Keldysh Institute • Publication search

Keldysh Institute preprints • Preprint No. 73, 2013

Klyuchnikov I.G., Romanenko S.A.

TT Lite: a supercompiler for Martin-Löf's type theory

Recommended form of bibliographic references: Klyuchnikov I.G., Romanenko S.A. TT Lite: a supercompiler for Martin-Löf's type theory. Keldysh Institute preprints, 2013, No. 73, 28 p. URL: http://library.keldysh.ru/preprint.asp?id=2013-73&lg=e

KELDYSH INSTITUTE OF APPLIED MATHEMATICS
Russian Academy of Sciences

Ilya G. Klyuchnikov, Sergei A. Romanenko

TT Lite: a supercompiler for Martin-Löf's type theory

Moscow
2013

Ilya G. Klyuchnikov, Sergei A. Romanenko. TT Lite: a supercompiler for Martin-Löf's type-theory

The paper describes the design and implementation of a *certifying* supercompiler TT Lite, 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. The proof generated by TT Lite can be verified by a type checker which is independent from the supercompiler and is not based on supercompilation. This is essential in cases where the reliability of results obtained by supercompilation is of fundamental importance.

Supported by Russian Foundation for Basic Research grant No. 12-01-00972-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

Илья Ключников, Сергей Романенко. TT Lite: суперкомпилятор для теории типов Мартина-Лёфа

Описывается структура и реализация *сертифицирующего* суперкомпилятора TT Lite, преобразующего исходную программу в пару, содержащую остаточную программу и доказательство того, что остаточная программа эквивалентна исходной. Насколько можно судить по существующим публикациям, сертифицирующая суперкомпиляция для нетривиального функционального языка высшего порядка реализована впервые. Доказательства, порождаемые суперкомпилятором TT Lite могут быть верифицированы проверяльщиком типов, который независим от суперкомпилятора и не основан на суперкомпиляции. Это существенно в ситуациях, когда решающее значение имеет надежность результатов, полученных с помощью суперкомпиляции.

Работа выполнена при поддержке гранта РФФИ № 12-01-00972-а и гранта Президента РФ для ведущих научных школ № НШ-4307.2012.9.

1 Introduction

Supercompilation 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) [22], for which reason the first supercompilers were designed and developed for the language Refal [21].

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.

1.1 Analysis by transformation and the problem of correctness

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.

Following are a few examples of using supercompilation for the purposes of analysis and verification.

- Verification of protocols [13, 5].
- Proving the equivalence of programs [9].
- Contract checking (*e.g.* the verification of monadic laws) [7].
- Problem solving in Prolog style by inverse computation [17, 23].
- Proving the correctness of optimizations (verifying improvement lemmas) [10].

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.

And 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.

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

- 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 [8].) 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 [12].

1.2 Producing the result of supercompilation together with a proof of its correctness

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.

1.3 Supercompilation for Martin-Löf's type theory

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 or Agda). And semantic differences of that kind may give rise to a lot of annoying problems.

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].

The purpose of the present work was to show the feasibility and usefulness of certifying supercompilation. To this end, we have developed and implemented TT Lite, a proof-of-concept supercompiler for Martin-Löf's type theory (TT for short). 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 [19]. We have shown that this idea can be implemented and does work in the case of program transformation performed by supercompilation.

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

1.4 Outline of the preprint

Section 2 outlines the general structure of TT Lite (our supercompiler for Martin-Löf's type theory) and gives an example of its use for theorem proving. Section 3 describes the subject+proof language of TT Lite, and the rules of typing and normalization. Section 4 specifies the rules used by TT Lite for constructing graphs of configurations. Section 5 explains how the proof that the residual program is equivalent to the input one is extracted by TT Lite from the graph of configurations. Section 6 concludes the preprint.

2 TT Lite in action

The TT Lite project comprises 2 parts:

- TT Lite Core, which is a minimalistic implementation of the language of type theory (a type-checker, an interpreter and REPL).
- 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[3].

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.

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, however, is implemented in Scala, rather than in the TT Lite language.) The supercompiler takes as an 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 the 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

[3]This design is similar to that of Coq. 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  import "examples/id.tt";
2
3  plus : forall (x : Nat)(y : Nat). Nat;
4  plus = \(x : Nat)(y : Nat).
5         elim Nat (\(n : Nat). Nat) y (\(x1 : Nat). Succ) x;
6
7  $x : Nat; $y : Nat; $z : Nat;
8
9  e1 = plus $x (plus $y $z);
10 e2 = plus (plus $x $y) $z;
11 (res1, proof1) = sc e1;
12 (res2, proof2) = sc e2;
13
14 id_e1_res1 : Id Nat e1 res1;
15 id_e1_res1 = proof1;
16
17 id_e2_res2 : Id Nat e2 res2;
18 id_e2_res2 = proof2;
19
20 id_res1_res2 : Id Nat res1 res2;
21 id_res1_res2 = Refl Nat res1;
22
23 id_e1_e2 : Id Nat e1 e2;
24 id_e1_e2 = proof_by_sc Nat e1 e2 res1 proof1 proof2;
```

Figure 1: Proving the associativity of addition by means of normalization by supercompilation.

(reflected) to an AST, which then processed by the supercompiler written in Scala. The result of supercompilation is then reified into values of the TT Lite language[4].

Let us consider the example in Fig. 1 illustrating the use of TT Lite SC for proving the equivalence of two expressions [9].

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

Lines 3–5 define the function of addition for natural numbers. The interesting part of this definition is the use of an *eliminator* for both case analysis and the implementation of recursion. The first argument of an eliminator specifies the type of the value to be "eliminated". (In this case the type is `Nat`.) The

[4]Thus the proof by supercompilation can be regarded as a special case of *proof by reflection* [24].

second argument is a *motive* [15], the last argument is an expression whose value is to be "eliminated" and the other arguments correspond to various cases that can be encountered in the process of elimination. (Since a natural number is either a zero or a successor of another natural number, here we have two arguments.)

Line 7 declares (assumes) 3 free variables `$x`, `$y` and `$z` whose type is `Nat`. By convention, the names of free variables start with `$`.

Lines 9–10 define two expressions whose equivalence is to be proved.

Now we come to the most interesting point: line 11 calls the built-in function `sc`, which takes as input the expression `e1` and returns its supercompiled version `res1` along with the proof `proof1` or the fact that `e1` and `res1` are extensionally equivalent (i.e., given `$x`, `$y` and `$z`, `e1` and `res1` return the same value).

Line 12 does the same for `e2`, `res2` and `proof2`.

Lines 14–15 formally state that `proof1` is *indeed* a proof of the equivalence of `e1` and `res1`, having as it does the appropriate type, and this fact is verified by the type checker built into TT Lite Core.

Lines 17–18 do the same for `e2`, `res2` and `proof2`.

And now, the final stroke! Lines 20–21 verify that `res1` and `res2` are "propositionally equivalent" or, in simpler words, they are just textually the same. Hence, by transitivity, `e1` is extensionally equivalent to `e2`. And the proof has been found (by supercompilation) and verified (by type checking) automatically [9]. The function `proof_by_sc` is coded in 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 [18, 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.

3.1 Syntax

The Syntax of the TT Lite language is shown in Fig. 2. A program is a list of declarations and definitions. A definition (as in Haskell) can be of two kinds:

$p ::= (def \mid dec)*$	program
$def ::= id : e;\ id = e;$	definition with explicit typing
$\mid id = e;$	definition
$dec ::= \$id :\ e;$	declaration (assumption)
$e ::= v$	variable
$\mid \mathbf{c}$	built-in constant
$\mid \mathbf{b}(v :\ e).\ e$	built-in binder
$\mid e_1\ e_2$	application
$\mid \mathbf{elim}\ e_t\ e_m\ \overline{e_i}\ e_d$	elimination (special form of an application)
$\mid (e)$	parenthesized expression

Figure 2: TT Lite: syntax

Constant	Constant	Constant	Constant	Binder
$\mathcal{U}_k$	Inr	$\mathbb{N}$	Nil	$\Pi(x : e).e(x)$
σ	$\bot$	0	$Cons$	$\lambda(x : e).e(x)$
$+$	$\top$	$Succ$	$\mathcal{I}$	$\Sigma(x : e).e(x)$
Inl	$\star$	$List$	$Refl$	

Figure 3: TT Lite: built-in constants

with or without an explicit type declaration. There is also a possibility to declare the type of an identifier without defining its value (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 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` [3].

In general, an eliminator in the TT Lite language has the form

$$\mathtt{elim}\ e_t\ e_m\ \overline{e_i}\ e_d$$

where e_t is the type of the values that are to be eliminated, e_m is a "motive" [15], 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].

[5] See [2, Section 1.2] describing *abstract binding trees*.

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

The site of the project `https://github.com/ilya-klyuchnikov/ttlite` contains a tutorial on programming in the TT Lite language with examples taken from [18, 20].

Fig. 3 shows built-in constants and binders of the TT Lite language.

3.2 Semantics

The typing and normalization rules implemented in TT Lite can be found in Appendix A. Essentially, they correspond to the rules described in [18, 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 kind: $x := e$ 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*, *introduction* and *elimination* 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 [18, 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, its normalization is guaranteed to terminate. So, any function definable in the TT Lite language is total by construction.

Fig. 15 gives a definition of the neutral variable [14] of an expression. Essentially, a neutral variable is the one that prevents an elimination step from being performed[7].

4 TT Lite: supercompilation

The implementation of TT Lite SC is based on the MRSC Toolkit [11], which builds graphs of configurations [22] 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].

Fig. 4 schematically depicts the graph building operations that can be performed by the MRSC Toolkit. (Incomplete nodes are shown by dashed lines,

[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.

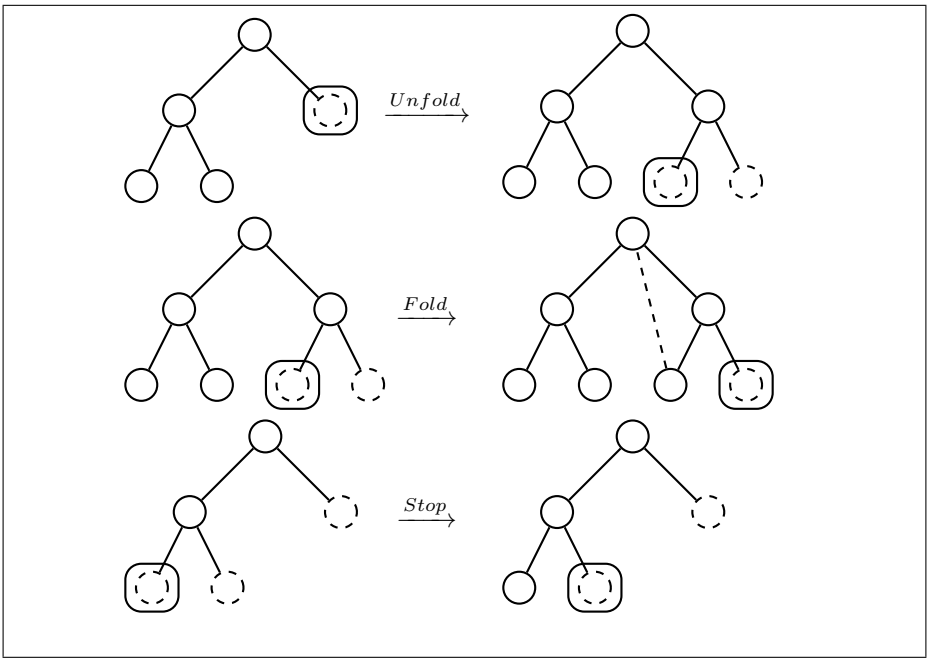

Figure 4: Basic operations of MRSC

$I(c)$	-	decomposition
$E(y \mapsto t_1,\ t_{rec})$	-	case analysis and (possible) recursion

Figure 5: Labels on graph edges in TT Lite SC

the current node is inside 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.

4.1 Graphs of configurations

The MRSC toolkit allows the nodes and edges of a graph to hold arbitrary information. 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: $\boxed{t \mid \Gamma}$. The edges of a graph can be assigned labels of two kinds (Fig. 5). The first kind corresponds to the elimination 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) $\boxed{t_0 \mid \Gamma_0} \xrightarrow{I(Cons)} \boxed{t_1 \mid \Gamma_1}$

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

(c) $\boxed{t_0 \mid \Gamma_0} \longleftarrow \boxed{t_1 \mid \Gamma_1}$

(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 an unfolding. (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 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 can be found in Appendix B. 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.

4.2 Unfolding rules

All rules enumerated in Appendix B, except for Fold, Whistle and Default, add new nodes to the graph by applying the operation Unfold. Any unfolding rule can be classified as either a decomposition or a case analysis by means of an eliminator.

4.2.1 Decomposition

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 new nodes, and label the edges with some information about the construct that has been decomposed. The peculiarity of decomposition is that it makes no analysis of type information, only examining the structure of values. For example, when the supercompiler encounters $\sigma\,(\Sigma(x : A).\,B(x))\,t_1\,t_2$, it supercompiles t_1 and t_2, but does not touch $\Sigma(x : A).\,B(x)$. The reason is that supercompilation is required to preserve the types of expressions, but the supercompilation of a type parameter can, in general, change the type of the expression.

Decomposition of binders is not performed, either. 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. Thus we do not decompose Π, Σ and λ.

Note that decomposition does not change the context.

4.2.2 Case analysis for neutral eliminators

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 labeling the corresponding edge with information about this instantiation. In the case of "ordinary" types, this information is enough for generating both the residual program and the proof of correctness (in a straightforward way). However, an attempt to perform case analysis for the eliminator $elim\ (\mathcal{I}\ A\ x\ y)$ and then generate a residual program would require solving a system of equations (for example, by means of higher-order unification). Thus, to keep our proof-of-concept supercompiler simple, driving is just not performed for $elim\ (\mathcal{I}\ A\ x\ y)$.

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. So, to keep the supercompiler simple, we prefer not to decompose such applications.

When dealing with eliminators for recursive types (such as $\mathbb{N}$ and $List$), we record the expression corresponding to the "previous step of elimination"[9] in the edge label. For example, for the expression $elim\ \mathbb{N}\ m\ f_0\ f_s\ (Succ\ n)$, the expression corresponding to the previous step is $elim\ \mathbb{N}\ m\ f_0\ f_s\ n$[10].

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

[10] For eliminators implemented in TT Lite (lists, Peano numbers) there is at maximum one recursive call on each branch. For cases when there are more than one recursive calls to the eliminator on the same branch (e.g. tree data type) labeling will be more technically

4.3 Folding rule

In the rule *Fold* $anc(\beta)$ is a set of ancestor nodes of the current node β. The rule itself is very simple. If 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.

4.4 Default rule

If no folding/unfolding 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.

4.5 Stop rule

In general, the process of repeatedly applying unfolding rules together with the rules *Fold* and *Default* may never terminate. Thus, in order to ensure termination, we use the rule *Whistle*, 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 as is into the residual program.

4.6 Termination

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.

4.7 Code generation

The generation of the residual program corresponding to a completed graph of configurations is performed just by recursive descent. The function that implements the residualization algorithm is defined in Appendix C. A call to this function has the form $\mathcal{C}[\alpha]_\rho$, where α is the current node, and ρ is an enviroment (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.

---

involved, but the essence will be the same.

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.) To avoid cluttering the notation, we use the following conventions. The current node is α, $tp(\alpha)$ is the type of the expression appearing in the node α. If $t \mid \Gamma$ is the configuration in the node α, and $\Gamma \vdash t : T$, then $tp(\alpha) = T$.

5 Proof generation

The generation of the proof corresponding to a completed graph of configurations is performed by recursive descent (and in this respect is similar to the generation of residual program). The function that implements the proof generation algorithm is defined in Appendix D. A call to this function has the form $\mathcal{P}[\alpha]_{\rho,\phi}$. Where α is the current node, ρ is an environment for folding of code generation and ϕ is an environment for folding of proof generation, ϕ is an analog of ρ in the world of proofs, – ϕ binds a node to an inductive hypothesis of the proof. However, ρ is used for two purposes (see explanations below). 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.

A proof generated by $\mathcal{P}$ is 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 (that arguments of constructors are equal) with the aid of the combinators $cong_1$ and $cong_2$ (defined in Figure 22).

- The proofs corresponding to eliminators are by (structural) induction. When specifying a type of a proof for eliminator (as a motive), $\mathcal{C}[\alpha]_\rho$ is used as during code generation (the same environment ρ). But when generating a subproof for recursive eliminators (see rules for $elim\ \mathcal{N}$ and $elim\ (ListA)$), an environment ρ is extended with a recursive call of a supercompiled eliminator. Also ϕ is extended with a mapping of the current node to a subproof (inductive hypothesis) to "fold" a proof.

Now we can explain why, when constructing graphs of configurations, we do not decompose λ-abstractions (in spite of the fact that this is done in [6]). We could have used the graph building rule

$\boxed{\lambda\ (x : T).e \mid \Gamma} \xrightarrow{I(\lambda(T))} \boxed{e \mid \Gamma, x : T}$

and the corresponding code generation rule

$\mathcal{C}\ \left[\ \xrightarrow{I(\lambda(T))} \alpha_1\ \right]_\rho = I(\lambda(x : T)\ \mathcal{C}[\alpha_1]_\rho)$

However, it would be unclear how to generate a proof of correctness (based on intensional equality)?

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 [1], such reuse would be problematic, which would produce a negative effect on the modularity of our design.

6 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.

The main limitation of the current version of TT Lite SC is that it does not perform generalization of configurations. It would be interesting to investigate generalization in the context of certifying supercompilation.

References

[1] M. Bolingbroke and S. Peyton Jones. Supercompilation by evaluation. In *Proceedings of the third ACM Haskell symposium on Haskell*, pages 135–146. ACM, 2010.

[2] R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012.

[3] G. Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9:355–372, 1999.

[4] A. Klimov and S. Romanenko, editors. *Third International Valentin Turchin Workshop on Metacomputation in Russia*. Publishing House "University of Pereslavl", 2012.

[5] A. V. Klimov, I. G. Klyuchnikov, and S. A. Romanenko. Automatic verification of counter systems via domain-specific multi-result supercompilation. In Klimov and Romanenko [4], pages 112–141.

[6] I. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009. URL: `http://library.keldysh.ru/preprint.asp?id=2009-63`.

[7] I. Klyuchnikov. *Inferring and proving properties of functional programs by means of supercompilation*. PhD thesis, Keldysh Institute of Applied Mathematics, 2010.

[8] I. Klyuchnikov. Supercompiler HOSC: proof of correctness. Preprint 31, Keldysh Institute of Applied Mathematics, Moscow, 2010. URL: `http://library.keldysh.ru/preprint.asp?id=2010-31`.

[9] I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *Proceedings of the 7th international Andrei Ershov Memorial conference on Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 193–205. Springer, 2010.

[10] I. Klyuchnikov and S. Romanenko. Towards higher-level supercompilation. In Nemytykh [16].

[11] I. G. Klyuchnikov and S. A. Romanenko. MRSC: a toolkit for building multi-result supercompilers. Preprint 77, Keldysh Institute of Applied Mathematics, 2011. URL: `http://library.keldysh.ru/preprint.asp?lg=e&id=2011-77`.

[12] D. Krustev. A simple supercompiler formally verified in Coq. In Nemytykh [16], pages 102–127.

[13] A. Lisitsa and A. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.

[14] A. Löh, C. McBride, and W. Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta Informaticae*, 21:1001–1032, 2010.

[15] C. McBride. Elimination with a motive. In *Selected papers from the International Workshop on Types for Proofs and Programs*, volume 2277 of *LNCS*. Springer, 2002.

[16] A. Nemytykh, editor. *Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia*. Publishing House "University of Pereslavl", 2010.

[17] A. P. Nemytykh and V. A. Pinchuk. Program transformation with metasystem transitions: Experiments with a supercompiler. In *Perspectives of System Informatics*, volume 1181 of *LNCS*. Springer, 1996.

[18] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's type theory*. Oxford University Press, 1990.

[19] A. Pardo and S. da Rosa. Program transformation in Martin-Löf's type theory. In *CADE-12, Workshop on Proof-search in type-theoretic languages*, 1994.

[20] S. Thompson. *Type theory and functional programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1991.

[21] V. F. Turchin. The language Refal: The theory of compilation and metasystem analysis. Technical Report 20, Courant Institute, 1980.

[22] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.

[23] V. F. Turchin. Supercompilation: Techniques and results. In *Perspectives of System Informatics*, volume 1181 of *LNCS*. Springer, 1996.

[24] P. van der Walt and W. Swierstra. Engineering proof by reflection in Agda. In R. Hinze, editor, *Implementation and Application of Functional Languages - 24th International Symposium*, 2012.

A Formation, Introduction and Elimination Rules

(v)	$\Gamma,\ x : T \vdash x : [\![T]\!]_\Gamma$
$([\![v]\!])$	$[\![x]\!]_{\Gamma,\ x:=t} = [\![t]\!]_\Gamma$

Figure 6: TT: variables

$(\mathcal{U})$	$\mathcal{U}_n : \mathcal{U}_{n+1}$

Figure 7: TT: universes

(ΠF)	$\dfrac{\Gamma \vdash A : \mathcal{U}_m \quad \Gamma, x : [\![A]\!]_\Gamma \vdash B(x) : \mathcal{U}_n}{\Gamma \vdash \Pi(x : A).\, B(x) : \mathcal{U}_{max(m,n)}}$
(ΠI)	$\dfrac{\Gamma \vdash A : \mathcal{U}_m \quad \Gamma, x : [\![A]\!]_\Gamma \vdash t(x) : B(x)}{\Gamma \vdash \lambda(x : A).\, t(x) : [\![\Pi(x : A).\, B(x)]\!]_\Gamma}$
(ΠE)	$\dfrac{\Gamma \vdash f : \Pi(x : A).\, B(x) \quad \Gamma \vdash t : A}{\Gamma \vdash f\ t : [\![B(x)]\!]_{\Gamma, x:=t}}$
$([\![\Pi F]\!])$	$[\![\Pi(x : A).\, B(x)]\!]_\Gamma = \Pi(x : [\![A]\!]_\Gamma).\, [\![B(x)]\!]_\Gamma$
$([\![\Pi I]\!])$	$[\![\lambda(x : A).\, t(x)]\!]_\Gamma = \lambda(x : [\![A]\!]_\Gamma).\, [\![t(x)]\!]_\Gamma$
$([\![\Pi E]\!])$	$[\![f\ t]\!]_\Gamma = \lceil [\![f]\!]_\Gamma [\![t]\!]_\Gamma \rceil_\Gamma$
$(\lceil \Pi E \rceil)$	$\lceil (\lambda(x : A).t(x))u \rceil_\Gamma = [\![t(x)]\!]_{\Gamma, x:=u}$

Figure 8: TT: dependent functions (products)

(ΣF)	$\dfrac{\Gamma \vdash A : \mathcal{U}_m \quad \Gamma, x : [\![A]\!]_\Gamma \vdash B(x) : \mathcal{U}_n}{\Gamma \vdash \Sigma(x : A).\, B(x) : \mathcal{U}_{max(m,n)}}$
(ΣI)	$\dfrac{\Gamma \vdash \Sigma(x : A).\, B(x) : \mathcal{U}_k \quad \Gamma, t_1 : [\![A]\!]_\Gamma \vdash t_2 : [\![B(x)]\!]_{\Gamma, x:=t_1}}{\Gamma \vdash \sigma\,(\Sigma(x : A).\, B(x))\, t_1\, t_2 : [\![\Sigma(x : A).\, B(x)]\!]_\Gamma}$
(ΣE)	$\dfrac{\begin{array}{c}\Gamma \vdash \Sigma(x : A).\, B(x) : \mathcal{U}_k \quad \Gamma \vdash p : [\![\Sigma(x : A).\, B(x)]\!]_\Gamma \\ \Gamma \vdash m : [\![\Pi(z : \Sigma(x : A).\, B(x))\ \mathcal{U}_k]\!]_\Gamma \\ \Gamma \vdash f : [\![\Pi(x : A).\, \Pi(y : B(x)).\, m\,(\sigma\,(\Sigma(x : A).\, B(x))\, x\, y)]\!]_\Gamma\end{array}}{elim\,(\Sigma(x : A).\, B)\, m\, f\, p : [\![m\, p]\!]_\Gamma}$
$([\![\Sigma F]\!])$	$[\![\Sigma(x : A).\, B(x)]\!]_\Gamma = \Sigma(x : [\![A]\!]_\Gamma).\, [\![B(x)]\!]_\Gamma$
$([\![\Sigma I]\!])$	$[\![\sigma\,(\Sigma(x : A).\, B(x))\, t_1\, t_2]\!]_\Gamma = \sigma\, [\![\Sigma(x : A).\, B(x)]\!]_\Gamma\, [\![t_1]\!]_\Gamma\, [\![t_2]\!]_\Gamma$
$([\![\Sigma E]\!])$	$[\![elim\,(\Sigma(x : A).\, B(x))\, m\, f\, p]\!]_\Gamma = \lceil elim\, [\![\Sigma(x : A).\, B(x)]\!]_\Gamma\, [\![m]\!]_\Gamma\, [\![f]\!]_\Gamma\, [\![p]\!]_\Gamma \rceil_\Gamma$
$(\lceil \Sigma E \rceil)$	$\lceil elim\,(\Sigma(x : A).\, B(x))\, m\, f\,(\sigma\,(\Sigma(x : A).\, B(x))\, t_1\, t_2) \rceil_\Gamma = [\![f\ t_1\ t_2]\!]_\Gamma$

Figure 9: TT: dependent sums (pairs)

$$(+F) \quad \frac{\Gamma \vdash A : \mathcal{U}_m \quad \Gamma \vdash B : \mathcal{U}_n}{\Gamma \vdash A + B : \mathcal{U}_{max(m,n)}}$$

$$(+I_1) \quad \frac{\Gamma \vdash A + B : \mathcal{U}_k \quad \Gamma \vdash a : [\![A]\!]_\Gamma}{\Gamma \vdash Inl\ (A + B)\ a : [\![(A + B)]\!]_\Gamma}$$

$$(+I_2) \quad \frac{\Gamma \vdash A + B : \mathcal{U}_k \quad \Gamma \vdash b : [\![B]\!]_\Gamma}{\Gamma \vdash Inr\ (A + B)\ b : [\![(A + B)]\!]_\Gamma}$$

$$(+E) \quad \frac{\begin{array}{c}\Gamma \vdash A + B : \mathcal{U}_k \quad \Gamma \vdash m : [\![\Pi(x : A + B).\ \mathcal{U}_k]\!]_\Gamma \\ \Gamma \vdash f_l : [\![\Pi(x : A).\ m\ (Inl\ (A + B)\ x)]\!]_\Gamma \\ \Gamma \vdash f_r : [\![\Pi(x : A).\ m\ (Inr\ (A + B)\ x)]\!]_\Gamma \\ \Gamma \vdash t : [\![A + B]\!]_\Gamma\end{array}}{\Gamma \vdash elim\ (A + B)\ m\ f_l\ f_r\ t : [\![m\ t]\!]_\Gamma}$$

$$([\![+F]\!]) \quad [\![A + B]\!]_\Gamma = [\![A]\!]_\Gamma + [\![B]\!]_\Gamma$$

$$([\![+I_1]\!]) \quad [\![Inl\ (A + B)\ a]\!]_\Gamma = Inl\ [\![(A + B)]\!]_\Gamma\ [\![a]\!]_\Gamma$$

$$([\![+I_2]\!]) \quad [\![Inr\ (A + B)\ b]\!]_\Gamma = Inr\ [\![(A + B)]\!]_\Gamma\ [\![b]\!]_\Gamma$$

$$([\![+E]\!]) \quad \begin{array}{c}[\![elim\ (A + B)\ m\ f_l\ f_r\ v]\!]_\Gamma = \\ \lceil elim\ [\![A + B]\!]_\Gamma\ [\![m]\!]_\Gamma\ [\![f_l]\!]_\Gamma\ [\![f_r]\!]_\Gamma\ [\![v]\!]_\Gamma \rceil_\Gamma\end{array}$$

$$(\lceil +E_1 \rceil) \quad \lceil elim\ (A + B)\ m\ f_l\ f_r\ (Inl\ (A + B)\ a) \rceil_\Gamma = [\![f_l\ a]\!]_\Gamma$$

$$(\lceil +E_2 \rceil) \quad \lceil elim\ (A + B)\ m\ f_l\ f_r\ (Inr\ (A + B)\ b) \rceil_\Gamma = [\![f_r\ b]\!]_\Gamma$$

Figure 10: TT: coproducts (sums)

$$(\bot F) \quad \frac{}{\Gamma \vdash \bot : \mathcal{U}_0}$$

$$(\top F) \quad \frac{}{\Gamma \vdash \top : \mathcal{U}_0}$$

$$(\top I) \quad \frac{}{\Gamma \vdash \star : \top}$$

$$(\bot E) \quad \frac{\Gamma \vdash m : [\![\Pi(x : \bot).\ \mathcal{U}_k]\!]_\Gamma \quad \Gamma \vdash t : \bot}{\Gamma \vdash elim\ \bot\ m\ t : [\![m\ t]\!]_\Gamma}$$

$$(\top E) \quad \frac{\Gamma \vdash m : [\![\Pi(x : \top).\ \mathcal{U}_k]\!]_\Gamma \quad \Gamma \vdash f : [\![m\ \star]\!]_\Gamma \quad \Gamma \vdash t : \top}{\Gamma \vdash elim\ \top\ m\ f\ t : [\![m\ t]\!]_\Gamma}$$

$$([\![\bot E]\!]) \quad [\![elim\ \bot\ m\ t]\!]_\Gamma = elim\ \bot\ [\![m]\!]_\Gamma\ [\![t]\!]_\Gamma$$

$$([\![\top E]\!]) \quad [\![elim\ \top\ m\ f\ t]\!]_\Gamma = \lceil elim\ \top\ [\![m]\!]_\Gamma\ [\![f]\!]_\Gamma\ [\![t]\!]_\Gamma \rceil_\Gamma$$

$$(\lceil \top E \rceil) \quad \lceil elim\ \top\ m\ f\ \star \rceil_\Gamma = [\![f\ \star]\!]_\Gamma$$

Figure 11: TT: the empty (bottom) type and the unit type

$$(\mathbb{N}F) \quad \frac{}{\Gamma \vdash \mathbb{N} : \mathcal{U}_0}$$

$$(\mathbb{N}I_1) \quad \frac{}{\Gamma \vdash 0 : \mathbb{N}}$$

$$(\mathbb{N}I_2) \quad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash Succ\ n : \mathbb{N}}$$

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

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

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

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

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

Figure 12: TT: natural numbers

$$(List\ F) \quad \frac{\Gamma \vdash A : \mathcal{U}_k}{\Gamma \vdash List\ A : \mathcal{U}_k}$$

$$(List\ I_1) \quad \frac{\Gamma \vdash List\ A : \mathcal{U}_k}{\Gamma \vdash Nil\ (List\ A) : [\![List\ A]\!]_\Gamma}$$

$$(List\ I_2) \quad \frac{\Gamma \vdash List\ A : \mathcal{U}_k \quad \Gamma \vdash t_1 : [\![A]\!]_\Gamma \quad \Gamma \vdash t_2 : [\![List\ A]\!]_\Gamma}{\Gamma \vdash Cons\ (List\ A)\ t_1\ t_2 : [\![List\ A]\!]_\Gamma}$$

$$(List\ E) \quad \frac{\begin{array}{c}\Gamma \vdash List\ A : \mathcal{U}_n \quad \Gamma \vdash m : [\![\Pi(x : List\ A).\ \mathcal{U}_k]\!]_\Gamma \\ \Gamma \vdash f_s : [\![\Pi(x : A)\ (y : List\ A)\ (z : m\ y).\ m\ (Cons\ A\ x\ y)]\!]_\Gamma \\ \Gamma \vdash t : [\![List\ A]\!]_\Gamma \quad \Gamma \vdash f_0 : [\![m\ (Nil\ A)]\!]_\Gamma\end{array}}{\Gamma \vdash elim\ (List\ A)\ m\ f_0\ f_s\ t : [\![m\ t]\!]_\Gamma}$$

$$([\![List\ F]\!]) \quad [\![List\ A]\!]_\Gamma = List\ [\![A]\!]_\Gamma$$

$$([\![List\ I_1]\!]) \quad [\![Nil\ (List\ A)]\!]_\Gamma = Nil\ [\![List\ A]$$

$$([\![List\ I_2]\!]) \quad [\![Cons\ (List\ A)\ t_1\ t_2]\!]_\Gamma = Cons\ [\![List\ A]\!]_\Gamma\ [\![t_1]\!]_\Gamma\ [\![t_2]\!]_\Gamma$$

$$([\![List\ E]\!]) \quad \begin{array}{c}[\![elim\ (List\ A)\ m\ f_0\ f_1\ t]\!]_\Gamma = \\ \lceil elim\ [\![List\ A]\!]_\Gamma\ [\![m]\!]_\Gamma\ [\![f_0]\!]_\Gamma\ [\![f_1]\!]_\Gamma\ [\![t]\!]_\Gamma \rceil_\Gamma\end{array}$$

$$(\lceil List\ E_1 \rceil) \quad \lceil elim\ (List\ A)\ m\ f_0\ f_1\ (Nil\ (List\ A)) \rceil_\Gamma = [\![f_0]\!]_\Gamma$$

$$(\lceil List\ E_2 \rceil) \quad \begin{array}{c}\lceil elim\ (List\ A)\ m\ f_0\ f_1\ (Cons\ (List\ A)\ t_1\ t_2) \rceil_\Gamma = \\ [\![f_1\ t_1\ t_2\ (elim\ (List\ A)\ m\ f_0\ f_s\ t_2)]\!]_\Gamma\end{array}$$

Figure 13: TT: lists

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

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

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

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

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

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

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

Figure 14: TT: equality

$$\begin{array}{llll}
nv(x\ t) & = x & nv(elim\ \bot\ m\ x) & = x \\
nv(t_1\ t_2) & = nv(t_1) & nv(elim\ \bot\ m\ t) & = nv(t) \\
nv(elim\,(\Sigma(x : A).\,B)\,m\,f\,x) & = x & nv(elim\ \top\ m\ f\ x) & = x \\
nv(elim\,(\Sigma(x : A).\,B)\,m\,f\,t) & = nv(t) & nv(elim\ \top\ m\ f\ t) & = nv(t) \\
nv(elim\ (A + B)\ m\ f_l\ f_r\ x) & = x & nv(elim\ \mathbb{N}\ m\ f_0\ f_s\ x) & = x \\
nv(elim\ (A + B)\ m\ f_l\ f_r\ t) & = nv(t) & nv(elim\ \mathbb{N}\ m\ f_0\ f_s\ t) & = nv(t) \\
nv(elim\ (List\ A)\ m\ f_0\ f_s\ x) & = x & nv(elim\,(\mathcal{I}\ A\ t_1\ t_2)\,m\,f\,x) & = x \\
nv(elim\ (List\ A)\ m\ f_0\ f_s\ t) & = nv(t) & nv(elim\,(\mathcal{I}\ A\ t_1\ t_2)\,m\,f\,t) & = nv(t)
\end{array}$$

Figure 15: Finding the neutral variable of a term

B Supercompilation rules

(Fold) $$\frac{\exists \alpha \in anc(\beta) : \alpha \xrightarrow{E(y \mapsto t_1, c)} \alpha_1}{\alpha \leftarrow \beta}$$

(WH) $$\frac{depth(\beta) > n}{\boxed{c \mid \Gamma} \xrightarrow{Stop(c)} \bullet}$$

($\Sigma I'$) $$\boxed{\sigma\,(\Sigma(x : A).\,B(x))\,t_1\,t_2 \mid \Gamma} \begin{array}{l} \xrightarrow{I(\sigma)} \boxed{t_1 \mid \Gamma} \\ \xrightarrow{I(\sigma)} \boxed{t_2 \mid \Gamma} \end{array}$$

($\Sigma E'$) $$\frac{nv(c) = y \quad y : (\Sigma(x : A).\,B(x)) \quad t_1 = \sigma\,(\Sigma(x : A).\,B(x))\,v_1\,v_2}{\boxed{c \mid \Gamma} \xrightarrow{E(y \mapsto t_1, \bullet)} \boxed{[\![c]\!]_{\Gamma, y := t_1} \mid \Gamma, v_1 : A, v_2 : B(v_1)}}$$

($+F'$) $$\boxed{t_1 + t_2 \mid \Gamma} \begin{array}{l} \xrightarrow{I(+)} \boxed{t_1 \mid \Gamma} \\ \xrightarrow{I(+)} \boxed{t_2 \mid \Gamma} \end{array}$$

($+I_1'$) $$\boxed{Inl\,(A + B)\,t \mid \Gamma} \xrightarrow{I(Inl)} \boxed{t \mid \Gamma}$$

($+I_2'$) $$\boxed{Inr\,(A + B)\,t \mid \Gamma} \xrightarrow{I(Inr)} \boxed{t \mid \Gamma}$$

($+E'$) $$\frac{nv(c) = y \quad y : A + B \quad t_1 = Inl\,(A + B)\,v_1 \quad t_2 = Inl\,(A + B)\,v_2}{\boxed{c \mid \Gamma} \begin{array}{l} \xrightarrow{E(y \mapsto t_1, \bullet)} \boxed{[\![c]\!]_{\Gamma, y := t_1} \mid \Gamma, v_1 : A} \\ \xrightarrow{E(y \mapsto t_2, \bullet)} \boxed{[\![c]\!]_{\Gamma, y := t_2} \mid \Gamma, v_2 : B} \end{array}}$$

($\top E'$) $$\frac{nv(c) = y \quad y : \top \quad t_1 = \star}{\boxed{c \mid \Gamma} \xrightarrow{E(y \mapsto t_1, \bullet)} \boxed{[\![c]\!]_{\Gamma, y := t_1} \mid \Gamma}}$$

Figure 16: Building graphs of configurations: part 1

$(\mathbb{N}I_2')$ $\quad \boxed{Succ\ t \mid \Gamma} \xrightarrow{I(Succ)} \boxed{t \mid \Gamma}$

$(\mathbb{N}E')$
$$\frac{nv(c) = y \quad y : \mathbb{N} \quad t_1 = 0 \quad t_2 = Succ\ v_1 \quad t_r = [\![c]\!]_{\Gamma, y:=v_1}}{\boxed{c \mid \Gamma} \xrightarrow{E(y \mapsto t_1, \bullet)} \boxed{[\![c]\!]_{\Gamma, y:=t_1} \mid \Gamma} \quad \boxed{c \mid \Gamma} \xrightarrow{E(y \mapsto t_2, t_r)} \boxed{[\![c]\!]_{\Gamma, y:=t_2} \mid \Gamma, v_1 : \mathbb{N}}}$$

$(\mathcal{I}\ F')$ $\quad \boxed{\mathcal{I}\ A\ t_1\ t_2 \mid \Gamma} \xrightarrow{I(\mathcal{I})} \boxed{t_1 \mid \Gamma}$, $\boxed{\mathcal{I}\ A\ t_1\ t_2 \mid \Gamma} \xrightarrow{I(\mathcal{I})} \boxed{t_2 \mid \Gamma}$

$(\mathcal{I}I')$ $\quad \boxed{Refl\ A\ t \mid \Gamma} \xrightarrow{I(Refl)} \boxed{t \mid \Gamma}$

$(List\ F')$ $\quad \boxed{List\ t \mid \Gamma} \xrightarrow{I(List)} \boxed{A \mid \Gamma}$

$(List\ I_2')$ $\quad \boxed{Cons\ (List\ A)\ t_1\ t_2 \mid \Gamma} \xrightarrow{I(Cons)} \boxed{t_1 \mid \Gamma}$, $\boxed{Cons\ (List\ A)\ t_1\ t_2 \mid \Gamma} \xrightarrow{I(Cons)} \boxed{t_2 \mid \Gamma}$

$(List\ E')$
$$\frac{nv(c) = y \quad y : List\ A \quad t_1 = Nil\ (List\ A) \quad t_2 = Cons\ (List\ A)\ v_1\ v_2 \quad t_r = [\![c]\!]_{\Gamma, y:=v_2}}{\boxed{c \mid \Gamma} \xrightarrow{E(y \mapsto t_1, \bullet)} \boxed{[\![c]\!]_{\Gamma, y:=t_1} \mid \Gamma} \quad \boxed{c \mid \Gamma} \xrightarrow{E(y \mapsto t_2, t_r)} \boxed{[\![c]\!]_{\Gamma, y:=t_2} \mid \Gamma, v_1 : A, v_2 : List\ A}}$$

(Default) $\quad \boxed{c \mid \Gamma} \xrightarrow{Stop} \bullet$

Figure 17: Building graphs of configurations: part 2

C Code generation

$$\mathcal{C}\ \left[\beta \leftarrow\ \right]_\rho = \rho(\beta)$$

$$\mathcal{C}\ \left[\rightarrow \bullet\ \right]_\rho = \alpha.c$$

$$\mathcal{C}\ \left[\begin{array}{l} \xrightarrow{I(\sigma)} \alpha_1 \\ \xrightarrow{I(\sigma)} \alpha_2 \end{array}\right]_\rho = \sigma\ tp(\alpha)\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{C}[\alpha_2]_\rho$$

$$\mathcal{C}\left[\xrightarrow{E(y\mapsto\sigma\ (\Sigma(x:A).\,B(x))\ v_1\ v_2,\bullet)} \alpha_1\ \right]_\rho = \begin{array}{l} elim\ (\Sigma(x:A).\,B(x)) \\ \quad (\lambda(y:\Sigma(x:A).\,B(x)).\,tp(\alpha)) \\ \quad (\lambda(v_1:A)\ (v_2:B(v_1)).\,\mathcal{C}[\alpha_1]_\rho) \\ \quad y \end{array}$$

$$\mathcal{C}\ \left[\begin{array}{l} \xrightarrow{I(+)} \alpha_1 \\ \xrightarrow{I(+)} \alpha_2 \end{array}\right]_\rho = \mathcal{C}[\alpha_1]_\rho + \mathcal{C}[\alpha_2]_\rho$$

$$\mathcal{C}\ \left[\xrightarrow{I(Inl)} \alpha_1\ \right]_\rho = Inl\ tp(\alpha)\ \mathcal{C}[\alpha_1]_\rho$$

$$\mathcal{C}\ \left[\xrightarrow{I(Inr)} \alpha_1\ \right]_\rho = Inr\ tp(\alpha)\ \mathcal{C}[\alpha_1]_\rho$$

$$\mathcal{C}\left[\begin{array}{l} \xrightarrow{E(y\mapsto Inl\ (A+B)\ v_1,\bullet)} \alpha_1 \\ \xrightarrow{E(y\mapsto Inr\ (A+B)\ v_2,\bullet)} \alpha_2 \end{array}\right]_\rho = \begin{array}{l} elim\ (A+B)\ (\lambda(y:A+B).\,tp(\alpha)) \\ \quad (\lambda(v_1:A).\,\mathcal{C}[\alpha_1]_\rho) \\ \quad (\lambda(v_2:B).\,\mathcal{C}[\alpha_2]_\rho) \\ \quad y \end{array}$$

$$\mathcal{C}\left[\xrightarrow{E(y\mapsto\star,\bullet)} \alpha_1\ \right]_\rho = elim\ \top\ (\lambda(y:\top).\,tp(\alpha))\ \mathcal{C}[\alpha_1]_\rho\ y$$

$$\mathcal{C}\left[\xrightarrow{I(Succ)} \alpha_1\ \right]_\rho = Succ\ \mathcal{C}[\alpha_1]_\rho$$

$$\mathcal{C}\left[\begin{array}{l} \xrightarrow{E(y\mapsto 0,\bullet)} \alpha_1 \\ \xrightarrow{E(y\mapsto Succ\ v_2,r)} \alpha_2 \end{array}\right]_\rho = \begin{array}{l} elim\ \mathbb{N}\ (\lambda(y:\mathbb{N}).\,tp(\alpha)) \\ \quad \mathcal{C}[\alpha_1]_\rho \\ \quad (\lambda(v_2:\mathbb{N})(v_3:(\lambda((y:\mathbb{N}).\,tp(\alpha))\ v_2). \\ \qquad \mathcal{C}[\alpha_2]_{\rho+(\alpha\to v_3)}) \\ \quad y \end{array}$$

Figure 18: Generating residual expressions: part 1

$$\mathcal{C}\left[\begin{array}{l}\xrightarrow{I(\mathcal{I})}\alpha_1\\ \xrightarrow{I(\mathcal{I})}\alpha_2\end{array}\right]_\rho = \mathcal{I}\ tp(\alpha)\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{C}[\alpha_2]_\rho$$

$$\mathcal{C}\left[\xrightarrow{I(Refl)}\alpha_1\right]_\rho = Refl\ tp(\alpha)\ \mathcal{C}[\alpha_1]_\rho$$

$$\mathcal{C}\left[\xrightarrow{I(List)}\alpha_1\right]_\rho = List\ \mathcal{C}[\alpha_1]_\rho$$

$$\mathcal{C}\left[\begin{array}{l}\xrightarrow{I(Cons)}\alpha_1\\ \xrightarrow{I(Cons)}\alpha_2\end{array}\right]_\rho = Cons\ tp(\alpha)\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{C}[\alpha_2]_\rho$$

$$\mathcal{C}\left[\begin{array}{l}\xrightarrow{E(y\mapsto Nil\ (List\ A),\bullet)}\alpha_1\\ \xrightarrow{E(y\mapsto Cons\ (List\ A)\ v_2\ v_3,r)}\alpha_2\end{array}\right]_\rho = \begin{array}{l} elim\ (List\ A)\ (\lambda(y:(List\ A)).\ tp(\alpha))\\ \quad \mathcal{C}[\alpha_1]_\rho\\ \quad (\lambda(v_2:A)(v_3:List\ A)\\ \quad\quad (v_4:(\lambda(y:(List\ A)).\ tp(\alpha))\ v_3).\\ \quad\quad\quad \mathcal{C}[\alpha_2]_{\rho+(\alpha\to v_4)})\\ \quad y\end{array}$$

Figure 19: Generating residual expressions: part 2

D Proof generation

$$\mathcal{P}\ \left[\beta \leftarrow\ \right]_{\rho,\phi} = \phi(\beta)$$

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

$$\mathcal{P}\ \left[\begin{array}{l}\xrightarrow{I(\sigma)}\ \alpha_1\\ \xrightarrow{I(\sigma)}\ \alpha_2\end{array}\right]_{\rho,\phi} = \begin{array}{l} cong_2\ tp(\alpha_1)\ tp(\alpha_2)\ tp(\alpha)\ (\lambda(\_ : tp(\alpha_1)).\ \sigma\ tp(\alpha)\ \alpha_1.c)\\ \alpha_1.c\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{P}[\alpha_1]_{\rho,\phi}\\ \alpha_2.c\ \mathcal{C}[\alpha_2]_\rho\ \mathcal{P}[\alpha_2]_{\rho,\phi}\end{array}$$

$$\mathcal{P}\left[\xrightarrow{E(y\mapsto\sigma\ (\Sigma(x:A).\ B(x))\ v_1\ v_2,\bullet)}\ \alpha_1\ \right]_{\rho,\phi} = \begin{array}{l} elim\ (\Sigma(x : A).\ B)\\ (\lambda(y : \Sigma(x : A).\ B(x)).\ \mathcal{I}\ tp(a)\ \alpha.c\ \mathcal{C}[\alpha]_\rho)\\ (\lambda(v_1 : A)\ (v_2 : B(v_2)).\ \mathcal{P}[\alpha_1]_{\rho,\phi})\\ y\end{array}$$

$$\mathcal{P}\ \left[\begin{array}{l}\xrightarrow{I(+)}\ \alpha_1\\ \xrightarrow{I(+)}\ \alpha_2\end{array}\right]_{\rho,\phi} = \begin{array}{l} cong_2\ tp(\alpha_1)\ tp(\alpha_2)\ tp(\alpha)\ (\_ + \_)\\ \alpha_1.c\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{P}[\alpha_1]_{\rho,\phi}\\ \alpha_2.c\ \mathcal{C}[\alpha_2]_\rho\ \mathcal{P}[\alpha_2]_{\rho,\phi}\end{array}$$

$$\mathcal{P}\ \left[\xrightarrow{I(Inl)}\ \alpha_1\ \right]_{\rho,\phi} = cong_1\ tp(\alpha_1)\ tp(\alpha)\ (Inl\ (tp(\alpha.c)))\ \alpha_1.c\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{P}[\alpha_1]_{\rho,\phi}$$

$$\mathcal{P}\ \left[\xrightarrow{I(Inr)}\ \alpha_1\ \right]_{\rho,\phi} = cong_1\ tp(\alpha_1)\ tp(\alpha)\ (Inr\ (tp(\alpha.c)))\ \alpha_1.c\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{P}[\alpha_1]_{\rho,\phi}$$

$$\mathcal{P}\ \left[\begin{array}{l}\xrightarrow{E(y\mapsto Inl\ (A+B)\ v_1,\bullet)}\ \alpha_1\\ \xrightarrow{E(y\mapsto Inr\ (A+B)\ v_2,\bullet)}\ \alpha_2\end{array}\right]_{\rho,\phi} = \begin{array}{l} elim\ (A + B)\\ (\lambda(y : A + B).\ \mathcal{I}\ tp(\alpha.c)\ \alpha.c\ \mathcal{C}[\alpha]_\rho)\\ (\lambda(v_1 : A).\ \mathcal{P}[\alpha_1]_{\rho,\phi})\\ (\lambda(v_2 : A).\ \mathcal{P}[\alpha_2]_{\rho,\phi})\\ y\end{array}$$

$$\mathcal{P}\ \left[\xrightarrow{E(y\mapsto\star,\bullet)}\ \alpha_1\ \right]_{\rho,\phi} = elim\ \top\ (\lambda(y : \top).\ \mathcal{I}\ tp(\alpha)\ \alpha.c\ \mathcal{C}[\alpha]_\rho)\ \mathcal{P}[\alpha_1]_{\rho,\phi}\ y$$

$$\mathcal{P}\ \left[\xrightarrow{I(Succ)}\ \alpha_1\ \right]_{\rho,\phi} = cong_1\ tp(\alpha_1)\ tp(\alpha)\ Succ\ \alpha_1.c\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{P}[\alpha_1]_{\rho,\phi}$$

$$\mathcal{P}\ \left[\begin{array}{l}\xrightarrow{E(y\mapsto 0,\bullet)}\ \alpha_1\\ \xrightarrow{E(y\mapsto Succ\ v_2,r)}\ \alpha_2\end{array}\right]_{\rho,\phi} = \begin{array}{l} elim\ \mathbb{N}\\ (\lambda(y : \mathbb{N}).\ \mathcal{I}\ tp(\alpha)\ \alpha.c\ \mathcal{C}[\alpha]_\rho)\ \mathcal{P}[\![\alpha_1]\!]_{\rho,\phi}\\ (\lambda(v_2 : \mathbb{N})(v_3 : (\lambda(y : \mathbb{N}).\ \mathcal{I}\ tp(\alpha)\ \alpha.c\ \mathcal{C}[\alpha]_\rho)\ v_2).\\ \qquad (\lambda(y : \mathbb{N}).\mathcal{P}[\alpha_2]_{\rho+(\alpha\to\mathcal{C}[\![\alpha]\!]_\rho),\phi+(\alpha\to v_3)})\ v_2)\ y\end{array}$$

Figure 20: Generating proofs of correctness: part 1

$$\mathcal{P}\left[\begin{array}{l}\xrightarrow{I(\mathcal{I})}\alpha_1\\ \xrightarrow{I(\mathcal{I})}\alpha_2\end{array}\right]_{\rho,\phi} = \begin{array}{l} cong_2\ tp(\alpha_1)\ tp(\alpha_2)\ tp(\alpha)\ (\mathcal{I}\ tp(\alpha_1))\\ \alpha_1.c\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{P}[\alpha_1]_{\rho,\phi}\\ \alpha_2.c\ \mathcal{C}[\alpha_2]_\rho\ \mathcal{P}[\alpha_2]_{\rho,\phi}\end{array}$$

$$\mathcal{P}\left[\xrightarrow{I(Refl)}\alpha_1\right]_{\rho,\phi} = cong_1\ tp(\alpha_1)\ tp(\alpha)\ (Refl\ tp(\alpha_1))\ \alpha_1.c\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{P}[\alpha_1]_{\rho,\phi}$$

$$\mathcal{P}\left[\xrightarrow{I(List)}\alpha_1\right]_{\rho,\phi} = cong_1\ tp(\alpha_1)\ tp(\alpha)\ List\ \alpha_1.c\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{P}[\alpha_1]_{\rho,\phi}$$

$$\mathcal{P}\left[\begin{array}{l}\xrightarrow{I(Cons)}\alpha_1\\ \xrightarrow{I(Cons)}\alpha_2\end{array}\right]_{\rho,\phi} = \begin{array}{l} cong_2\ tp(\alpha_1)\ tp(\alpha_2)\ tp(\alpha)\ (\lambda(\_ : tp(\alpha_1)).\ Cons\ tp(\alpha)\ \alpha_1.c)\\ \alpha_1.c\ \mathcal{C}[\alpha_1]_\rho\ \mathcal{P}[\alpha_1]_{\rho,\phi}\\ \alpha_2.c\ \mathcal{C}[\alpha_2]_\rho\ \mathcal{P}[\alpha_2]_{\rho,\phi}\end{array}$$

$$\mathcal{P}\left[\begin{array}{l}\xrightarrow{E(y\mapsto Nil\ (List\ A),\bullet)}\alpha_1\\ \xrightarrow{E(y\mapsto Cons\ (List\ A)\ v_2\ v_3,r)}\alpha_2\end{array}\right]_{\rho,\phi} = \begin{array}{l} elim\ (List\ A)\\ (\lambda(y : List\ A).\ \mathcal{I}\ tp(\alpha)\ \alpha.c\ \mathcal{C}[\alpha]_\rho)\\ \mathcal{P}[\alpha_1]_{\rho,\phi}\\ (\lambda(v_2 : A)(v_3 : List\ A)\\ \qquad (v_4 : (\lambda(y : List\ A).\\ \qquad\qquad \mathcal{I}\ tp(\alpha)\ \alpha.c\ \mathcal{C}[\alpha]_\rho)\ v_3).\\ \qquad\qquad (\lambda(y : List\ A).\\ \qquad\qquad \mathcal{P}[\alpha_2]_{\rho+(\alpha\to\mathcal{C}[\alpha]_\rho),\phi+(\alpha\to v_4)})v_4)\\ y\end{array}$$

Figure 21: Generating proofs of correctness: part 2

```
cong1 : Π(A : U_i)(B : U_j)(f : Π(_ : A).B)(x : A)(y : A)(_ : I A x y). I B (f x) (f y);
cong1 = λ(A : U_i)(B : U_j)(f : Π(_ : A).B)(x : A)(y : A)(i : I a x y).
          elim (I A x y)
          (λ(x : A)(y : A)(_ : I A x y).I B (f x) (f y))
          (λ(x : A).Refl B (f x))
          i;

cong2 : Π(A : U_i)(B : U_j)(C : U_k) (f : Π(_ : A)(_ : B).C)
      (x1 : A)(x2 : A)(_ : I A x1 x2)
      (y1 : B)(y2 : B)(_ : I B y1 y2). I C (f x1 y1) (f x2 y2);
cong2 = λ(A : U_i)(B : U_j)(C : U_k) (f : Π(_ : A)(_ : B).C)
      (x1 : A)(x2 : A)(i_x : I A x1 x2)
      (y1 : A)(y2 : A)(i_y : I B y1 y2).
        elim (I (Π(_ : B).C) (f x1) (f x2))
             (λ(g1 : Π(_ : B).C)(g2 : Π(_ : B).C)(_ : Id(Π(_ : B).C) g1 g2).
                I C (g1 y1) (g2 y2))
             (λ(g : Π(_ : B).C).cong1 B C g y1 y2 i_y)
             (cong1 A (Π(_ : B).C)f x1 x2 i_x);
```

Figure 22: Proof combinators