supercompilation "normalizes" and "trivializes" the structure of programs by removing modularity and levels of abstraction (carefully elaborated by the programmer). Thus, although the transformed program becomes less human-friendly, it may be more convenient for *automatic* analysis.

Examples of using supercompilation for the purposes of analysis and verification are: verification of protocols [5,6], proving the equivalence of programs [7], contract checking (*e.g.* the verification of monadic laws) [8], problem solving in Prolog style by inverse computation [9], proving the correctness of optimizations (verifying improvement lemmas) [10], proving the productivity of corecursive functions [11]. It should be noted that the use of supercompilation for analysis and verification is based on the assumption:

The supercompiler we use preserves the semantics of programs.

In the following we will silently assume that this requirement is satisfied[1].

At this point we are faced with the problem of correctness of supercompilation itself, which has a number of aspects. A non-trivial supercompiler is a sophisticated construction, whose proof of correctness is bound to be messy and cumbersome, involving as it does several areas of computer science. (For example, the proof of correctness of the supercompiler HOSC takes more than 30 pages [12].) Such a proof may contain some bugs and overlooks. Even if the proof is perfect, the implementation of the supercompiler may be buggy. The correctness of the implementation can be verified by means of formal methods. However, even the verification of a "toy" supercompiler is technically involved [13].

As we have seen, ensuring the correctness of a supercompiler is a difficult task. But, what we are *really* interested in is the correctness of the *results* of supercompilation. Thus we suggest the following solution.

Let the supercompiler produce a pair: a residual program, and a proof of the fact that this residual program is equivalent to the original program. The essential point is that the proof must be verifiable with a proof checker that is not based on supercompilation and is (very!) much simpler than the supercompiler.

The advantages of such *certifying supercompilation* are the following.

- The supercompiler can be written in a feature-rich programming language (comfortable for the programmer), even if programs in this language are not amenable to formal verification.
- The implementation of the supercompiler can be buggy, and yet its results can be verified and relied upon.
- The supercompiler can be allowed to apply incorrect techniques, or, more exactly, some techniques that are only correct under certain conditions that the supercompiler is unable to check. In this case, some results of supercompilation may be incorrect, but it is possible to filter them out later.

A certifying supercompiler, in general, has to deal with two languages: the programs transformed by the supercompiler are written in the *subject* language, while the *proof* language is used for formulating the proofs generated by the supercompiler. The problem is that the proof language and the subject language must be consistent with each other in some subtle respects. For example, the functions in the subject language may be partial (as in Haskell), but total in the proof language (as in Coq [14] or Agda [15]). And semantic differences of that kind may cause a lot of trouble.

The above problem can be circumvented if the subject language of the supercompiler is also used as its proof language! Needless to say, in this case the subject language must have sufficient expressive power[2].

[1] Note that some supercompilers are not semantics-preserving, changing as they do termination properties and/or error handling behavior of programs.

[2] Note, however, that the implementation language of the supercompiler does not need to coincide with either the subject language or the proof language.

The purpose of the present work is to show the feasibility and usefulness of certifying supercompilation. To this end, we have developed and implemented TT Lite [16], a proof-of-concept supercompiler for Martin-Löf's type theory (TT for short) [17]. The choice of TT as the subject+proof language was motivated as follows.

- The language of type theory is sufficiently feature-rich and interesting. (It provides inductive data types, higher-order functions and dependent types.)
- The type theory is easy to extend and can be implemented in a simple, modular way.
- Programs and proofs can be written in the same language.
- The typability of programs is decidable, and type checking can be easily implemented.

To our knowledge, the supercompiler described in the present work is the first one capable of producing residual programs together with proofs of their correctness. It is essential that these proofs can be verified by a type checker that is not based on supercompilation and is independent from the supercompiler.

The general idea that a certifying program transformation system can use Martin-Löf's type theory both for representing programs and for representing proofs of correctness was put forward by Albert Pardo and Sylvia da Rosa [18]. We have shown that this idea can be implemented and does work in the case of program transformations performed by supercompilation.

The TT Lite project[3] comprises 2 parts: TT Lite Core, which is a minimalistic implementation of the language of type theory (a type-checker, an interpreter and REPL), and TT Lite SC, which is a supercompiler. The results produced by TT Lite SC are verified by the type checker implemented in TT Lite Core. TT Lite Core does not depend on TT Lite SC and is not based on supercompilation[4].

TT Lite Core implements the collection of constructs and data types that can be usually found in textbooks on type theory: dependent functions, pairs, sums, products, natural numbers, lists, propositional equality, the empty (bottom) type and the unit (top) type. Also the site of the project contains a tutorial on programming in the TT Lite language with (a lot of) examples taken from [19,20].

While [16] contains full technical information about TT Lite in detail, this paper describes a small subset of TT Lite and can be regarded as a gentle, step-by-step introduction to [16]. In this paper we limit ourselves to the language which only contains dependent functions, natural numbers and identity (we use the abbreviation $\Pi\mathcal{N}\mathcal{I}$ for this subset of TT Lite). This allows us to present and explain the fundamental principles of our certifying supercompiler without going into too much technical detail.

[3] `https://github.com/ilya-klyuchnikov/ttlite`

[4] This design is similar to that of Coq [14]. The numerous and sophisticated Coq "tactics" generate proofs written in Coq's Core language, which are then verified by a relatively small type checker. Thus, occasional errors in the implementation of tactics do not undermine the reliability of proofs produced by tactics.

```
1  plus : forall (x y : Nat). Nat;
2  plus = \(x y : Nat). elim Nat (\(n : Nat). Nat) y (\(n r : Nat). Succ r) x;
3  $x : Nat; $y : Nat; $z : Nat;
4  in1 = plus $x (plus $y $z);
5  in2 = plus (plus $x $y) $z;
6  (out1, pr1) = sc in1;
7  (out2, pr2) = sc in2;
8  id_in1_out1 : Id Nat in1 out1;
9  id_in1_out1 = pr1;
10 id_in2_out2 : Id Nat in2 out2;
11 id_in2_out2 = pr2;
12 id_out1_out2 : Id Nat out1 out2;
13 id_out1_out2 = Refl Nat out1;
14 id_in1_in2 : Id Nat in1 in2;
15 id_in1_in2 = proof_by_trans Nat in1 in2 out1 pr1 pr2;
```

Fig. 1. Proving the associativity of addition via normalization by supercompilation.

2 TT Lite SC in action

TT Lite SC implements a supercompiler which can be called by programs written in the TT Lite input language by means of a built-in construct `sc`. (This supercompiler, as well as TT Lite Core, however, is implemented in Scala, rather than in the TT Lite language.) The supercompiler takes as input an expression (with free variables) in the TT Lite language and returns a pair: an output expression and a proof that the output expression is equivalent to the input one. The proof is also written in the TT Lite language and certifies that two expressions are *extensionally* equivalent, which means that, if we assign some values to the free variables appearing in the expressions, the evaluation of the expressions will produce the same result.

Both the output expression and the proof produced by the supercompiler are first-class values and can be further manipulated by the program that has called the supercompiler. Technically, the input expression is converted (reflected) to an AST, which is then processed by the supercompiler written in Scala. The result of supercompilation is then reified into values of the TT Lite language.

Let us consider the example in Figure 1 illustrating the use of TT Lite SC for proving the equivalence of two expressions [7].

As in Haskell and Agda [15], the types of defined expressions do not have to be specified explicitly. However, type declarations make programs more understandable and easier to debug.

Lines 1–2 define the function of addition for natural numbers. Line 3 declares (assumes) 3 free variables `$x`, `$y` and `$z` whose type is `Nat`. By convention, the names of free variables start with `$`. Lines 4–5 define two expressions whose equivalence is to be proved.

Now we come to the most interesting point: line 6 calls the built-in function `sc`, which takes as input the expression `in1` and returns its supercompiled version `out1` along with the proof `pr1` for the fact that `in1` and `out1` are extensionally equivalent (i.e., given `$x`, `$y` and `$z`, `in1` and `out1` return the same value). Line 7 does the same for `in2`, `out2` and `pr2`.

$p ::= (def \mid dec)*$	program
$def ::= id = e; \mid id : e;\ id = e;$	optionally typed definition
$dec ::= \$id :\ e;$	declaration (assumption)
$e ::= x$	variable
$\mid \mathbf{c}$	built-in constant
$\mid \mathbf{b}(x :\ e).\ e(x)$	built-in binder
$\mid e_1\ e_2$	application
$\mid \mathbf{elim}\ e_t\ e_m\ \overline{e_i}\ e_d$	elimination
$\mid (e)$	parenthesized expression

Fig. 2. TT Lite: syntax

Lines 8–9 formally state that `pr1` is *indeed* a proof of the equivalence of `in1` and `out1`, having as it does the appropriate type, and this fact is verified by the type checker built into TT Lite Core. Lines 10–11 do the same for `in2`, `out2` and `pr2`.

And now, the final stroke! Lines 12–13 verify that `out1` and `out2` are "propositionally equivalent" or, in simpler words, they are just textually the same. Hence, by transitivity (lines 14–15), `in1` is extensionally equivalent to `in2`. And this proof has been *automatically* found by supercompilation and verified by type checking [7]. The function `proof_by_trans` is coded in the TT Lite language in the file `examples/id.tt`.

3 TT Lite: syntax and semantics

In the following, the reader is assumed to be familiar with the basics of programming in Martin-Löf's type theory [19,20].

TT Lite Core provides a modular and extensible implementation of type theory. Technically speaking, it deals with a monomorphic version of type theory with intensional equality and universes.

TT Lite SC is based on TT Lite Core and makes heavy use of the expression evaluator (normalizer) and type checker provided by TT Lite Core. Hence, before looking into the internals of the supercompiler, we have to consider the *details* of how normalization and type checking are implemented in TT Lite Core.

The Syntax of the TT Lite language is shown in Figure 2. A program is a list of declarations and definitions. A definition (as in Haskell) can be of two kinds: with or without an explicit type declaration. There is also a possibility to declare the type of an identifier without defining its value (quite similar to module parameters in Agda), in which case the identifier must start with `$`.

A TT Lite expression is either a variable, a built-in constant, a binder[5], an application, an application of an eliminator [22] or an expression enclosed in parentheses. This syntax should be familiar to functional programmers: variables and applications have usual meaning, binders are a generalization of λ-abstractions, eliminators are a "cross-breed" of `case` and `fold`.

[5] See [21, Section 1.2] describing *abstract binding trees*.

In general, an eliminator in the TT Lite language has the form $\texttt{elim}\ e_t\ e_m\ \overline{e_i}\ e_d$ where e_t is the type of the values that are to be eliminated, e_m is a "motive" [22], e_i correspond to the cases that can be encountered when eliminating a value, and e_d is an expression that produces values to be eliminated[6].

The typing and normalization rules implemented in the $\Pi\mathcal{NI}$ subset of TT Lite can be found in Figure 3. Essentially, they correspond to the rules described in [19,20], but have been refactored, in order to be closer to their actual implementation in TT Lite.

The typing and normalization rules are formulated with respect to a context Γ, where Γ is a list of pairs of two kinds: $x := e$ binds a variable to an expression defining its value, while $x : T$ binds a variable to a type. By tradition, we divide the rules into 3 categories: *formation* (F), *introduction* (I) and *elimination* (E) rules. A rule of the form $\Gamma \vdash e : T$ means that e has the type T in the context Γ, while $[\![e]\!]_\Gamma = e'$ means that e' is the result of normalizing e in the context Γ.

Our rules mainly differ from the corresponding ones in [19,20] in that subexpressions are explicitly normalized in the process of type checking. It should be also noted that these expressions, in general, may contain free variables. If a TT Lite expression is well-typed, the normalization of this expression is guaranteed to terminate. So, any function definable in the TT Lite language is total by construction. Figure 4 gives a definition of the neutral variable [22] of an expression. Essentially, a neutral variable is the one that prevents an elimination step from being performed[7].

4 TT Lite SC: supercompilation

The implementation of TT Lite SC is based on the MRSC Toolkit [23], which builds graphs of configurations [3] by repeatedly applying a number of graph rewrite rules. The nodes of a partially constructed graph are classified as either complete or incomplete. The supercompiler selects an incomplete node, declares it to be the *current* one, and turns it into a complete node by applying to it the rules specified by the programmer. The process starts with a graph containing a single (initial) configuration and stops when all nodes become complete[8].

Figure 5 schematically depicts the graph building operations that can be performed by the MRSC Toolkit. (Incomplete nodes are shown as dashed circles, the current node is inside a rounded box.) These operations are applied to the current node (which, by definition, is incomplete). The operation *unfold* adds child nodes to the current node. This node becomes complete, while the new nodes are declared to be incomplete. The operation *fold* adds a "folding" edge from the current node to one of its parents, and the node becomes complete. The operation *stop* just declares the current node to be complete, and does nothing else.

[6] By the way, application is essentially an eliminator for functional values.

[7] Recall that application is also a special case of eliminator

[8] Or the graph is declared by the whistle to be "dangerous" (in this case the supercompiler just discards the graph), but this feature is not used by TT Lite SC.

$$(v)\ \ \Gamma,\ x : T \vdash x : [\![T]\!]_\Gamma \qquad ([\![v]\!])\ \ [\![x]\!]_{\Gamma,\ x:=t} = t \qquad (\mathcal{U})\ \ \mathcal{U}_n : \mathcal{U}_{n+1}$$

$$(\Pi F)\ \frac{\Gamma \vdash A : \mathcal{U}_m \quad \Gamma, x : [\![A]\!]_\Gamma \vdash B(x) : \mathcal{U}_n}{\Gamma \vdash \Pi(x : A).\, B(x) : \mathcal{U}_{max(m,n)}}$$

$$(\Pi I)\ \frac{\Gamma \vdash A : \mathcal{U}_m \quad \Gamma, x : [\![A]\!]_\Gamma \vdash t(x) : B(x)}{\Gamma \vdash \lambda(x : A).\, t(x) : [\![\Pi(x : A).\, B(x)]\!]_\Gamma}$$

$$(\Pi E)\ \frac{\Gamma \vdash f : \Pi(x : A).\, B(x) \quad \Gamma \vdash t : A}{\Gamma \vdash f\ t : [\![B(x)]\!]_{\Gamma, x:=[\![t]\!]_\Gamma}}$$

$$([\![\Pi F]\!])\ [\![\Pi(x : A).\, B(x)]\!]_\Gamma = \Pi(x : [\![A]\!]_\Gamma).\, [\![B(x)]\!]_\Gamma$$

$$([\![\Pi I]\!])\ [\![\lambda(x : A).\, t(x)]\!]_\Gamma = \lambda(x : [\![A]\!]_\Gamma).\, [\![t(x)]\!]_\Gamma$$

$$([\![\Pi E]\!])\ [\![f\ t]\!]_\Gamma = \lceil [\![f]\!]_\Gamma [\![t]\!]_\Gamma \rceil_\Gamma \qquad (\lceil \Pi E \rceil)\ \lceil (\lambda(x : A).t(x))u \rceil_\Gamma = [\![t(x)]\!]_{\Gamma, x:=[\![u]\!]_\Gamma}$$

$$(\mathbb{N}F)\ \frac{}{\Gamma \vdash \mathbb{N} : \mathcal{U}_0} \qquad (\mathbb{N}I_1)\ \frac{}{\Gamma \vdash 0 : \mathbb{N}} \qquad (\mathbb{N}I_2)\ \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash Succ\ n : \mathbb{N}}$$

$$(\mathbb{N}E)\ \frac{\begin{array}{c}\Gamma \vdash m : [\![\Pi(x : \mathbb{N}).\, \mathcal{U}_k]\!]_\Gamma \quad \Gamma \vdash f_0 : [\![m\ 0]\!]_\Gamma \\ \Gamma \vdash f_s : [\![\Pi(x : \mathbb{N})\ (y : m\ x).\, m\ (Succ\ x)]\!]_\Gamma \quad \Gamma \vdash n : \mathbb{N}\end{array}}{\Gamma \vdash elim\ \mathbb{N}\ m\ f_0\ f_s\ n : [\![m\ n]\!]_\Gamma}$$

$$([\![\mathbb{N}I_2]\!])\ [\![Succ\ n]\!]_\Gamma = Succ\ [\![n]\!]_\Gamma$$

$$([\![\mathbb{N}E]\!])\ [\![elim\ \mathbb{N}\ m\ f_0\ f_s\ n]\!]_\Gamma = \lceil elim\ \mathbb{N}\ [\![m]\!]_\Gamma\ [\![f_0]\!]_\Gamma\ [\![f_s]\!]_\Gamma\ [\![n]\!]_\Gamma \rceil_\Gamma$$

$$(\lceil \mathbb{N}E_1 \rceil)\ \lceil elim\ \mathbb{N}\ m\ f_0\ f_s\ 0 \rceil_\Gamma = [\![f_0]\!]_\Gamma$$

$$(\lceil \mathbb{N}E_2 \rceil)\ \lceil elim\ \mathbb{N}\ m\ f_0\ f_s\ (Succ\ n) \rceil_\Gamma = [\![f_s\ n\ (elim\ \mathbb{N}\ m\ f_0\ f_s\ n)]\!]_\Gamma$$

$$(\mathcal{I}F)\ \frac{\Gamma \vdash A : \mathcal{U}_m \quad \Gamma \vdash t_1 : [\![A]\!]_\Gamma \quad \Gamma \vdash t_2 : [\![A]\!]_\Gamma}{\Gamma \vdash \mathcal{I}\ A\ t_1\ t_2 : \mathcal{U}_m}$$

$$(\mathcal{I}I)\ \frac{\Gamma \vdash A : \mathcal{U}_m \quad \Gamma \vdash t : [\![A]\!]_\Gamma}{\Gamma \vdash Refl\ A\ t : [\![\mathcal{I}\ A\ t\ t]\!]_\Gamma}$$

$$(\mathcal{I}E)\ \frac{\begin{array}{c}\Gamma \vdash \mathcal{I}\ A\ t_1\ t_2 : \mathcal{U}_k \quad \Gamma \vdash m : [\![\Pi(x : A)\,(y : A)\,(z : \mathcal{I}\ A\ x\ y).\, \mathcal{U}_k]\!]_\Gamma \\ \Gamma \vdash f : [\![\Pi(x : A).\, m\ x\ x\ (Refl\ A\ x)]\!]_\Gamma \quad \Gamma \vdash eq : [\![\mathcal{I}\ A\ t_1\ t_2]\!]_\Gamma\end{array}}{elim\,(\mathcal{I}\ A\ t_1\ t_2)\ m\ f\ eq : [\![m\ t_1\ t_2\ eq]\!]_\Gamma}$$

$$([\![\mathcal{I}F]\!])\ [\![\mathcal{I}\ A\ t_1\ t_2]\!]_\Gamma = \mathcal{I}\ [\![A]\!]_\Gamma\ [\![t_1]\!]_\Gamma\ [\![t_2]\!]_\Gamma$$

$$([\![\mathcal{I}I]\!])\ [\![Refl\ A\ t]\!]_\Gamma = Refl\ [\![A]\!]_\Gamma\ [\![t]\!]_\Gamma$$

$$([\![\mathcal{I}E]\!])\ [\![elim\,(\mathcal{I}\ A\ t_1\ t_2)\ m\ f\ eq]\!]_\Gamma = \lceil elim\ [\![\mathcal{I}\ A\ t_1\ t_1]\!]_\Gamma\ [\![m]\!]_\Gamma\ [\![p]\!]_\Gamma\ [\![eq]\!]_\Gamma \rceil_\Gamma$$

$$(\lceil \mathcal{I}E \rceil)\ \lceil elim\,(\mathcal{I}\ A\ t_1\ t_2)\ m\ f\ (Refl\ A\ t_3) \rceil_\Gamma = [\![f\ t_3]\!]_\Gamma$$

Fig. 3. TT Lite: rules

$$\begin{array}{ll} nv(x\ e) = x & nv(elim\ \mathbb{N}\ m\ f_0\ f_s\ e) = nv(e) \\ nv(e_1\ e_2) = nv(e_1) & nv(elim\,(\mathcal{I}\ A\ t_1\ t_2)\ m\ f\ x) = x \\ nv(elim\ \mathbb{N}\ m\ f_0\ f_s\ x) = x & nv(elim\,(\mathcal{I}\ A\ t_1\ t_2)\ m\ f\ e) = nv(e) \end{array}$$

Fig. 4. Finding the neutral variable of a term

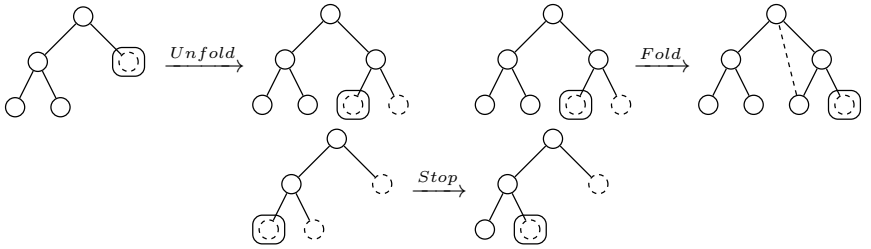

Fig. 5. Basic operations of MRSC

The MRSC toolkit allows the nodes and edges of a graph to hold arbitrary information. Information in a node is called a configuration. In the case of TT Lite SC, a configuration is a pair consisting of a term (expression) and a context. Schematically, a graph node will be depicted as follows: $\boxed{t \mid \Gamma}$. We use two kind of edge labels (I and E) in TT Lite SC. The first kind corresponds to the decomposition of a constructor, while the second kind corresponds to case analysis and (in general case) primitive recursion performed by an eliminator. In the case of recursive eliminators (such as $\mathbb{N}$, $List$) the label also holds information to be used for finding possible foldings.

We use the following notation for depicting nodes and transitions between nodes:

(a) $\boxed{t_0 \mid \Gamma_0} \xrightarrow{I(Succ)} \boxed{t_1 \mid \Gamma_1}$ (c) $\boxed{t_0 \mid \Gamma_0} \longleftarrow \boxed{t_1 \mid \Gamma_1}$

(b) $\boxed{t_0 \mid \Gamma_0} \xrightarrow{E(y \to c_1, r)} \boxed{t_1 \mid \Gamma_1}$ $\xrightarrow{E(y \to c_1, \bullet)} \boxed{t_2 \mid \Gamma_2}$ (d) $\boxed{t_0 \mid \Gamma_0} \longrightarrow \bullet$

An *unfolding edge* is schematically represented by a right arrow, and a *folding edge* by a left arrow. (a) represents a decomposition. (b) corresponds to case analysis performed by an eliminator. If the eliminator is a recursive one, the edge label contains a recursive term r, otherwise this position is occupied by the dummy placeholder $\bullet$. (c) represents a folding edge. (d) represents a complete node without child nodes. Sometimes, nodes will be denoted by greek letters. For example, a folding edge from β to α will be depicted as $a \leftarrow \beta$.

The rules used by TT Lite SC for building graphs of configurations are presented in Figure 6(a). In simple cases, the left part of a rule is a pattern that specifies the structure of the nodes the rule is applicable to. But, sometimes a rule has the form of an inference rule with a number of premises ("guarded pattern matching" in programmer's terms). The rules are ordered.

Let us consider rules of various kinds in more details.

There are two kinds of rules for building graphs of configurations: type-specific and general ones. Type-specific rules determine how *driving* [24] is performed for constructions introduced by a specific type. General rules do not

(a) Construction of a graph of configurations

$$\text{(Fold)}\ \frac{\exists \alpha \in anc(\beta) : \alpha \xrightarrow{E(y \mapsto t_1, c)} \alpha_1}{\alpha \leftarrow \beta} \qquad \text{(Stop)}\ \frac{depth(\beta) > n}{\boxed{c \mid \Gamma} \longrightarrow \bullet}$$

$$(\mathbb{N}I_2')\ \frac{}{\boxed{Succ\ t \mid \Gamma} \xrightarrow{Succ} \boxed{t \mid \Gamma}} \qquad (\mathbb{N}E')\ \frac{nv(c) = y \quad y : \mathbb{N}}{\boxed{c \mid \Gamma} \begin{array}{l} \xrightarrow{y \mapsto 0, \bullet} \boxed{[\![c]\!]_{\Gamma, y:=0} \mid \Gamma} \\ \xrightarrow{y \mapsto Succ\ y, c} \boxed{[\![c]\!]_{\Gamma, y:=Succ\ y} \mid \Gamma} \end{array}}$$

$$\text{(Default)}\ \frac{}{\boxed{c \mid \Gamma} \longrightarrow \bullet}$$

(b) Code generation from a graph of configurations

$$\mathcal{C}\left[\beta \leftarrow\right]_\rho = \rho(\beta) \qquad \mathcal{C}\left[\rightarrow \bullet\right]_\rho = \alpha.e \qquad \mathcal{C}\left[\xrightarrow{I(Succ)} \alpha_1\right]_\rho = Succ\ \mathcal{C}[\alpha_1]_\rho$$

$$\mathcal{C}\left[\begin{array}{l} \xrightarrow{E(y \mapsto 0, \bullet)} \alpha_1 \\ \xrightarrow{E(y \mapsto Succ\ y, r)} \alpha_2 \end{array}\right]_\rho = \begin{array}{l} elim\ \mathbb{N}\ (\lambda(y : \mathbb{N}).\ tp(\alpha)) \\ \qquad \mathcal{C}[\alpha_1]_\rho\ (\lambda(y : \mathbb{N})(v : tp(\alpha)).\ \mathcal{C}[\alpha_2]_{\rho + (\alpha \to v)})\ y \end{array}$$

(c) Proof generation from a graph of configurations

$$\mathcal{P}\left[\beta \leftarrow\right]_{\rho,\phi} = \phi(\beta) \qquad \mathcal{P}\left[\rightarrow \bullet\right]_{\rho,\phi} = Refl\ tp(\alpha)\ \alpha.e$$

$$\mathcal{P}\left[\xrightarrow{I(Succ)} \alpha_1\right]_{\rho,\phi} = cong\ \mathbb{N}\ \mathbb{N}\ Succ\ \alpha_1.e\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{P}[\alpha_1]_{\rho,\phi}$$

$$\mathcal{P}\left[\begin{array}{l} \xrightarrow{y \mapsto 0, \bullet} \alpha_1 \\ \xrightarrow{y \mapsto Succ\ y, r} \alpha_2 \end{array}\right]_{\rho,\phi} = \begin{array}{l} elim\ \mathbb{N}\ (\lambda(y : \mathbb{N}).\ \mathcal{I}\ tp(\alpha)\ \alpha.e\ \mathcal{C}[\alpha]_\rho)\mathcal{P}[\alpha_1]_{\rho,\phi} \\ (\lambda(y : \mathbb{N})(v : \mathcal{I}\ tp(\alpha)\ \alpha.e\ \mathcal{C}[\alpha]_\rho).\ \mathcal{P}[\alpha_2]_{\rho + (\alpha \to \mathcal{C}[\![\alpha]\!]_\rho), \phi + (\alpha \to v)})\ y \end{array}$$

(d) Utility function

$$\begin{array}{l} cong\ :\ \Pi(A : \mathcal{U}_i)(B : \mathcal{U}_j)(f : \Pi(\_ : A).B)(x : A)(y : A)(\_ : \mathcal{I}\ A\ x\ y).\ \mathcal{I}\ B\ (f\ x)\ (f\ y); \\ cong = \lambda(A : \mathcal{U}_i)(B : \mathcal{U}_j)(f : \Pi(\_ : A).B)(x : A)(y : A)(i : \mathcal{I}\ A\ x\ y). \\ \quad elim\ (\mathcal{I}\ A\ x\ y)\ (\lambda(x\ y : A)(\_ : \mathcal{I}\ A\ x\ y).\mathcal{I}\ B\ (f\ x)\ (f\ y))\ (\lambda(x : A).Refl\ B\ (f\ x))\ i; \end{array}$$

Fig. 6. Supercompilation rules

correspond to a specific type and ensure the finiteness of graphs of configurations. In this paper we present all general rules of TT Lite SC and the driving rules for the $\Pi\mathcal{NI}$ subset[9]. Rules *Fold*, *Stop* and *Default* are general ones. Rules $(\mathbb{N}I_2')$ and $(\mathbb{N}E')$ are driving rules for $\mathbb{N}$. All other driving rules can be found in [16].

Rules $(\mathbb{N}I_2')$ and $(\mathbb{N}E')$ add new nodes to the graph by applying the MRSC operation Unfold. Any driving rule in TT Lite SC can be classified as either a decomposition or a case analysis by means of an eliminator.

From the perspective of the type theory, decomposition corresponds to formation and introduction rules. The essence of decomposition is simple: we take a construct to pieces (which become the new nodes) and label the edges with some information (about the construct that has been decomposed). In the case of the $\Pi\mathcal{NI}$ subset there is only one decomposition rule: $(\mathbb{N}I_2')$.

Note that, in general, for each formation and introduction rule there exists a corresponding decomposition rule, provided that the corresponding value has

[9] In this paper the identity type is only used for constructing proofs of correctness. Thus, for brevity, we do not discuss here driving rules for identity.

some internal structure. However, in the case of the type $\mathbb{N}$, the constructs $\mathbb{N}$ and 0 have no internal structure.

Decomposition is not performed for binders Π and λ. The reason is that in this case we could be unable to generate a proof of correctness of decomposition, because the core type theory does not provide means for dealing with extensional equality.

When the supercompiler encounters an expression with a neutral variable, it considers all instantiations of this variable that are allowed by its type. Then, for each possible instantiation, the supercompiler adds a child node and labels the corresponding edge with some information about this instantiation.

When dealing with eliminators for recursive types (such as $\mathbb{N}$ and $List$), we record the expression corresponding to the "previous step of elimination"[10] in the edge label. For example, for the expression $elim\ \mathbb{N}\ m\ f_0\ f_s\ (Succ\ y)$, the expression corresponding to the previous step of elimination is $elim\ \mathbb{N}\ m\ f_0\ f_s\ y$. In the case of the subset $\Pi\mathcal{NI}$, the only rule for case analysis is $(\mathbb{N}E')$, which considers two possible instantiations of the neutral variable y of the type $\mathbb{N}$: 0 and $Succ\ y$. Note that by putting an instantiation in the context and then normalizing the expression, we perform *positive information propagation*: the key step of supercompilation. Storing "the previous step of eliminator" is crucial for folding and code generation, since the only way to introduce recursive functions in TT Lite is via eliminators.

A technical note should be done here: in TT Lite SC we generate new variables for instantiations and put them into context. So, in TT Lite SC, given a neutral variable y of type $\mathbb{N}$, the child configuration corresponding to the case $y = Succ\ y_1$ is $\boxed{[\![c]\!]_{\Gamma, y:=Succ\ y_1} \mid \Gamma, y_1 : \mathbb{N}}$, where y_1 is a fresh variable of the type $\mathbb{N}$. However, in this specific case we can avoid the generation of a new variable by reusing the variable y and not extending the context (since y is already in the context). This small trick allows us to make presentation of code generation and proof generation rules (given in the next sections) shorter and less cumbersome.

Note that the information stored on the edge label is enough for generating both the residual program and the proof of correctness (in a straightforward way).

Another special case is the application of a neutral variable. Since a neutral variable is bound to have a functional type, we cannot enumerate all its possible instantiations. In such situation, most supercompilers (*e.g.* HOSC [25]) perform a decomposition of the application, but, to keep the supercompiler simple, we prefer not to decompose such applications.

General rules ($Fold$, $Stop$, $Default$) are the core of TT Lite SC. In short, general rules ensure the finiteness of graphs of configurations. This is achieved either by folding the current expression to a previously encountered one or by stopping the development of the current branch of the graph.

In the rule $Fold$, $anc(\beta)$ is a set of ancestor nodes of the current node β and c is an expression in the node β. The rule itself is very simple. Suppose that

[10] = "recursive call" of the same eliminator

the current node has an ancestor node whose "previous step of elimination" in the outgoing edge is (literally) the same as the current term. Then the rule *Fold* is applicable, and the current configuration can be folded to the parent one. In the residual program this folding will give rise to a function defined by primitive recursion.

Folding in TT Lite SC differs from that in traditional supercompilers. Namely, most supercompilers perform folding when the current expression is a renaming of some expression in the history. However, since TT Lite SC has to encode recursion by means of eliminators, the mixture of folding and renaming would create some technical problems. So, we prefer to separate them.

If no folding/driving rule is applicable, the rule *Default* is applied. (This rule is the last and has the lowest priority.) In this case, the current node becomes complete and the building of the current branch of the graph is stopped.

In general, the process of repeatedly applying driving rules, together with the rules *Fold* and *Default*, may never terminate. Thus, in order to ensure termination, we use the rule *Stop*, whose priority is higher than that of the unfolding rules and the rule *Default*. In this supercompiler we use a very simple termination criterion: the building of the current branch stops if its depth exceeds some threshold n. Note that, in the case of TT Lite SC, the expressions appearing in the nodes of the graph are self-contained, so that they can be just output into the residual program.

Since the graph of configurations is finitely branching, and all branches have finite depth, the graph of configurations cannot be infinite. Therefore, the process of graph building eventually terminates.

The generation of the residual program corresponding to a completed graph of configurations is performed by recursive descent. The function that implements the residualization algorithm is defined in Figure 6 (b). A call to this function has the form $\mathcal{C}[\alpha]_\rho$, where α is the current node, and ρ is an environment (mapping of nodes to variables) to "tie the knot" on "folding" edges. The initial call to the function $\mathcal{C}$ has the form $\mathcal{C}[root]_{\{\}}$, where $root$ is the root node of the graph of configurations.

The function $\mathcal{C}$ performs pattern matching against the edges going out of the current node. (In the rules, the patterns are enclosed into square brackets.) We use the following conventions: the current node is α, $\alpha.e$ is an expression in the node α, $tp(\alpha)$ is the type of the expression appearing in the node α. If $e \mid \Gamma$ is the configuration in the node α, and $\Gamma \vdash e : T$, then $tp(\alpha) = T$. In the last rule (corresponding to a case analysis of a neutral var of type $\mathbb{N}$) v is a fresh variable.

Note that rules for construction of a graph of configurations take into account that residualization facilities of $\mathcal{C}$ are limited (by eliminators) and produce a graph that *can* be residualized by $\mathcal{C}$.

5 Proof generation

The function that implements the proof generator is defined in Figure 6 (c). A call to this function has the form $\mathcal{P}[\alpha]_{\rho,\phi}$, where α is the current node, while ρ

and ϕ are two environments. ρ is used for folding in residual programs (when encoding recursion), while ϕ is used for folding in proofs (when encoding proofs by induction). Technically, ϕ binds some nodes to corresponding inductive hypotheses. The initial call to the function $\mathcal{P}$ has the form $\mathcal{P}[root]_{\{\},\{\}}$, where *root* is the root node of the graph of configurations.

Generated proofs are based on the use of propositional *equality* (*i.e.* syntactic identity of normalized expressions), functional *composition* and *induction*.

- The residual expression corresponding to a childless node is the same as the one appearing in this node. Hence, the proof amounts to the use of reflexivity of equality (*i.e.* is a call to *Refl*).
- The proofs corresponding to decompositions of configurations exploit the congruence of equality: the whole proof is constructed by combining subproofs (demonstrating that the arguments of constructors are equal) with the aid of the combinator *cong* defined in Figure 6 (d) ([16] uses more congruence combinators).
- The proofs corresponding to eliminators are by structural induction. The motive of a new eliminator is now a proof. When specifying a motive of eliminator, $\mathcal{C}[\alpha]_\rho$ is used in the same way as during code generation (using the same environment ρ). But, when generating subproofs for recursive eliminators, $\mathcal{C}[\alpha]_\rho$ is used to extend the environment ρ. Also ϕ is extended, to bind the current node to a subproof (inductive hypothesis).

Note that the same graph of configurations is used both for generating the residual program and for generating the proof. If TT Lite SC would have been implemented in "direct" style (without explicit graphs of configurations) like in [26], such reuse would be problematic, which would produce a negative effect on the modularity of our design.

Despite the fact that the rules for $\mathcal{P}$ are compact, they are technically involved since there is an intricate use of two "folding contexts" ρ and ϕ. Another technically interesting point is that code generation function $\mathcal{C}$ is used as a "type inferencer" when constructing a motive for proof term encoded via eliminators.

6 Example

Let us consider supercompilation of the expression `in2` from the Figure 1 (which is `plus (plus $x $y) $z`). The graph of configurations for this expression is shown in Figure 7. The generated code is shown in Figure 8. The proof that `in2` and `out2` are equivalent is not shown here because of the lack of space (but can be found at the project website). The project site contains more examples of certifying supercompilation. In particular, TT Lite SC is capable of proving most equivalences of expressions that have been presented in the paper [7] (and proved by HOSC). The difference from HOSC, however, is that TT Lite SC generates explicit proofs, which can be verified by type checking.

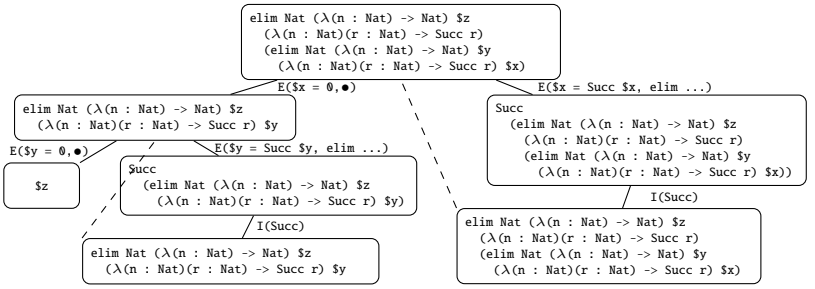

Fig. 7. Example: graph of configurations and "graph of residualization"

```
1  elim Nat (\($x : Nat) -> Nat)
2    (elim Nat (\ ($y : Nat) -> Nat) $z (\ ($y : Nat)(r2 : Nat) -> Succ r2) $y)
3    (\ ($x : Nat) (r1 : Nat) -> Succ r1) $x
```

Fig. 8. Example: residual expression (`out2`)

7 Related work

One of the ideas exploited by TT Lite SC is that of *supercompilation by evaluation*. As was shown in [26], supercompilation can be based on an evaluator which tries to reduce an open term to head normal form. Unfortunately, if the subject language is not a total one (as in the case of Haskell), the evaluation may not terminate. For this reason, Bolingbroke and Peyton Jones had to equip their evaluator with a termination check ("local whistle"), which may interfere with global termination check in subtle ways. Hence, the evaluator had to be fused, to some extent, with other parts of their supercompiler.

In the case of TT Lite SC, the totality of the subject language has allowed us to greatly *simplify* and *modularize* the structure and implementation of our supercompiler. In particular, since the evaluation of all expressions, including expressions with free variables, terminates, the evaluator is trivialized into a standard normalizer, so that TT Lite SC just reuses the interpreter provided by TT Lite Core. Thus, the supercompiler knows nothing about the internals of the evaluator, using it as a "black box".

Another point, where we exploit the totality of the subject language, is case analysis. Since a function returns a result for *any* input, all possible instantiations of a neutral variable can be found by just examining the type of this variable. Note that supercompilers dealing with partial functions usually find instantiations of a variable by taking into account how this variable is actually used in the program. Technically, this means that driving has to be implemented as a combination of case analysis and variable instantiation. TT Lite SC, however, completely decouples case analysis from variable instantiation.

Since our goal was to investigate whether certifying supercompilation is possible *in principle*, we tried to keep our proof-of-concept certifying supercompiler

as simple and comprehensible as possible. And, indeed, the ingredients of our supercompiler for the $\Pi\mathcal{NI}$ subset fit on half a page! And yet, this is a supercompiler for a non-trivial language (of Martin-Löf's type theory), which is powerful enough in order to be able to prove most term equivalences from [7].

In some application areas, however, TT Lite SC is inferior even to "naïve" partial evaluators. In particular, it fails to pass the classical KMP test [24]. The reason is that, in residual programs produced by TT Lite SC, all loops and recursive functions has to be encoded in terms of eliminators. Thus, TT Lite SC can only fold a configuration to "the previous step of an eliminator". This is a restricted form of folding, which is not sufficient in the case of the KMP test. We could generalize our graph building rules to allow TT Lite SC to use more complicated forms of folding. However, in this case, residual programs would be difficult to express in terms of eliminators and, as a consequence, the generation of correctness proofs, corresponding to residual programs, would become more technically involved. This problem needs to be further investigated.

8 Conclusions

We have developed and implemented a *certifying* supercompiler TT Lite SC, which takes an input program and produces a residual program paired with a proof of the fact that the residual program is equivalent to the input one. As far as we can judge from the literature, this is the first implementation of a certifying supercompiler for a non-trivial higher-order functional language.

A proof generated by TT Lite SC can be verified by a type checker of TT Lite Core which is independent from TT Lite SC and is not based on supercompilation. This is essential in cases where the reliability of results obtained by supercompilation is of fundamental importance. For example, when supercompilation is used for purposes of program analysis and verification. Some "technical" details in the design of TT Lite SC are also of interest.

- The subject language of the supercompiler is a total, statically typed, higher-order functional language. Namely, this is the language of Martin-Löf's type theory (in its monomorphic version).
- The proof language is the same as the subject language of the supercompiler.
- Recursive functions in the subject language are written in a well-structured way, by means of "eliminators". An eliminator for an inductively defined data type performs both the case analysis and recursive calls.
- Driving is type-directed.
- There is an intricate interplay of supercompilation, type theory and limitations imposed by usage of eliminators.
- TT Lite programs (including proofs produced by supercompilation) can be exported into Agda and checked by Agda system.

References

1. Sørensen, M.H., Glück, R., Jones, N.D.: A positive supercompiler. Journal of Functional Programming **6**(6) (1996) 811–838

2. Sørensen, M.H., Glück, R.: Introduction to supercompilation. In: Partial Evaluation. Practice and Theory. Volume 1706 of LNCS. (1998) 246–270
3. Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems (TOPLAS) **8**(3) (1986) 292–325
4. Turchin, V.F.: The language Refal: The theory of compilation and metasystem analysis. Technical Report 20, Courant Institute of Mathematical Sciences (1980)
5. Lisitsa, A., Nemytykh, A.: Verification as a parameterized testing. Programming and Computer Software **33**(1) (2007) 14–23
6. Klimov, A., Klyuchnikov, I., Romanenko, S.: Automatic verification of counter systems via domain-specific multi-result supercompilation. [27]
7. Klyuchnikov, I., Romanenko, S.: Proving the equivalence of higher-order terms by means of supercompilation. In: PSI 2009. Volume 5947 of LNCS., Springer (2010)
8. Klyuchnikov, I.: Inferring and proving properties of functional programs by means of supercompilation. PhD thesis, Keldysh Institute of Applied Mathematics (2010)
9. Turchin, V.F.: Supercompilation: Techniques and results. In: Perspectives of System Informatics. Volume 1181 of LNCS., Springer (1996)
10. Klyuchnikov, I., Romanenko, S.: Towards higher-level supercompilation. [28]
11. Mendel-Gleason, G., Hamilton, G.: Development of the productive forces. [27]
12. Klyuchnikov, I.: Supercompiler HOSC: proof of correctness. Preprint 31, KIAM, Moscow (2010) URL: `http://library.keldysh.ru/preprint.asp?id=2010-31`.
13. Krustev, D.: A simple supercompiler formally verified in Coq. [28] 102–127
14. Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions. Springer (2010)
15. : The agda wiki (2013) URL: `http://wiki.portal.chalmers.se/agda/`.
16. Klyuchnikov, I., Romanenko, S.: TT Lite: a supercompiler for Martin-Löf's type theory. Preprint, KIAM, Moscow (2013) URL: `http://library.keldysh.ru//preprint.asp?lg=e&id=2013-73`.
17. Martin-Lof, P.: Intuitionistic type theory. Bibliopolis, Naples (1984)
18. Pardo, A., da Rosa, S.: Program transformation in Martin-Löf's type theory. In: CADE-12, Workshop on Proof-search in type-theoretic languages. (1994)
19. Nordström, B., Petersson, K., Smith, J.M.: Programming in Martin-Löf's type theory. Oxford University Press (1990)
20. Thompson, S.: Type theory and functional programming. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1991)
21. Harper, R.: Practical foundations for programming languages. Cambridge University Press (2012)
22. Löh, A., McBride, C., Swierstra, W.: A tutorial implementation of a dependently typed lambda calculus. Fundamenta Informaticae **21** (2010) 1001–1032
23. Klyuchnikov, I.G., Romanenko, S.A.: MRSC: a toolkit for building multi-result supercompilers. Preprint 77, KIAM (2011) URL: `http://library.keldysh.ru/preprint.asp?lg=e&id=2011-77`.
24. Jones, N.D.: The essence of program transformation by partial evaluation and driving. In: PSI 99. Volume 1755 of LNCS., Springer-Verlag (2000)
25. Klyuchnikov, I.: Supercompiler HOSC 1.0: under the hood. Preprint 63, KIAM, Moscow (2009) URL: `http://library.keldysh.ru/preprint.asp?id=2009-63`.
26. Bolingbroke, M., Peyton Jones, S.: Supercompilation by evaluation. In: Proceedings of the third ACM Haskell symposium on Haskell, ACM (2010) 135–146
27. Klimov, A., Romanenko, S., eds.: Third International Valentin Turchin Workshop on Metacomputation in Russia, Publishing House "University of Pereslavl" (2012)
28. Nemytykh, A., ed.: Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia, Publishing House "University of Pereslavl" (2010)