# A Supercompiler Assisting Its Own Formal Verification

Dimitur Krustev

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

**Abstract.** Program verification is a well known potential application of supercompilation. There are, however, few examples of using supercompilation for practical verification problems. We consider the correctness proof of the supercompiler itself as an interesting and practical task, on which to test the potential of supercompilation in this area. We show that even a simple supercompiler – treating a small first-order subset of Lisp, and working in cooperation with a traditional proof assistant (J-Bob) – can provide a lot of help for checking its own correctness.

## 1 Introduction

Supercompilation is a program transformation technique – a particularly strong form of partial evaluation [4,8] – originally proposed by Turchin [24], with a long history [12] and many potential applications: program optimization, software testing, program analysis, formal verification. While many of these applications were already described by Turchin in his early works, most of the research on supercompilation has concentrated for many years on program optimization. In recent years researchers show renewed interest in other applications, including analysis and verification of programs and other systems [11,13,18,19]. A good way to increase the adoption of supercompilation for verification is to demonstrate it is applicable and helpful for a wider range of practical problems. By analogy with self-application of supercompilers (and partial evaluators in general) – which has been a hot research topic for many years [4,20] – we consider verification of the supercompiler itself to be an interesting task. A challenge in the context of verification is how to trust the results of a supercompiler, especially if it is being used for its own verification. A good solution to this problem has recently been proposed by Klyuchnikov et al. [14]: a supercompiler producing not only a transformed program, but also a certificate proving the result is semantically equivalent to the input.

We take this idea a step further by showing that a supercompiler – which is both self-applicable and certifying – can be used successfully as a proof automation tool during the creation of its own correctness proof. Note that we do not speak of fully automatic self-verification: the supercompiler automatically generates only parts of its correctness proof. The overall proof is created manually in a traditional proof assistant. We argue that this organization is actually

an advantage: while typically supercompilers are used as automatic black-boxes, and if they fail to produce the desired result, their user is left with nothing, in our approach the supercompiler can be called many times during an interactive proof session, each time incrementally helping to get closer to the final goal. We outline as particular achievements of the proposed approach the following:

- a way to construct a supercompiler, which is both self-applicable and certifying (Sect. 3);
- an alternative two-phase approach to the construction of a certifying supercompiler: the supercompiler returns only a certificate, and the resulting program can be reconstructed from this certificate as a second step (Sect. 3.1);
- successful application of the described supercompiler as a generic proof automation tool in the context of a traditional proof assistant (Sect. 3.1);
- successful application of the supercompiler to fill in parts of its own correctness proofs (Sect. 4.2).

In Sect. 3 we discuss the overall architecture of our supercompiler and the adaptations made – compared to a classical supercompiler organization – in order to fit better as a proof automation tool inside the J-Bob proof assistant. One important omission is the lack of folding (that is, the supercompiler cannot produce new function definitions). It turns out folding is of no critical importance for our particular application domain, while its lack simplifies a lot both the supercompiler and its correctness proof. Section 4 discusses the more interesting details of the implementation itself, as well as some of the tricks used to simplify further the correctness proof. In Sect. 5 we analyze the performance of the proposed system in two aspects. First, we use the ratio of high-level proof steps (as entered manually by the user) versus low-level proof steps (obtained after the application of supercompilation) as a measure of the level of proof automation the system provides. Second, we analyze the processing time requirements of the overall system (supercompiler + underlying proof assistant), and how they can be improved.

We assume readers are familiar with supercompilation [13, 22, 24], but not necessarily with J-Bob. As J-Bob's principles of operation are important for understanding the rest of the article, we briefly review them in the following section.

## 2 J-Bob Crash Course

For this supercompiler verification experiment we use a proof assistant called J-Bob. It has been recently published [2] to accompany the new book "The Little Prover" [3] – a gentle introduction to program verification and proof assistants in general. J-Bob can be used to check proofs of properties of programs written in a first-order purely functional subset of Lisp. While this Lisp dialect contains only a few built-in data types and a handful of primitives, it can still be used to

write interesting and complicated programs. For example, J-Bob itself is written in the same Lisp subset it can reason about. Programs are a sequence of function definitions (ordered by the definition-use relation), with their bodies being expressions of 4 syntactic categories (Fig. 1): variables, constants, conditional expressions, and function calls (to both built-in primitives and defined functions). Only direct recursion is allowed and recursive function definitions must be accompanied by a proof of termination. The built-in data types consist only of atoms (natural numbers and symbols) and `cons` pairs. The built-in operations are: `equal`, `atom`, `car`, `cdr`, `cons`, `natp`, `size`, `+`, `<`. As a small example the function for concatenating 2 lists can be written as in Fig. 2.

$$
\begin{aligned}
\mathit{Exp} \ni e &::= x \mid \texttt{'}c \mid \texttt{(if } e_q\ e_a\ e_e\texttt{)} \mid \texttt{(}f\ e_1 \ldots e_n\texttt{)} \\
\mathit{Def} \ni \mathit{def} &::= \texttt{(defun } f\ \texttt{(}x_1 \ldots x_n\texttt{)}\ e_{body}\texttt{)} \\
\mathit{Prg} \ni p &::= \mathit{def}_1 \ldots \mathit{def}_n
\end{aligned}
$$

**Fig. 1.** Lisp syntax

```
(defun append (xs ys)
  (if (atom xs) ys
    (cons (car xs) (append (cdr xs) ys))))
```

**Fig. 2.** List append in Lisp

Unlike many other proof assistants J-Bob does not make a distinction between logical statements and Boolean expressions: logical statements *are* represented as Boolean expressions[1]. This feature is particularly useful for our purposes, as supercompilation deals easily with program expressions (including Boolean ones), but not with logical statements. The if-expression serves as the only built-in Boolean operation, but other operations are easily expressible, for example:

$$
\begin{aligned}
a \wedge b &\equiv \texttt{(if } a\ b\ \texttt{'nil)} \\
a \vee b &\equiv \texttt{(if } a\ \texttt{'t}\ b\texttt{)} \\
a \rightarrow b &\equiv \texttt{(if } a\ b\ \texttt{'t)}
\end{aligned}
$$

[1] As Lisp is dynamically typed, there is no static distinction between Boolean-valued and other expressions. By "Boolean expression" we mean an expression, which always results in a Boolean value in all dynamic contexts

As J-Bob's logical statements are just Boolean expressions, its proofs are actually program transformations whose goal is to transform the expression representing a given theorem statement into the constantly true expression (`'t`). Namely, a J-Bob proof is a sequence of steps, each step being one of three kinds:

- rewriting – possibly conditional – based on some equality known to be true (either an axiom or a previously proved theorem);
- unfolding or folding of a defined function;
- evaluation of a built-in function called with constant arguments.

Each step is a pair consisting of:

- a path defining a subexpression;
- a transformation to perform on this subexpression.

Paths are just sequences of navigation steps inside compound expressions: the single-letter atoms `Q`(uestion), `A`(nswer), and `E`(lse) are used to select one of the 3 subexpressions of an if-expression and natural numbers (starting from 1) index the arguments of a function call.

As an example we can consider the proof that the list append function is associative:

```
((dethm append-assoc (xs ys zs)
    (equal (append (append xs ys) zs) (append xs (append ys zs))))
    (list-induction xs)
    ((A 1 1) (append xs ys))
    ((A 1 1) (if-nest-A (atom xs) ys (cons (car xs) (append (cdr xs) ys)
        )))
    ((A 2) (append xs (append ys zs)))
    ((A 2) (if-nest-A (atom xs) (append ys zs) (cons (car xs) (append (
        cdr xs) (append ys zs)))))
    ((A) (equal-same (append ys zs)))
    ...
    (() (if-same (atom xs) 't))
)
```

The proof starts with an indication that we are to proceed by induction on argument `xs`. At that point (if we have not added other steps to the proof yet), J-Bob presents us with a proof obligation based on the body of the theorem and the selected induction scheme:

```
(if (atom xs)
  (equal (append (append xs ys) zs) (append xs (append ys zs)))
  (if (equal (append (append (cdr xs) ys) zs) (append (cdr xs) (append
      ys zs)))
    (equal (append (append xs ys) zs) (append xs (append ys zs))) 't))
```

In the base case when `xs` is an atom, we must prove the statement directly, otherwise we can use the induction hypothesis `(equal (append (append (cdr xs)ys)zs) (append (cdr xs)(append ys zs)))` inside the proof. The first step of the actual proof – `((A 1 1) (append xs ys))` – unfolds the corresponding occurrence of `append` and the current expression becomes:

```
(if (atom xs)
  (equal (append (if (atom xs) ys (cons (car xs) (append (cdr xs) ys)))
      zs) (append xs (append ys zs)))
  (if (equal (append (append (cdr xs) ys) zs) (append (cdr xs) (append
      ys zs)))
    (equal (append (append xs ys) zs) (append xs (append ys zs))) 't))
```

We notice the nested occurrence of a check for `(atom xs)`, which is redundant. We eliminate it with the next proof step – ((A 1 1) (if-nest-A (**atom** xs)ys (**cons** (**car** xs)(append (**cdr** xs)ys)))) – where `if-nest-A` is an axiom from the J-Bob standard library. The resulting expression is:

```
(if (atom xs)
  (equal (append ys zs) (append xs (append ys zs)))
  (if (equal (append (append (cdr xs) ys) zs) (append (cdr xs) (append
       ys zs)))
    (equal (append (append xs ys) zs) (append xs (append ys zs))) 't))
```

With 2 analogous proof steps we can also simplify the second call (append xs ...) and we get:

```
(if (atom xs) (equal (append ys zs) (append ys zs))
  (if (equal (append (append (cdr xs) ys) zs) (append (cdr xs) (append
       ys zs)))
    (equal (append (append xs ys) zs) (append xs (append ys zs))) 't))
```

There is a call to `equal` with identical arguments, which we can simplify with another library axiom in the next step – ((A) (equal-same (append ys zs))):

```
(if (atom xs) 't
  (if (equal (append (append (cdr xs) ys) zs) (append (cdr xs) (append
       ys zs)))
    (equal (append (append xs ys) zs) (append xs (append ys zs))) 't))
```

At this point the answer-arm of the outermost if-expression (which corresponds to the base case of the induction) has been reduced to `'t`. After 12 more proof steps (not shown for brevity), we also reduce the else-arm to `'t`:

```
(if (atom xs) 't 't)
```

The last step of the proof – ((() (if-same (**atom** xs)'t)) – reduces this last expression to `'t`, which completes the proof. We have shown – by using an induction scheme plus a sequence of elementary program transformations – that the statement of the theorem will always evaluate to `'t`, no matter what arguments we pass.

## 3 A Supercompiler for J-Bob

### 3.1 Overview

The example from the previous section shows that many steps in a typical J-Bob proof are "obvious" reductions of subexpressions. Such proof steps are very tedious to write by hand, especially as J-Bob – being a very minimalistic prover – insists on fully specifying all arguments of the axiom or theorem each step uses. On the other hand many of these steps coincide directly with the steps a typical supercompiler would take when presented with such an expression as input. So our first goal is to make a supercompiler automate – as much as possible – the tedious parts of a J-Bob proof. To support this goal, our supercompiler must return not only the resulting expression, but also the sequence of transformation steps used to convert the input to the output expression, similar to the certifying supercompiler of Klyuchnikov et al. [14]:

$$scp : Prg \times Exp \rightarrow Exp \times Steps$$

If the input expression is the current goal of an unfinished J-Bob proof, we can insert the returned steps inside the proof and change the goal to the final expression produced by the supercompiler – so such an interface to the supercompiler ensures that it can be used directly for J-Bob proof automation. The sequence of transformation steps is actually sufficient to reconstruct the final expression from the original one. Supposing a function for performing transformation steps (which already exists in J-Bob)

$$rewrite : Prg \times Exp \times Steps \to Exp$$

it should hold that:

$$scp(defs, e_1) = (e_2, steps) \to rewrite(defs, e_1, steps) = e_2.$$

This observation permits us to simplify the type of the supercompiler by returning only the necessary transformation steps:

$$scp' : Prg \times Exp \to Steps$$

This approach makes easier the construction of the supercompiler, but also – and more importantly – its correctness proof.

Our second goal is to see if, and to what extent, the proof automation provided by our supercompiler can help in its own correctness proof. Several observations follow from this goal:

- we must formalize the correctness proof inside J-Bob;
- the supercompiler itself must be written in the Lisp dialect that J-Bob can handle;
- the precise formal statement of what it means for the supercompiler to be correct must be compatible with the semantics of J-Bob.

While the first 2 points are trivial, the last one requires some elaboration. J-Bob tacitly assumes that Lisp programs are evaluated in accordance with some semantics, but this semantics is not fully and explicitly described. There are Lisp expressions, about which J-Bob cannot reason. For example, there is no rule in the standard library, which could tell the value of `(if (cons x y) a b)`, (unless both `x` and `y` are constant values), although it is expected that this value is always defined. We sidestep this lack of explicit program semantics by changing what we mean by supercompiler correctness: we simply require that the supercompiler return a valid sequence of transformation steps, which will not get stuck if applied to the original expression. To formalize this definition in a simple and explicit way, we need a modified version of the rewriting function, which also returns an explicit flag if all the steps have been performed successfully:

$$rewrite' : Prg \times Exp \times Steps \to Bool \times Exp$$

We can then define:

$$correct(scp') \equiv \forall defs \forall e(fst(rewrite'(defs, e, scp'(defs, e))) = true)$$

If we assume – or have a separate proof – that J-Bob transformation steps respect the underlying semantics of the language, then the above definition is equivalent to the traditional definition of correctness as semantics preservation.

To keep the supercompiler simple – both as implementation and as correctness proof, we reuse an architecture we have already applied in another simple supercompiler [15]. The implementation is split into layers:

- simplification by basic supercompilation transformations. This layer performs reductions on open expressions, such as if-expressions with a constant condition, or calls to built-in functions with all arguments being constant. This layer also performs if-lifting (to be defined precisely below) and information propagation. No unfolding or folding happens in this layer;
- unfolding layer, using different strategies.

The overall organization of the supercompiler is to perform a sequence of unfolding steps and a subsequence of basic transformations before and after each unfolding:

$$scp = (simplify) * \cdot (unfold \cdot (simplify) *) *$$

The following subsections explain in detail the transformations applied by each layer. They also explain what happens with the other typical supercompiler ingredients – *folding*, *generalization*, and *whistle* – which do not appear explicitly in the architecture as outlined above.

Before we continue, we show how a proof for the same property (list append associativity) can look like:

```
(dethm append-assoc (xs ys zs)
    (equal (append (append xs ys) zs) (append xs (append ys zs))))
    (list-induction xs)
    (scp 5 50)
    (expand ((append 5)) (equal-if (append (append (cdr xs) ys) zs) (
        append (cdr xs) (append ys zs))))
    (scp 0 50)
)
```

The proof has only 3 steps (versus 18 in the manual proof) – 2 calls to the supercompiler (with different options) and 1 manual step. These 3 macro-steps are expanded to 18 steps before being submitted to J-Bob to check – incidentally the same 18 steps as in the manual proof, but in a slightly different order.

### 3.2 Basic Supercompilation Transformations

The proof example from the previous section has already hinted that many of the elementary program transformations J-Bob performs are similar to those performed by supercompilation. A detailed analysis of all available transformations (mostly in the form of axioms in the standard library) confirms, that we have all the ingredients to simulate supercompiler actions by J-Bob proof steps. Table 1 lists a subset of the axioms suitable for our needs. The name and the definition are given exactly as found in the J-Bob standard library, while the remaining 2 columns illustrate the action of each rule on a typical expression.

We can roughly divide these transformation rules in 3 groups:

**Table 1.** Transformation rules

| Name | Definition | Original expression | Result expression |
|---|---|---|---|
| **atom/cons** | (**equal** (**atom** (**cons** x y))'nil) | (**atom** (**cons** x y)) | 'nil |
| **car/cons** | (**equal** (**car** (**cons** x y))x) | (**car** (**cons** x y)) | x |
| **cdr/cons** | (**equal** (**cdr** (**cons** x y))y) | (**cdr** (**cons** x y)) | y |
| if–true | (**equal** (**if** 't x y) x) | (**if** 't x y) | x |
| if–false | (**equal** (**if** 'nil x y) y) | (**if** 'nil x y) | y |
| if–nest–A | (**if** x (**equal** (**if** x y z) y) 't) | (**if** x<br>(... (**if** x y z)<br>...)<br>...<br>) | (**if** x<br>(... y ...)<br>...<br>) |
| if–nest–E | (**if** x 't (**equal** (**if** x y z) z)) | (**if** x<br>...<br>(... (**if** x y z)<br>...)<br>) | (**if** x<br>...<br>(... z ...)<br>) |
| if–same | (**equal** (**if** x y y) y) | (**if** x y y) | y |
| equal–same | (**equal** (**equal** x x)'t) | (**equal** x x) | 't |

– rules for performing evaluation of open expressions (`atom/cons`, `car/cons`, `cdr/cons`, `if-true`, `if-false`). They correspond to a form of simple partial evaluation.
– rules for nested repeated conditions (`if-nest-A`, `if-nest-E`). Their definition in J-Bob is less obvious, but it simply corresponds to the way J-Bob encodes conditional rewriting rules – as equalities inside if-expressions. These rules correspond to a form of information propagation (both positive and negative) as found in supercompilation.
– rules dealing with duplicated arguments in `equal` and `if` (`if-same`, `equal-same`). With a few exceptions, such transformations are usually not employed by supercompilers, but it is easy to include them in our supercompiler.

If we compare these rules to the basic transformations used in other simple supercompilers [15], there is one ingredient missing: the ability to lift if-expression appearing as conditions of other if-expressions. It turns out, however, that such *if-lifting* can be performed with a combination of existing rules by using the fact, that J-Bob rewriting rules can be applied in both directions:

```
    (if (if q a e) a' e')
= (if q (if (if q a e) a' e') (if (if q a e) a' e'))   {if-same, right-to-left}
= (if q (if a a' e') (if (if q a e) a' e'))            {if-nest-A, left-to-right}
= (if q (if a a' e') (if e a' e'))                     {if-nest-E, left-to-right}
```

Actually, we can use the same trick to lift if-expressions outside of function calls. This is not performed by most supercompilers, because it is not safe in general (it can make a program less terminating). The unsafe step in our derivation above would be the use of `if-same` right-to-left. But J-Bob ensures totality of all expressions, so general if-lifting is a valid transformation in our case.

### 3.3 Unfolding and Generalization

Unfolding is possible as a basic transformation step in J-Bob, so we are free to add it to our supercompiler. The only question is when and in what order – since J-Bob requires totality proofs for all functions, we are not tied to any particular evaluation order. We can consider the decision whether to unfold a given call or not as a form of generalization. That is why we consider unfolding and generalization together. Experiments with early versions of the supercompiler for performing proofs have shown, that it is useful to have several strategies for unfolding:

- *best unfolding*: We tentatively perform each of the possible unfoldings, simplify the resulting term with the basic transformations from the previous subsection, and select the one resulting in the smallest final expression.
- *call-by-name*: We select the first possible unfolding in leftmost-outermost order.
- *call-by-value*: We select the first possible unfolding in leftmost-innermost order.

During each (extended) proof step, the proof author can decide which strategy is most appropriate, and select it by a switch.

### 3.4 Folding

As already mentioned, our supercompiler does not perform folding (yet). There are several reasons for this decision:

- Folding seems much less useful for a supercompiler aimed at producing proofs, than in a supercompiler tailored for program optimization or analysis. The reason is that the goal of a J-Bob proof is to reduce a given expression to `'t`. If we introduce new function definitions by folding, it means we have given up any chance to arrive to a final expression equal to `'t`.
- It is not obvious how to integrate arbitrary folding inside a J-Bob proof. J-Bob supports folding steps, but only for existing function definitions. It is not possible to introduce new function definitions on-the-fly inside a proof. Besides, even if there were a way, we would still have to produce a proof of termination for each newly introduced function definition, which is in general a non-trivial task.
- The lack of folding simplifies a lot the correctness proof of the supercompiler. As we describe just a proof-of-concept experiment, whose main goal is to have a supercompiler correctness proof performed with the help of the supercompiler itself, this simplification appears a worthy compromise.

Of course, if we want to use this supercompiler for other tasks beside proof automation, adding support for folding will be highly desirable. We leave this task for future work.

### 3.5 Whistle

Whistles are critical for the performance of supercompilers aimed at program optimization and analysis. They must not stop the transformation process too early, as opportunities for optimization might be lost. But they should not stop too late either – the supercompiler should not spend too much time producing a bloated result with a lot of duplication, or even fail to stop at all. In our case, however, the supercompiler can be called many times inside a single proof, with different goals in mind. So it is more important to provide better control to the user in the selection of a suitable transformation strategy in each case, than to rely on a sophisticated general whistle. To keep our implementation – and its proof – simple, we have currently settled for a basic whistle using 2 counters: one limiting the total number of unfoldings to perform and one limiting the number of basic transformation steps performed between 2 consecutive unfoldings.

## 4 Implementation Details

### 4.1 Coq Prototype

Before the actual implementation of the supercompiler in J-Bob Lisp was started, we implemented a prototype in Coq to study different approaches to the implementation and their impact on the correctness proof. We used Coq, because creating proofs of such scale and complexity by hand in J-Bob appeared so lengthy and tedious as to be completely impractical. Currently the Coq prototype contains implementations of the main supercompiler components – basic transformations and unfolding – as well as a re-implementation of the rewriting component of J-Bob itself. These implementations match very closely the corresponding code in the J-Bob version. The correctness proofs of the implemented supercompiler components are almost complete, and demonstrate the feasibility of formally verifying such a proof in full.

### 4.2 Implementation in J-Bob

J-Bob is distributed in 2 parallel versions: one that can run inside ACL2 and one that can run inside any Scheme implementation. The J-Bob sources themselves are almost identical in the 2 versions, but there are different thin wrappers, which emulate J-Bob Lisp on top of ACL2 and Scheme respectively. We have chosen to use the Scheme version and our implementation is in Scheme[2], although most parts of the code are in the restricted Lisp subset supported by J-Bob. The files[3] containing the different parts of the source code are briefly described in Table 2.

The source files contain a fair amount of deliberate code duplication, because we need to use many pieces of code in two different ways. We must execute

[2] The code was tested with Racket 6.4, in R5RS emulation mode, with redefinition of initial bindings allowed.

[3] `https://bitbucket.org/dkrustev/jbobscp`

**Table 2.** Source code files

| File | Description |
|---|---|
| `Coq/JBobScp.v` | Coq prototype |
| `j-bob/*` | A copy of J-Bob sources, as a git submodule |
| `Scheme/j-bob-rewriter.scm` | Copies of some definitions of J-Bob itself, which are only needed for the supercompiler correctness proofs |
| `Scheme/j-bob-rewriter2.scm` | patched versions of some definitions of J-Bob. The main goal of the changes is to add an explicit return flag if a transformation step or a sequence of steps was successfully applied. All patched versions have the same names as the original definitions, with an added suffix "2" |
| `Scheme/j-bob-scp.scm` | Implementation of the supercompiler for J-Bob List programs |
| `Scheme/j-bob-expand-proofs.scm` | A "proof expander": taking a list of proofs with steps using an extended syntax, and expanding them to simple steps directly accepted by J-Bob |
| `Scheme/j-bob-scp-proofs.scm` | Proof of correctness for the supercompiler, using proof automation supplied by the supercompiler itself. |

those parts directly (J-Bob itself, the supercompiler). We also need to reason about the same code inside J-Bob, which, as a minimum, requires to wrap each function definition with a termination proof and to package all such definitions in an environment, which can be passed to J-Bob at runtime. Simple text file comparison can convince us that the parallel versions of duplicated definitions are identical. For example, we can compare `j-bob-rewriter.scm` (used only in the proofs) with the original source of J-Bob. Similarly, we can compare `j-bob-scp.scm` (which is executed) to `j-bob-scp-proofs.scm` (which contains (a part of) the same definitions inside the proof environment).

**Supercompiler Implementation** The main functions of the supercompiler implementation are as follows:

- `(simplify-current fullscp eroot path e)` tries to find a sequence of suitable basic transformation steps (Sect. 3.2) for the current subexpression `e`, which is at position `path` inside the top-level expression `eroot`. The boolean flag `fullscp` indicates whether we want full supercompilation or just a simple form of partial evaluation.
- `(simplify-top fullscp e)` returns (if possible) a basic transformation sequence for a *single* subexpression of the top-level expression `e`.
- `(simplify* defs fullscp fuel e)` returns a sequence of basic transformation steps, which simplify up to `(length fuel)` subexpressions of the top-level expression `e`. `defs` is the list of current definitions.

- (unfold−steps−top defs e) returns a list of all possible unfolding steps inside the expression e.
- (choose−unfold−step defs fullscp whitelist blacklist simplfuel e) returns a single unfolding step according to the specified strategy:
  - if whitelist is not empty, the first unfolding (in outermost leftmost order) for a function in the white-list is returned;
  - if blacklist is not empty, the first unfolding (in innermost leftmost order) for a function *not* in the black-list is returned;
  - if both lists are empty, we select the unfolding step, which results in the smallest new expression (after simplification using simplify∗).
- (scp−steps defs fullscp unfoldfuel whitelist blacklist simplfuel e) is the top-level supercompiler function. It returns a sequence of transformation steps for the expression e, which contains up to (length unfoldfuel) unfolding steps, around each of which we can have up to (length simplfuel) basic simplification steps.

As we can deduce from these descriptions, the implementation follows closely the architecture outlined in Sect. 3.1.

**Proof Expander** On top of the supercompiler implementation we have built a preprocessor, which takes proofs with a richer set of possible steps, and expands them into a sequence of standard proof steps, which J-Bob can verify. With this organization the proofs produced by the supercompiler (or by other extended tactics) are always checked by J-Bob, therefore we do not need to trust them. Some of the new proof steps include:

- `(scp` [<*unfolding limit*> [<*simplification limit*> [<*unfolding white-list*> [<*unfolding black-list*>]]]]`)`. This is the step, which calls the supercompiler on the current goal, and pastes its result at the current point of the proof. There is also a variant starting with the keyword `simpl`, which performs a reduced set of basic transformations, roughly equivalent to simple partial evaluation. It is useful, for example, when we need to simplify a call where most of the arguments are constants, as it produces a shorter sequence of steps.
- `(expand (`<*extended path*>`)` <*transformation*>`)`. This step always corresponds to a single standard J-Bob step, but it permits an extended syntax for paths, which is easier to use: $(path_1\ (f\ n)\ path_2)$ corresponds to $(path_1\ path_3\ path_2)$, where $path_3$ is the path to the $n$-th occurrence of a call to $f$ in the subexpression found at $path_1$.

Readers interested in examining the source code of the implementation may find it, in places, unnecessarily convoluted. Such complicated tricks are, however, necessary to overcome limitations of the J-Bob Lisp dialect: no higher-order functions, no mutual recursion, no let-expressions. While the first limitation was not felt heavily during the development of the supercompiler (which is, after all, just a few hundred lines), the combination of the last two restrictions proved to be a major hurdle.

**Supercompiler Correctness Proof** All the main supercompiler functions listed above, as well as many of the auxiliary definition they use, return a list of transformation steps. As a result, the structure of the correctness proof is quite simple, and follows the structure of the implementation itself: for each such function definition we have a lemma, stating that the returned list of steps – in a suitable context – can be successfully executed by J-Bob. As an example, here is the proof for the function simplify-current:

```
((dethm simplify-current-correct (fullscp eroot path e)
    (if (focus-is-at-path?2 path eroot)
      (if (equal (find-focus-at-path2 path eroot) e)
        (equal (car (rewrite/steps2 (axioms) eroot (simplify-current
            fullscp eroot path e))) 't)
        't)
      't))
    nil
    (scp 1 50 (simplify-current))
    (insert-Q (A A A) (equal (find-focus-at-path2 path eroot) (if-c (if.
        Q e) (if.A e) (if.E e))))
    ((A A A A 1) (simplify-if-correct fullscp eroot path (if.Q e) (if.A
        e) (if.E e)))
    ((A A A Q 2) (if?/if-c/if.Q/if.A/if.E e))
    (scp 0 50)
    ((A A E A 1) (simplify-app-correct eroot path e))
    (expand ((rewrite/steps2 1)) (rewrite/steps2 (axioms) eroot '()))
    (scp 0 50))
```

It contains appeals to some lemmas about auxiliary functions (simplify-if-correct, simplify-app-correct) and some other manual steps, interspersed with calls to the supercompiler to fill in the tedious parts of the proof.

There are a few important design decisions, which substantially simplified the formal proofs of supercompiler correctness. As the definition of correctness (Sect. 3.1) uses the J-Bob rewriting machinery as a reference point, we use the same machinery as much as possible in the implementation of the supercompiler as well. For example, we could implement positive/negative information propagation by keeping track of the set of conditions we know to be true/false, as we descend recursively inside subexpressions. The J-Bob rewriter uses, however, a different approach (likely more adapted to its own architecture). There are functions (prem-A?/prem-E? prem path e), which check if condition prem occurs positively/negatively somewhere on the given path inside the top-level expression e. So we chose to use the same functions for information propagation inside the supercompiler, which is the main reason to carry around the top-level expression eroot in most supercompiler functions. Another decision, explicitly aimed at simplifying the correctness proof, was to use a small-step-style implementation not only for the unfolding steps, but for the basic transformation steps as well. The reason can be explained with the following example. If we simplify the expression $(f\ e_1\ e_2)$ using a big-step style, we first compute recursively the simplification steps for $e_1$ and $e_2$ (say $steps_1$ and $steps_2$), and the result for the whole expression will be (append $steps_1$ $steps_2$). What is important is that we compute $steps_2$ in the context of the original expression, but the rewriting engine will have to apply them on the result of applying $steps_1$ to this initial expression. This mismatch prevents a simple inductive argument, because we cannot use directly the inductive hypothesis for $e_2$. In order to make such a proof feasible,

we would have to first formalize that the transformation steps for $e_1$ and $e_2$ are independent, as they treat disjoint subexpressions, which would complicate and lengthen the proof considerably.

The proof of the full J-Bob supercompiler is far from complete – mostly for reasons we discuss in the next section. Existing proofs cover many of the basic transformation steps, however, and clearly demonstrate the importance of supercompiler proof automation to make them feasible. Completing the formal proofs should simply be a matter of investing more time and solving some problems unrelated to the use of supercompilation in verification. The Coq prototype shows there are no important technical difficulties in the formal proofs themselves.

## 5 Performance Evaluation

Table 3 contains some statistics about the currently existing lemmas in the supercompiler correctness proof[4]. The proofs are classified – subjectively – in 3 categories:

1. "typical" proofs, which rely mostly on logical reasoning (analysis by cases, appeals to existing lemmas, rewriting, ...);
2. proofs by direct computation (which requires, however, many J-Bob standard steps);
3. a mixture of the above 2 categories – proofs that for the most part are like those in the first category, but also contain steps using computation over known values.

The statistics presented in the table support this classification – the values in the last two columns are very similar within the categories 1 and 2, but quite distinct between the two categories. Category 3 has more diverse values, but on average they are between those for category 1 and category 2.

The good news first: Even if we completely ignore the statistics of categories 2 and 3, we can conclude that supercompilation is of great help as a form of proof automation: category 1 has almost an order of magnitude of savings in the number of proof steps one has to enter manually (7.14 expanded steps per single original step). If we include all categories, the savings are even more impressive – almost two orders of magnitude.

The bad news is that the current implementation is too slow to be used in an interactive fashion. The expansion of all existing proofs (which are only a part of the full supercompiler correctness proof) takes almost 40 sec in this experiment; with the time J-Bob requires to verify the expanded proofs, the full time is almost 75 sec. As J-Bob reevaluates all existing proofs after each user modification of the current proof, working on the supercompiler correctness proof requires waiting for over a minute between each 2 interactive proof changes.

[4] Tests performed on a laptop with a Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz, 8 GB RAM, OS Microsoft Windows 7 Pro 64 bit, using DrRacket 6.4 with debug info switched off.

**Table 3.** Correctness proof statistics

| (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) | (10) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | path-b-exp-induction | 3 | 2 | 6 | 110 | 120 | 0 | 66.67% | 2.00 |
| 1 | focus-is-at-path/rewrite-focus-at-path | 18 | 11 | 143 | 5475 | 5530 | 611 | 61.11% | 7.94 |
| 1 | focus-is-at-path/rewrite-focus-at-path/cons-true | 6 | 1 | 28 | 187 | 190 | 31 | 16.67% | 4.67 |
| 1 | simplify-if-correct | 14 | 7 | 87 | 3588 | 3590 | 764 | 50.00% | 6.21 |
| 1 | if?/if-c/if.q/if.a/if.e | 8 | 2 | 58 | 250 | 260 | 0 | 25.00% | 7.25 |
| 1 | app?-cons/car+cdr | 3 | 2 | 19 | 156 | 150 | 16 | 66.67% | 6.33 |
| 1 | simplify-app-quoted-correct | 14 | 9 | 188 | 577 | 590 | 16 | 64.29% | 13.43 |
| 1 | list2?-expand-list | 5 | 2 | 18 | 156 | 150 | 0 | 40.00% | 3.60 |
| 1 | simplify-app-not-quoted-correct | 12 | 6 | 92 | 2668 | 2670 | 80 | 50.00% | 7.67 |
| 1 | simplify-app-correct | 7 | 4 | 30 | 265 | 270 | 16 | 57.14% | 4.29 |
| 1 | simplify-current-correct | 8 | 3 | 37 | 359 | 350 | 32 | 37.50% | 4.63 |
| 2 | lookup/if-true | 1 | 1 | 324 | 422 | 430 | 32 | 100.00% | 324.00 |
| 2 | lookup/if-false | 1 | 1 | 370 | 421 | 430 | 32 | 100.00% | 370.00 |
| 2 | lookup/if-nest-a | 1 | 1 | 462 | 593 | 590 | 78 | 100.00% | 462.00 |
| 2 | lookup/if-nest-e | 1 | 1 | 416 | 577 | 580 | 64 | 100.00% | 416.00 |
| 2 | lookup/atom/cons | 1 | 1 | 48 | 172 | 180 | 47 | 100.00% | 48.00 |
| 2 | lookup/equal-same | 1 | 1 | 186 | 296 | 290 | 15 | 100.00% | 186.00 |
| 3 | simplify-if-correct-if-true | 16 | 12 | 1126 | 2840 | 2880 | 330 | 75.00% | 70.38 |
| 3 | simplify-if-correct-if-false | 16 | 12 | 1135 | 3026 | 3030 | 392 | 75.00% | 70.94 |
| 3 | simplify-if-correct-if-nest-a | 14 | 11 | 1722 | 5257 | 5380 | 623 | 78.57% | 123.00 |
| 3 | simplify-if-correct-if-nest-e | 14 | 11 | 1003 | 3932 | 3950 | 376 | 78.57% | 71.64 |
| 3 | simplify-atom-correct | 25 | 11 | 1180 | 5397 | 5410 | 95 | 44.00% | 47.20 |
| 3 | simplify-equal-correct | 29 | 16 | 589 | 2434 | 2430 | 47 | 55.17% | 20.31 |
| 1 | Total by category | 99 | 49 | 707 | 13916 | 13990 | 1566 | 49.49% | 7.14 |
| 2 | Total by category | 6 | 6 | 1806 | 2481 | 2500 | 268 | 100.00% | 301.00 |
| 3 | Total by category | 114 | 73 | 6755 | 22886 | 23080 | 1863 | 64.04% | 59.25 |
| | Total | 219 | 128 | 9268 | 39283 | 39570 | 3697 | 58.45% | 42.32 |
| | Total time with proof checking | | | | 73726 | 74140 | 4335 | | |

*Column* *Description*

(1) Lemma category
(2) Lemma name
(3) Total number of steps in original proof
(4) Number of supercompilation steps in original proof
(5) Total number of steps after proof expansion
(6)-(8) CPU/real/GC time (msec) for proof expansion (including supercompilation), as reported by Racket's `time` function
(9) Frequency of supercompilation steps in original proof
(10) Ratio of expanded to original proof steps

Initially the performance of the system was even worse, but we managed to improve it substantially by adding some new options to the supercompiler: a call-by-value unfolding strategy was added beside the existing ones ("best" unfolding and call-by-name); a flag was added to perform only a reduced set of basic transformation steps, corresponding to a form of simple partial evaluation. Table 4 demonstrates the effect of these options on the most time-consuming proofs in category 2. These proofs contain only a single call to the supercompiler, and they can be completed by using any combination of options, which makes them suitable for this comparison. The table shows – as expected – that CBV outperforms a lot CBN and that simple partial evaluation is a little bit better than full supercompilation. Of course, such comparisons are meaningful only when all options lead to the same result in the corresponding proof.

**Table 4.** Evaluation strategy statistics

| | Full supercompilation | | | | Simple partial evaluation | | | |
|---|---|---|---|---|---|---|---|---|
| | CBN | | CBV | | CBN | | CBV | |
| | (1) | (2) | (1) | (2) | (1) | (2) | (1) | (2) |
| lookup/if-true | 1038 | 4960 | 498 | 592 | 716 | 2840 | 324 | 422 |
| lookup/if-false | 1226 | 6209 | 569 | 624 | 858 | 3478 | 370 | 421 |
| lookup/if-nest-a | 1632 | 8830 | 711 | 749 | 1172 | 4977 | 462 | 593 |
| lookup/if-nest-e | 1424 | 6989 | 640 | 717 | 1010 | 4134 | 416 | 577 |
| lookup/atom/cons | 120 | 390 | 72 | 203 | 74 | 327 | 48 | 172 |
| lookup/equal-same | 534 | 2028 | 285 | 406 | 350 | 1295 | 186 | 296 |

| *Column* | *Description* |
|---|---|
| (1) | Total number of steps after proof expansion |
| (2) | CPU time (msec) |

Even with these improvements we still need one "quick-and-dirty" trick to get around the performance problem. Proofs are split into smaller pieces (in the form of auxiliary lemmas). Especially those parts of a proof, which involve direct computation only, are extracted as separate lemmas whenever it is not too complicated. (This is the real reason for the existence of all lemmas in category 2.) After the proof of each such lemma is ready and checked, we shunt it by replacing it with a fake proof, which consists of only a single J-Bob standard step (using a fake axiom). Such shunting is especially useful for the lemmas in categories 2 and 3. When it is put in place, the total time for proof expansion and checking goes down to a little over 10 sec on the same machine, which is already (barely) acceptable for interactive proof editing. Still, the rechecking time between proof modifications will go up as we continue to make progress towards a full proof of supercompiler correctness (comparable to that in the Coq prototype). Given the current performance of the system, we decided to postpone the work towards completing the proof until we can achieve more improvements in its reactivity.

If we want better improvements in performance, we must first analyze the causes for the current long processing times. They seem mostly related to the underlying proof assistant:

- J-Bob does not have its own interactive editor or shell. It just provides a simple high-level API, which can be used directly from the Scheme REPL. As this API is stateless, it entails full rechecking of all current proofs after each user interaction.
- J-Bob only allows very elementary program transformations as proofs steps. Allowing even a simple form of partial evaluation as a built-in proof step (similar to what proof assistants like Coq and Agda provide) would eliminate a big source of inefficiency in the proofs listed in Table 3.
- The restrictions of J-Bob's Lisp subset – especially the lack of let-expressions – often make it too hard to write an efficient version of the algorithm one has in mind. Instances of this problem exist inside the sources of both J-Bob itself and our supercompiler: sometimes they perform multiple traversals or repeat some computations just because of the lack of let-expressions.

Of course, all these limitations stem from the goal of J-Bob: to be a minimalistic proof assistant used mostly for educational purposes. Solving some of these limitations – such as the introduction of a built-in partial evaluation step – would require modifying J-Bob itself, and making it bigger, more complex, and potentially less reliable. Some other limitations can probably be removed without touching the J-Bob core. Adding let-expressions can likely be done by a preprocessor. A dedicated J-Bob REPL (or even just a statefull API for the Scheme REPL) would avoid the need to recheck all proofs after each interaction. We leave the study of these possibilities for future work.

## 6 Related Work

Proof automation is a large and active research area, covering a broad range of methods. The bibliography of one recent book [7] has about 700 references. We shall therefore not attempt a thorough comparison of the current method to other existing methods for proof automation, limiting ourselves instead to just a few works we consider most relevant.

J-Bob is closely related to ACL2 [10] and Milawa [1], as they all follow the traditions of the early Boyer-Moore prover, Nqthm. But because of their different intended usage, these provers have important differences. While J-Bob is a minimalistic educational tool, with no proof automation at all, both ACL2 and Milawa have facilities for proof automation. ACL2 is an industrial-strength theorem prover with powerful methods for automatic proof search. Its architecture is monolithic, without a dedicated core, and bugs anywhere in the system can impact its soundness as a prover [1]. Milawa is another prover in the Nqthm family, which proposes an interesting solution to the soundness and trusted-core problems. Its minimal core proof checker has to be trusted, while a reflection mechanism allows a new proof checker to be installed, if its soundness can be

verified by the current checker. By repeatedly installing new proof checkers, which accept higher-level proof steps, the level of Milawa can be raised to one approaching in power ACL2 [1]. In our approach, we leave the core proof checker (J-Bob) untouched, and instead expand higher-level proof steps into sequences of steps it can check. Such proof expansion can lead to high processing-time requirements in larger proofs. On the other hand, Milawa requires trusting not only its core proof checker (which appears even simpler than J-Bob), but also its reflection mechanism. We leave a more detailed comparison of the two approaches for future work.

The idea to apply supercompilation for proof automation appears already in some of Turchin's early papers [23]. Different specific applications of verification by supercompilation have been studied [11, 13, 18, 19]. In all these cases we have to rely on the correctness of the used supercompiler, or have it proven correct, in order to trust the results of verification. There is even a theorem prover based on distillation (a program transformation method closely related to supercompilation) – Poitín [6]. Again, it appears that distillation is closely integrated into the kernel of this prover, so that bugs in its implementation may impact the soundness of the proved results.

Klyuchnikov et al. [14] propose an elegant solution to avoid the necessity to trust that the supercompiler is bug-free. They introduce a *certifying* supercompiler, which produces – together with the resulting transformed program – a proof that it is equivalent to the input program. This proof may be verified by an independent proof checker (hopefully much simpler than the supercompiler, and so with lower probability of soundness-critical bugs). We use the same idea, with a shortcut: our supercompiler produces only a proof, and the resulting program can be recovered from this proof by an independent process. Another important difference is that the supercompiler of Klyuchnikov et al. is implemented in a language (Scala) very different from the one it can treat (a version of Martin-Löf type theory). So their supercompiler cannot be used directly for its own verification.

Self-application has long been a desirable – but also somewhat elusive – goal in the context of supercompilation and partial evaluation in general. This interest stems mostly from the possibility to apply the Futamura projections [4], which enable the production of compilers from interpreters, and of compiler generators. Such optimizing self-application has been demonstrated first with partial evaluation [9], and then extended to cover online partial evaluation [5]. It appears harder to achieve in the context of supercompilation – there is a single description of successful experiments of self-application with a version of the Refal supercompiler [20]. The supercompiler we describe is self-applicable – in a sense that it can process programs in the same language it is written in. It cannot hope to achieve Futamura-projection-like self-application, mostly because it currently lacks folding. As we have demonstrated, it is still powerful enough to be used in proofs reasoning about its own sources.

The formal verification of supercompilers has recently emerged as an interesting research topic. In earlier work [15] we have demonstrated – on a simple

supercompiler for a tiny imperative language – the feasibility of this task. The current work reuses some ideas of that previous verification effort, most notably the decomposition of the supercompilation process in several phases. Subsequent research on formal correctness proofs for supercompilers has mostly concentrated on providing general frameworks, which can simplify the verification of many different supercompilers [16, 17, 21]. In all these cases a general proof assistant is used (Coq, Agda), and no attempt is made to use a supercompiler as a proof automation tool for its own verification.

## 7 Conclusions and Future Work

We have described the design of a certifying supercompiler, which can work together with a proof assistant (J-Bob) and supply automatically generated proof fragments upon request by the proof assistant user. The supercompiler is also self-applicable, as it is written in the same first-order subset of Lisp, which it can process. This feature cannot currently be used for producing Futamura projections, as the system does not implement folding yet. Self-application, however, permits the supercompiler to supply proof automation for its own correctness proof. To the best of our knowledge, this is the first successful experiment, where a supercompiler can assist its own formal verification. We have quantified the amount of proof automation the supercompiler provides by measuring the ratio of high-level proof steps (relying on supercompilation) versus low-level proof steps that the proof checker can verify directly. This ratio shows an almost two-orders-of-magnitude improvement, when calculated on the ready part of the supercompiler correctness proof. We estimate that such improvement is sufficient as proof-of-concept for the applicability of the proposed approach.

An interesting feature of our approach is that the user is not forced to use the supercompiler in a one-shot, all-or-nothing fashion on a given problem. Instead, the user builds formal proofs in interaction with a proof assistant, and at each step of the proof she may try to call the supercompiler for help. It would be interesting to study if such an incremental approach can work in other domains (like program analysis).

To make the implemented system really practical for users of J-Bob, we need to solve the performance problems we have detected while working on the supercompiler correctness proof. Our analysis indicates most of these performance issues are ultimately related to limitations of J-Bob itself. We have outlined some possible solutions, which we may try in the future.

Another interesting possibility is to apply the same approach to different proof assistants, featuring different programming languages. As Klyuchnikov et al. [14] have demonstrated, it is possible to build a certifying supercompiler for a language with higher-order functions and dependent types, such as those found in proof assistants like Coq and Agda. The challenge will be to produce a similar supercompiler, which is self-applicable and integrated with the corresponding proof assistant.

**Acknowledgments** I would like to thank Sergei Romanenko, whose comments helped to substantially improve the presentation of this article.

## References

1. Davis, J.C.: A Self-verifying Theorem Prover. Ph.D. thesis, University of Texas at Austin, Austin, TX, USA (2009)
2. Friedman, D.P., Eastlund, C.: J-bob source repository. https://github.com/the-little-prover/j-bob, accessed: 2015-04-15
3. Friedman, D.P., Eastlund, C.: The Little Prover. MIT Press (2015)
4. Futamura, Y.: Partial evaluation of computation process–an approach to a compiler-compiler. Higher-Order and Symbolic Computation 12(4), 381–391 (1999), http://dx.doi.org/10.1023/A:1010095604496
5. Glück, R.: A self-applicable online partial evaluator for recursive flowchart languages. Software: Practice and Experience 42(6), 649–673 (2012)
6. Hamilton, G.W.: Distilling programs for verification. Electr. Notes Theor. Comput. Sci. 190(4), 17–32 (2007)
7. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press (2009)
8. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1993)
9. Jones, N.D., Sestoft, P., Søndergaard, H.: An experiment in partial evaluation: The generation of a compiler generator. In: Jouannaud, J.P. (ed.) Rewriting Techniques and Applications: Dijon, France, May 20–22, 1985. pp. 124–140. Springer Berlin Heidelberg, Berlin, Heidelberg (1985)
10. Kaufmann, M., Moore, J.S.: The ACL2 home page. In: http://www.cs.utexas.edu/users/moore/acl2/. Dept. of Computer Sciences, University of Texas at Austin (2016)
11. Klimov, A.V., Klyuchnikov, I.G., Romanenko, S.A.: Automatic verification of counter systems via domain-specific multi-result supercompilation. In: Third International Valentin Turchin Workshop on Metacomputation. pp. 112–141 (2012)
12. Klyuchnikov, I., Krustev, D.: Supercompilation: Ideas and methods. The Monad Reader 23 (2014)
13. Klyuchnikov, I., Romanenko, S.: Proving the equivalence of higher-order terms by means of supercompilation. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) Perspectives of Systems Informatics: 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers. pp. 193–205. Springer Berlin Heidelberg, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-11486-1_17
14. Klyuchnikov, I., Romanenko, S.: Certifying supercompilation for Martin-Löf's type theory. In: Voronkov, A., Virbitskaite, I. (eds.) Perspectives of System Informatics: 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers. pp. 186–200. Springer Berlin Heidelberg, Berlin, Heidelberg (2015), http://dx.doi.org/10.1007/978-3-662-46823-4_16
15. Krustev, D.: A simple supercompiler formally verified in Coq. In: Nemytykh, A.P. (ed.) Proceedings of the Second International Workshop on Metacomputation in Russia (META 2010). pp. 102–127 (2010)

16. Krustev, D.: Towards a framework for building formally verified supercompilers in Coq. In: Loidl, H.W., PeÃśa, R. (eds.) Trends in Functional Programming, Lecture Notes in Computer Science, vol. 7829, pp. 133–148. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-40447-4_9
17. Krustev, D.N.: An approach for modular verification of multi-result supercompilers. In: Klimov, A., Romamenko, S. (eds.) Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation. pp. 177–193. University of Pereslavl Publishing House, Pereslavl-Zalessky, Russia (2014)
18. Lisitsa, A.P., Nemytykh, A.P.: Verification as a parameterized testing (experiments with the SCP4 supercompiler). Programming and Computer Software 33(1), 14–23 (2007), http://dx.doi.org/10.1134/S0361768807010033
19. Mendel-Gleason, G.: Types and verification for infinite state systems. PhD thesis, Dublin City University, Dublin, Ireland (2011)
20. Nemytykh, A.P., Pinchuk, V.A., Turchin, V.F.: A self-applicable supercompiler. In: Danvy, O., Glück, R., Thiemann, P. (eds.) Partial Evaluation: International Seminar Dagstuhl Castle, Germany, February 12–16, 1996 Selected Papers. pp. 322–337. Springer Berlin Heidelberg, Berlin, Heidelberg (1996), http://dx.doi.org/10.1007/3-540-61580-6_16
21. Reich, J.S.: Property-based Testing and Properties as Types: A hybrid approach to supercompiler verification. Ph.D. thesis, University of York (2013)
22. Sørensen, M.H., Glück, R.: Introduction to supercompilation. In: Hatcliff, J., Mogensen, T., Thiemann, P. (eds.) Partial Evaluation: Practice and Theory. Lecture Notes in Computer Science, vol. 1706, pp. 246–270. Springer-Verlag (1999)
23. Turchin, V.: The use of metasystem transition in theorem proving and program optimization. In: de Bakker, J., van Leeuwen, J. (eds.) Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 85, pp. 645–657. Springer Berlin / Heidelberg (1980)
24. Turchin, V.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (July 1986)