Formalisation of a frame stack semantics for a Java-like language*

Aleksy Schubert and Jacek Chrzaszcz

Institute of Informatics, University of Warsaw ul. S. Banacha 2, 02-097 Warsaw, Poland [alx,chrzaszcz]@mimuw.edu.pl

Abstract. We present a Coq formalisation of the small-step operational semantics of *Jafun*, a small Java-like language with classes. This format of semantics makes it possible to naturally specify and prove invariants that should hold at each computation step. In contrast to the Featherweight Java approach the semantics explicitly manipulates frame stack of method calls. Thanks to that one can express properties of computation that depend on execution of particular methods.

On the basis of the semantics, we developed a type system that makes it possible to delineate a notion of a compound value and classify certain methods as extensional functions operating on them. In our formalisation we make a mechanised proof that the operational semantics for the untyped version of the semantics agrees with the one for the typed one. We discuss different methods to make such formalisation effort and provide experiments that substantiate it.

1 Introduction

The small-step semantics [28] can serve as a framework in which interesting invariant properties of computations are naturally expressed. The primary reason for this is that by definition the property should hold at *each computation step*. Both in big-step semantics [15] and in denotational semantics [30] the main focus is on the resulting value and the syntactical structure of the program expression at hand while the intermediate computational steps become hidden.

In design of small-step semantics one can decide to take the Featherweight Java (FJ) [14] approach, in which a method body is directly expanded in place of its call. In this approach, one gets a small formal machinery that is simpler to work with. Still, this formal language model is too simple to relate certain desired properties of interest. A richer model in which the method call stack is directly represented makes it possible to deal with the following cases.

- It is possible to directly represent the management of static scopes, which is vital for escape analysis [32] and simplifies some optimisation analyses.

 $^{^{\}star}$ This work was partially supported by the Polish NCN grant 2013/11/B/ST6/01381.

- It is possible to express properties of evolving call stacks (e.g. that the call stack belongs to a particular regular language, a property that occurs in many security related specifications [6]).
- It is possible to directly express strong computational invariants that require management of access scopes at entry and exit of a method, these include immutability [12] or functionality [7].

These advantages are also recognised among the authors of notable formalisations (e.g. recently CompCert turns largely to small-step semantics [21]) although other approaches, e.g. co-inductive one as in [22,25], may give similar results.

The use of method frame stack was vital for our paper and pencil soundness proof of a type system that makes it possible to delineate the notion of a compound value in a Java-like language and define extensional functions that operate on such values [7]. Since there was no attempt to make a mechanised formalisation of a Java-like language small-step semantics with evaluation based upon method frame stack we decided to develop one and take our type system as a test bed for various approaches to formalisation and discuss their consequences.

As a consequence of these efforts we obtained a formalised semantics of a Java-like language $Jafun^1$ with

- a hierarchy of classes and related subtyping relation;
- small-step reduction relation defined in terms of method frame stacks;
- an example type system of the language that captures the notion of value and extensional functions that transform such values;
- a Church-style version of the type system together with proofs that the operational small-step semantics of the Church-style version agrees with the original, untyped semantics.

On the basis of these formalisation artefacts, we discuss various design decisions and their consequences for development of such semantics, which can be useful in other formalisation efforts. In particular, we stress the following points.

- The natural way to define small-step semantics relation in Coq may result in duplication of cases and cause excessive proving efforts. We propose a methodology to transform the natural definition into a sparing one and demonstrate savings it brings in proof development.
- Various coherence proofs, e.g. soundness and completeness of the typed reduction with regard to the untyped one, require case analysis with large number of cases. Any attempt at automatising of the case analysis requires the person who develops the proof to frequently recognise which case is currently analysed. We propose a method that makes this task significantly easier.
- When automatic case analysis is employed, different strategies of discharging the cases can be used. We discuss the advantages and disadvantages of two approaches to case analysis. In the first one, we destruct all the available case distinctions and discharge cases where no longer case analysis is possible.

 $^{^1}$ The formalisation is available from <code>http://www.mimuw.edu.pl/~alx/jafun.tgz</code>

```
class DList { rep DList prev; Data val; rep DList next;
  rwr DList rd copy() { return this.appRec(null); }
  rwr DList rd appRec(rwr DList newPrev) {
    DList newThis = new rwr DList(newPrev, val, null);
    if (newPrev != null) { newPrev.next = newThis; }
    if (next != null) { newThis.next = next.appRec(newThis); }
    return newThis; }
  rwr DList atm singleton(atm Data v) {
    return new rwr DList(null, v, null); }
}
```

 ${f Fig.\,1.}$ An example of Jafun annotations: a doubly linked list (in Java syntax)

In the second one, we destruct case distinctions in only one definition and apply its results in all the remaining ones. It turns out that although the first approach is more general, the second one results in shorter proofs.

This paper is organised as follows. Section 2 introduces our language. In Section 3, we present the Church-style typed version of our semantics while in Section 4 we discuss the advantages of different ways its coherence proofs can be done. At the end in Section 5 we give an account of works that are related to our efforts and at last in Section 6 we summarise our results.

2 Syntax and semantics of Jafun

Main features of Jafun. Fig. 1 presents an example implementation of doubly linked list with Jafun annotations, written in Java syntax for clarity. The rep annotations for fields (here: prev and next) serve to establish a notion of compound value in Jafun. When a field is marked with rep the reference in the field points to further representation of the value. If a field is not marked (here: val), the reference stored in the field is part of the current value representation, but the object it points to is not. Thanks to these annotations, a one element list containing a given object can be considered the same compound value at two points of a program execution even if the state of the contained object changes between the two points. Moreover, any single element lists can be considered equal, even in different runs of the program, when one considers only single argument list operations.

The goal of the Jafun type system is to establish that suitably annotated methods are extensional functions, i.e., always yield equal results when applied to equal values (e.g. at two points of a program execution) and do not change the state of pre-existing objects (for precise definition of extensional functions and compound value equality in an object-oriented context, we refer the reader to [7]). Such methods however can imperatively modify the state of newly created objects, also using auxiliary methods. This programming style is very flexible as

demonstrated by Okasaki [26], but the modifications performed by the auxiliary methods have to be strictly controlled and that is what access mode annotations rwr, rd, atm are for.

In the example from Fig. 1, the copy method is annotated as a function: its only argument (this) is annotated as rd, i.e. read-only. The other annotation rwr (for read-write) is the annotation of the result of the method, meaning in fact that the result is a freshly allocated object. The auxiliary method appRec has similar annotations and its argument newPrev is also annotated as read-write. Its annotations mean that appRec is not a function, as it can modify its argument newPrev, but this will remain unmodified (unless their representation is shared). Indeed, if newPrev points to the last cell of the copy of the beginning of the current list (until prev), the method appRec will correctly append the copy of rest of the current list to this cell, without modifying the current list. In the end it is clear that the method copy is indeed a function, returning an identical fresh copy of the original unmodified list.

In order to make sure the result of functions do not depend on internal elements of non-rep fields of objects, as they are not part of value representations, such references should be followed neither for reading nor writing which brings to the system a kind of sealed references which need however to be passed around also at the interprocedural level. To mark such references we use the atm annotation (for atomic), which are used in the singleton function in Fig. 1 (which ignores its **this** argument).

To complete the picture, we remark that parts of objects marked with rd (or atm) can be modified (or read and modified), but this can only be done through a different variable with suitable access mode that gives permission to write (or read and write) to the representation.

The complete syntax of Jafun is given in Fig. 2. Apart from annotations, its main differences wrt. Java consist in replacing all instructions with expressions, introducing let expressions, restricting expressions in many positions to identifiers (but thanks to let that does not restrict expressiveness) and replacing sequencing with a semi-colon by sequencing with let. To further simplify the language we do not consider visibility annotations (everything is public) and assume that every class has a single built-in "assign to all fields" constructor (that assumption is used in Fig. 1). The last two elements in Fig. 2 are needed for the reduction semantics and are explained in Sect. 2.1.

More details of the language can be found in [7].

Abstract syntax and its formalisation. The syntax of Jafun (see Fig. 2) is reflected in our formalisation as closely as possible by inductive types. For instance class declarations are defined as

```
 \begin{array}{ll} \textbf{Inductive} & JFClassDeclaration : \textbf{Set} := \\ | & JFCDecl & (cn:JFClassName) & (ex:option & JFClassName) \\ & & & (fields:list & JFFieldDeclaration) \\ & & & & (methods:list & JFMethodDeclaration). \end{array}
```

```
\operatorname{\mathsf{Prog}} \ni \mathbf{C} \quad ::= \operatorname{\mathbf{class}} C_1 \operatorname{\mathbf{ext}} C_2 \left\{ \overline{\mathbf{F}} \ \overline{\mathbf{M}} \right\}
     \mathsf{CId} \ni C
                          ::= \langle identifier \rangle \quad (class name)
\mathsf{AMod} \ni \mu
                           ::= rwr | rd | atm
                                                                            \phi ::= \operatorname{rep} \mid \emptyset
                            := \phi C x
        \mathsf{Id} \ni x
                            ::= \langle identifier \rangle \quad (variable/field name)
                  \mathsf{arg} \ ::= \ \mu \ C \ x
                                                                            argn ::= \emptyset C x
                  \mathsf{Exc} ::= \mu \, C
                                                                             \mathsf{Excn} ::= \emptyset \ C
                  \mathbf{M} ::= \mu \ C \ \mu \ m(\overline{\mathsf{arg}}) \ \mathbf{throws} \ \overline{\mathsf{Exc}} \ \{E\} \ | \ \emptyset \ C \ \emptyset \ m(\overline{\mathsf{argn}}) \ \mathbf{throws} \ \overline{\mathsf{Excn}} \ \{E\}
    \mathsf{MId} \ni m
                          ::= \langle identifier \rangle \pmod{name}
                          := \mathbf{new} \ \mu \ C(\overline{\mathsf{v}}) \ | \ \mathbf{let} \ C \ x = E_1 \ \mathbf{in} \ E_2 \ | \ \mathsf{fieldref} \ | \ \mathsf{fieldref} = \mathsf{v} \ |
   \mathsf{Expr} \ni E
                                      if v_1 == v_2 then E_3 else E_4 \mid v.m(\overline{v}) \mid v \mid throw v \mid
                                      try \{E_1\} catch (\mu C x) \{E_2\}
                        v ::= x \mid \mathbf{this} \mid \mathbf{null}
                                                                              fieldref ::= v.x
                  A ::= C \mid \emptyset
\mathsf{BCtxt} \ni \mathcal{C} \quad ::= \llbracket \ \rrbracket_A \mid \mathsf{let} \ C \ x = \mathcal{C} \ \mathsf{in} \ E \ \mid \mathsf{try} \ \{\mathcal{C}\} \ \mathsf{catch} \ (\mu \ C \ x) \ \{E\}
```

Fig. 2. Abstract syntax of Jafun

which follows the structure of the corresponding grammar rule. We use the *option* type here to represent the variant of class declaration which is not extended (possible only for Object in well formed programs). We use *option* type systematically to convey that some element of syntax may be missing.

A program in Jafun is a list $\overline{\mathbf{C}}$ of class declarations with unique names that contains two predefined classes Object and NPE (for NullPointerException). In our semantics, we represent programs as

```
 \textbf{Definition} \ \textit{JFProgram} \ : \ \textbf{Set} := \ \textit{list} \ \textit{JFClassDeclaration} \, .
```

This choice reflects the situation before the program is loaded into the memory and enables the study of the basic properties that are important for proper execution of the language and are enforced by the loading process. Still, this approach requires us to formulate and maintain in proofs certain well-formedness conditions (gathered in the Coq predicate Well_formed_program CC), since e.g. in post-loading view a program cannot contain duplicate class declarations.

2.1 Overview of mechanised Jafun semantics

Reduction relation. The small step semantics of Jafun is defined with a reduction relation \to presented in Fig. 3. The relation is defined for a fixed program $\overline{\mathbf{C}}$ and connects pairs: heap, frame stack. The general form of the relation is given at the top of the figure. A frame stack $C_1\llbracket E_1 \rrbracket_{A_1} :: \cdots :: C_n\llbracket E_n \rrbracket_{A_n}$, or $\overline{\mathcal{C}}$ for short, is, roughly speaking, a sequence of Jafun expressions in which the current execution point (redex) is marked with a special (unary) symbol $\llbracket \rrbracket_A$. The subscript A determines here if the execution is normal $A = \emptyset$ or exceptional $A \in \overline{\mathbf{C}}$. Each expression on the stack is divided into an evaluation context (the "outer

```
\overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathbf{new} \ \mu \ C(l_1, \dots, l_k)]\!]_{\emptyset} \to h'', \overline{\mathcal{C}} :: \mathcal{C}[\![l_0]\!]_{\emptyset}
(newk)
                where \mathsf{alloc}(h, \overline{\mathcal{C}}, C) = (l_0, h'), \; \mathsf{flds}(C) = x_1, \dots, x_k,
                o = \mathsf{empty}_C\{x_1 \mapsto l_1, \dots, x_k \mapsto l_k\}, \ h'' = h'\{l_0 \mapsto o\}
                                        \begin{split} \overline{\mathbf{C}}, h, \overline{\mathcal{C}} &:: \mathcal{C} \llbracket \mathbf{let} \ C \ x = E_1 \ \mathbf{in} \ E_2 \rrbracket_{\emptyset} \to h, \overline{\mathcal{C}} :: \mathcal{C} [\mathbf{let} \ C \ x = \llbracket E_1 \rrbracket_{\emptyset} \ \mathbf{in} \ E_2 ] \\ \overline{\mathbf{C}}, h, \overline{\mathcal{C}} &:: \mathcal{C} [\mathbf{let} \ C \ x = \llbracket t \rrbracket_{\emptyset} \ \mathbf{in} \ E_2 ] \\ \end{split} 
(letin)
(letgo)
                                        \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathbf{if}\ l_0 == l_1\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2]\!]_{\emptyset} \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![E_1]\!]_{\emptyset}
(ifeq)
                                                                                                                                                                                                                                          where l_0 = l_1
                                        \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C} \mathbf{lif} \ l_0 == l_1 \ \mathbf{then} \ E_1 \ \mathbf{else} \ E_2 \mathbf{l}_\emptyset \to h, \overline{\mathcal{C}} :: \mathcal{C} \mathbf{l}_0 \mathbf{l}_\emptyset
(ifneq)
                                                                                                                                                                                                                                          where l_0 \neq l_1
(mthdnpe) \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[[\mathbf{null}.m(\overline{l})]]_{\emptyset} \to h, \overline{\mathcal{C}} :: \mathcal{C}[[\mathbf{npe}]]_{\mathbb{NPE}}
(mthd)
                                        \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C} \llbracket l.m(\overline{l}) \rrbracket_{\emptyset} \to h, \overline{\mathcal{C}} :: \mathcal{C} \llbracket l.m(\overline{l}) \rrbracket_{\emptyset} :: \llbracket E \rrbracket_{\emptyset}
                where \mathsf{class}(h, l) = D, \mathsf{body}(D, m) = E_0, E = E_0\{l/\mathsf{this}, \bar{l}/\mathsf{parNms}(D, m)\}
(mthdret) \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C} \llbracket l.m(\overline{l}) \rrbracket_{\emptyset} :: \llbracket l' \rrbracket_{\emptyset} \to h, \overline{\mathcal{C}} :: \mathcal{C} \llbracket l' \rrbracket_{\emptyset}
(assignnpe) \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[[\mathbf{null}]] :: \mathcal{L}[[\mathbf{null}]] \to h, \overline{\mathcal{C}} :: \mathcal{C}[[\mathbf{npe}]]]
(assignev) \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[[l_1.x = l]]_{\emptyset} \to h', \overline{\mathcal{C}} :: \mathcal{C}[[l]]_{\emptyset}
where l_1 \neq \mathbf{null}, o = h(l_1)\{x \mapsto l\}, h' = h\{l_1 \mapsto o\}
                                       \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathbf{null}.x]\!]_{\emptyset} \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathbf{npe}]\!]_{\mathtt{NPE}}
(varnpe)
                                       \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C} \llbracket l.x \rrbracket_{\emptyset} \to h, \overline{\mathcal{C}} :: \mathcal{C} \llbracket l' \rrbracket_{\emptyset}
                                                                                                                                                             where l \neq \mathbf{null}, l' = h(l)(x)
(var)
(thrownull) \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathbf{throw\ null}]\!]_{\emptyset} \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathbf{npe}]\!]_{\mathbb{NPE}}
                                       \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathbf{throw}\ l]\!]_{\emptyset} \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![l]\!]_{D}
                                                                                                                                                                               where l \neq \mathbf{null}, \mathsf{class}(h, l) = D
(throw)
(ctchin)
                                       \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[\![\mathbf{try}\ \{E_1\}\ \mathbf{catch}\ (\mu\ C\ x)\ \{E_2\}]\!]_{\emptyset} \to
                                                  h, \overline{\mathcal{C}} :: \mathcal{C}[\mathbf{try} \{ [\![E_1]\!]_{\emptyset} \} \mathbf{catch} (\mu C x) \{ E_2 \} ]
(\text{ctchnrml}) \quad \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[\mathbf{try} \mid [\![l]\!]_{\emptyset}] \quad \mathbf{catch} \ (\mu \ C \ x) \mid \{E_2\}] \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![l]\!]_{\emptyset}
(ctchexok) \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[\mathbf{try} \{ \llbracket l \rrbracket_{C'} \} \mathbf{catch} (\mu \ C \ x) \{ E_2 \}] \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![E_2']\!]_{\emptyset}
                where E'_2 = E_2\{l/x\}, C' \le C
                                        \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[\mathbf{let}\ C\ x = [\![l]\!]_{C'}\ \mathbf{in}\ E] \to h, \overline{\mathcal{C}} :: \mathcal{C}[\![l]\!]_{C'}
                                                                                                                                                                                                                   where C' \neq \emptyset
(methodex) \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C} \llbracket l.m(\overline{l}) \rrbracket_{\emptyset} :: \llbracket l' \rrbracket_{C} \to h, \overline{\mathcal{C}} :: \mathcal{C} \llbracket l' \rrbracket_{C} where C \neq \emptyset
(ctchexnok) \overline{\mathbf{C}}, h, \overline{\mathcal{C}} :: \mathcal{C}[\mathbf{try} \{ \llbracket l \rrbracket_{C'} \} \mathbf{catch} (\mu C x) \{ E_2 \}] \to h, \overline{\mathcal{C}} :: \mathcal{C} \llbracket l \rrbracket_{C'} \}
                where C' \neq \emptyset, C' \not\leq : C
We assume here that l, l', l_0, l_1 \in \mathsf{Loc}, \bar{l} \in \mathsf{Loc}^*, h, h' \in \mathsf{Heap}, \overline{\mathbf{C}} \in \mathsf{Prog}, \overline{\mathcal{C}} \in \mathsf{Stacks},
\mathcal{C},\mathcal{C}_1,\ldots,\mathcal{C}_n \ \in \ \mathsf{BCtxt}, \ C,D \ \in \ \mathsf{Cld}, \ A_1,\ldots,A_n \ \in \ \mathsf{Cld} \ \cup \ \{\emptyset\}, \ m \ \in \ \mathsf{Mld}, \ x \ \in \ \mathsf{Id},
```

 $\overline{\mathbf{C}}, h, \mathcal{C}_1[\![E_1]\!]_{A_1} :: \cdots :: \mathcal{C}_n[\![E_n]\!]_{A_n} \rightarrow h', \mathcal{C}'_1[\![E_1]\!]_{A'_1} :: \cdots :: \mathcal{C}'_m[\![E_m]\!]_{A'_m}.$

 ${\bf Fig.\,3.}\ {\bf Semantic}\ {\bf reduction}\ {\bf relation}\ {\bf of}\ {\it Jafun}$

 $E, E_1, \dots, E_n \in \mathsf{Expr}, \text{ and alloc} : \mathsf{Heap} \times \mathsf{Prog} \times \mathsf{Cld} \to \mathsf{Loc} \times \mathsf{Heap}.$

layer"), denoted by $C_i \in \mathsf{BCtxt}$ (in Fig. 2), and the redex E_i . Since the evaluation context is already partially computed, its syntax essentially comes from a much restricted subset BCtxt of Expr . Naturally, in a stack frame as above, the redexes E_1, \ldots, E_{n-1} are method calls $o.m(\overline{v})$, and $A_1 = \ldots = A_{n-1} = \emptyset$, since a pending exception is actively dispatched only in the topmost (i.e. rightmost) frame.

The reduction rules either define the order of execution (basically, call-by-value from left to right) by specifying how the focus moves within an context expression (as in (letin)), or define the meaning of syntactic constructions (as in (letgo)).

Formalisation. We tried hard to make our Coq formalisation red of the reduction relation as visually close to its paper counterpart in Fig.3 as possible. Let us introduce its elements.

Heaps are defined as maps (suitable instance of a standard library functor from the FSets collection) from natural numbers to objects

```
Definition Heap: Type:= NatMap.t Obj.
```

Objects are pairs that consist of a map from field names (identifiers) to locations, and a class name

```
 \begin{array}{ll} \textbf{Definition} & RawObj := JFXIdMap.\ t\ Loc. \\ \textbf{Definition} & Obj : \ \textbf{Type} := (RawObj * JFClassName)\%type \,. \end{array}
```

All name and identifier types (*JFClassName*, *JFXId* etc.) are implemented by standard library ascii strings. Locations *Loc* are either *null* or natural numbers. Note that this definition of heaps requires us to define and maintain a heap coherence property which says that keys in the above mentioned maps agree with field names in the declared class (expressed as Coq predicate *type_correct_heap CC h*).

In our formalisation, we represent the reduction relation as a partial function:

The type FrameStack used above is a list of Frames, each of which is a triple denoted by $Ctx[[E]]_A$, consisting of a context Ctx, an expression E, and an execution mode A, where None represents a normal execution, and $Some\ D$ an exceptional state in which an exception of class D is being dispatched. To ease the manipulation of contexts (corresponding to BCtxt in Fig. 2), Ctx is a list

of JFCtxLet or JFCtxTry context-elements, where __ (a notation for the constructor tt of trivial type unit) represents a placeholder for deeper context-expression. The notation $Ctx \ [Ct \ [E]] A]_$ used above translates to a frame $(Ct :: Ctx) \ [E] A$, which represents a context Ctx nesting Ct with the redex E and execution mode E. Thanks to these Coq notations we obtain clear optical correspondence between the Coq and the paper versions of the semantic rules while having the comfort of working with list operations on our (single-hole) contexts. Apart from some more noise on the Coq side, the main optical difference between the two presentations is the order of frames on the stack, but this is only relevant for a few rules which consider more than one frame at a time.

3 Type system and typed semantics

Apart from the semantics we define also a type system for our language. We sketch its structure here since its full account is presented in [7]. The type system attributes not only a class, but also an access mode to expressions, specifying what kind of access to the object at hand is possible in the current context. An example rule, for the let-expression, is the following:

$$\frac{C,m;\ \Xi;\ \Gamma_1 \vdash E_1 : \langle C_1, \mu_1 \rangle}{C,m;\ \Xi;\ \Gamma_1, x : \langle C_1, \mu_1 \rangle \vdash E_2 : \langle C_2, \mu_2 \rangle} \\ \frac{C,m;\ \Xi;\ \Gamma_1 \vdash \mathbf{let}\ C_1\ x = E_1\ \mathbf{in}\ E_2 : \langle C_2, \mu_2 \rangle}{C,m;\ \Xi;\ \Gamma_1 \vdash \mathbf{let}\ C_1\ x = E_1\ \mathbf{in}\ E_2 : \langle C_2, \mu_2 \rangle}$$

The typing relation is specified in Coq as an inductive predicate:

Inductive types: $JFExEnv \rightarrow JFEnv \rightarrow JFExpr \rightarrow JFACId \rightarrow Prop := \dots$

It expresses that a judgement of the form C,m; Ξ ; $\Gamma \vdash E : \tau$ holds. The missing arguments for C,m come from the surrounding Coq section containing a class and a method declaration inside which the typing is supposed to hold. The correct typing requires us also to maintain the list of exceptions legal at the current point together with their access modes (variable Ξ represented in type JFExEnv), a regular environment assigning classes and access modes to variables (Γ represented in type JFEnv), the expression at hand (E in type JFExpr) and the pair consisting of a class and an access mode that are assigned to the expression (τ in type JFACId). The constructors of the relation are in one-to-one correspondence with the typing rules.

From the typing system and the semantics we produce a type annotated semantics akin to Church versions of type systems in λ -calculi. The idea of such a Church version is that the expression under consideration contains full information concerning the typing rule that is used to derive its type. Since our semantics and type system are complex, we observed that any attempt at a mechanised consistency proof for our type system would be extremely difficult to get through without a definition of an explicitly typed version of the frame stack. Therefore, we augment all frames in the stack with the typing information and state explicitly many invariants that hold on legal heaps and frame stacks during the correct execution of the semantics.

The basic structure used in the typed semantics is type annotated frame:

Given that a frame TFRfr is a triple $Ctx[[E]]_A$, the purpose of a typed frame is basically that the following typing judgement should be derivable:

```
types TFRcdecl TFRmdecl TFRXi TFRGamma (Ctx[[E]]) TFRAcid
```

This, together with some other obvious requirements (e.g. that the *TFRcdecl* class is part of the program) is formalized in the property called *DerivableTFR*.

Next, we state a number of properties that each frame on the correct typed frame stack should satisfy with respect to the heap. They are listed in the definition of *oneTFRConsistency* that conveys the following properties:

- the environment Gamma from the typed frame should contain unique nonnull locations.
- their types declared in Gamma are supertypes of their types on the heap,
- a reference for the null pointer exception is present in the context.

The properties binding every two adjacent typed frames in the stack are formalised in the inductive definition is TFS ind, these are in particular

- each internal frame of the stack represents a method call expression, in which method parameters are locations,
- the method call is on an object that resides in the heap,
- the class, method and exception context of the subsequent frame are respectively the class from which the method is called, the method itself, and exceptions allowed by the method declaration,
- the return type of the method agrees with the target type in the next frame.

The last auxiliary definition in the sequence is *isTFS* which adds to the former two the requirement that if the top frame is in an exceptional execution mode, then the evaluated expression should be a valid location corresponding to the class of exception that is being dispatched. In the very end, *DerivableTFS* combines *isTFS* with the requirement that each typed frame on the stack is indeed derivable (*DerivableTFR*).

Now we are ready to define a proper typed semantics:

```
Definition typed\_red:

Heap * TFSsupport \rightarrow option (Heap * TFSsupport) := ...
```

It is defined in a context with a current program and where *TFSsupport* is a stack of typed frames. This function extends the output of the normal semantic function *red* with typing information. For example, the rule for method call looks as follows:

```
 \begin{array}{l} (*\ \mathit{mthd}\ *) \\ Ctx[[\mathit{JFInvoke}\ (\mathit{JFVLoc}\ (\mathit{JFLoc}\ n))\ \mathit{m}\ \mathit{vs}]]\_None \Rightarrow \\ \textbf{let}\ \mathit{D0op} := \mathit{getClassName}\ \mathit{h}\ \mathit{n}\ \mathbf{in} \end{array}
```

```
match D0op with | None \Rightarrow None | Some D0 \Rightarrow
  match getInvokeBody CC D0op n m vs h Ctx (FSofTFS Cc) with
   None \Rightarrow None
  | Some (h', []) \Rightarrow None
| Some (h', fr :: _) \Rightarrow
  match find_class CC D0 with None \Rightarrow None | Some cdecl \Rightarrow
  match methodLookup CC D0 m with None \RightarrowNone | Some mdecl \Rightarrow
  match retTypM CC (JFClass D0) m with None \RightarrowNone | Some acid \Rightarrow
  let newTFR :=
   \{ | TFRcdecl := cdecl ; 
                                         TFRmdecl := mdecl;
       TFRXi := thrs_of_md \ mdecl;
       TFRGamma := loc2env D0 mdecl (JFVLoc (JFLoc n) :: vs);
       TFRfr := fr;
                                         TFRAcid := acid
   |} in Some (h', newTFR :: tfs)
  end end end end
while the corresponding one in red is the following:
| (* mthd *)
 (Ctx[[JFInvoke\ (JFVLoc\ (JFLoc\ n))\ m\ vs]]_None) :: Cc \Rightarrow
    let D0 := getClassName h n in
      getInvokeBody CC D0 n m vs h Ctx Cc
```

In spite of the fact that most of the functionality of red for method call is hidden in getInvokeBody, it is clear that much more information must be gathered and inspected in typed_red.

We proved the correspondence of the two relations in the following two theorems. The first of them says (soundness) that a well defined step in the Church version of the semantics implies a corresponding well defined step in the untyped version: both resulting heaps must agree and the type erasure *FSofTFS* of the resulting typed frame stack is the resulting untyped frame stack.

```
Theorem fs_from_tfs_after_tred:

forall h tfs fs h' tfs' res, FSofTFS tfs = fs \rightarrow

typed_red (h, tfs) = Some (h', tfs') \rightarrow

red CC (h, fs) = res \rightarrow res = Some (h', FSofTFS tfs').
```

Surprisingly enough the proof in the direction from the typed version to the untyped one did not require any additional well-formedness conditions. It was the proof in the opposite direction (completeness) that used it heavily.

```
Theorem tfs\_exists\_for\_fs\_after\_red:
for all tfs h fs h' fs' tfsres, Well\_formed\_program CC \rightarrow type\_correct\_heap CC h \rightarrow Derivable TFS CC h tfs \rightarrow FSofTFS tfs = fs \rightarrow well\_formed\_framestack fs \rightarrow red CC (h, fs) = Some (h', fs') \rightarrow typed\_red (h, tfs) = tfsres \rightarrow exists h'' tfs', Some (h'', tfs') = tfsres \land FSofTFS tfs' = tfs' \land h'' = h'.
```

Note that apart from typability constraint DerivableTFS CC h tfs, we assume here that the program is well formed (Well_formed_program CC), the heap is

consistent with the program (type_correct_heap CC h), and that the stack used in the untyped reduction is well formed (well_formed_framestack fs).

4 Proving experiments and improvements

In our proving effort we experimented with a number of techniques to make the proofs shorter and more readable without sacrificing the easy to follow optical correspondence with the paper and pencil version. In some cases the benefits of the experiments were clear from the start (or from the moment we decided that some improvement is needed) but we decided to evaluate the impact of given improvements on our proofs to have a tangible evidence of that.

Format of reduction definitions. Our Coq definition of red, a fragment of which is shown in Section 2.1, directly corresponds to the one in Fig. 3. Unfortunately, these natural expressions are internally transformed by Coq into a nested series of matches over particular datatypes in the following order: frame list, context list, context expression, redex expression, execution mode. In particular, even if only a few rules depend on the context (e.g. (letgo) or exception dispatching rules), the matching on context list and last context expression is done systematically and therefore a dozen rules (like (letin)) which do not depend on the context have to be repeated 3 times: once for the empty context, and twice for non-empty context: for JFCtxLet and JFCtxTry, respectively, as the last context node. This also means that in proofs over red, one would have to handle these repeated rules several times. To prevent this, we introduced the definition red2 where the matching on the frame stack data structure is done more carefully: first on the access mode A and redex expression E and only if E is a variable, we match on the context to get down to particular exception handling rules (if A is not null), or (mthdret), (letgo) or (ctchnrml) (if A is null). The definition red2 is a few lines longer and the semantic rules are reordered compared to red, but the difference in internal representation (obtained by switching off all prettyprinting of matching) is important: 163 vs 476 lines (compared to about 100 lines for each of the definitions in the source files). The two reduction definitions can be automatically proven to be equal for all inputs. In the end we have a close to paper definition red that can be switched to red2 in the proofs and therefore the cases need not to be repeated. We follow the same approach for our typed semantics. Again, we have a natural definition of typed_red and the reordered one called typed_red2.

The benefits of these reorderings can be seen in Fig. 4 by comparing "non-duplication" proofs with previous ones (S2 with S1 and C2 with C1). For the soundness proof the gain in script size was over 50% and in time about 80%. For the completeness, the gain in size and proof generation time was small (about 10%) but the gain in proof-checking time (the time of the final **Qed**) was also large (about 40%). The difference between the two directions is caused by the fact that the proofs of particular subcases in the proof of completeness are significantly longer than ones in the proof of soundness. Still, the number of cases is

			Proof	Proof-
Proof		Size	genera-	checking
		in lines	tion	time
			time	
Soundness non-systematic	(S1)	312	1.636	0.978
Soundness non-systematic, non-duplication	(S2)	149	0.429	0.192
Soundness systematic, automatic	(S3)	20	0.602	0.264
Completeness non-systematic	(C1)	498	3.390	1.300
Completeness non-systematic, non-duplication	(C2)	457	2.927	0.777
Completeness systematic, non-duplication	(C3)	327	1.798	0.662
Completeness systematic, automatic blind	(C4)	111	65.177	16.828
Completeness systematic, well chosen	(C5)	77	4.197	1.534

Fig. 4. Results of experiments. Times are averages from 10 runs on the Coq version 8.7.1 on a Lenovo X240 laptop with 8GiB RAM and Intel i5-4300U CPU at 1.90GHz running Linux Fedora 26 with kernel version 4.16.7.

always smaller, which explains why the typechecking took significantly less time even if proof generation time (which includes searching by **auto**) was comparable.

Comment the cases. Both our main properties relate typed and untyped semantics and therefore both theorems have premises of the form "typed semantics gives this" and "untyped semantics gives that", which both develop into huge nested match expressions. These premises can be simplified and split into (many many) subproofs using the **destruct** tactic and we developed an Ltac tactic that does that automatically. While some of the resulting simple goals are easy to discharge automatically, other require a manual proof. After some experiments and adjustments we restricted the number of these manual cases to 12 in one lemma and to 2 in the other. The main difficulty of the process we observed was that it took us a lot of time to understand in which particular case we are left for the manual proof. To help with this, using a few definitions and notations, we introduced mock comment premises. These comments are systematically created by our automatic tactics using **destruct** with **eqn** argument. In the end, a typical proof situation to analyse looked like this (with about 60 more premises above):

```
d11 : ### A0 = None #
d12 : ### l = JFLoc n0 #
d13 : ### find (TFRGamma e) (JFVLoc (JFLoc n0)) = None #

exists (h'' : Heap) (tfs' : TFSsupport),
Some (h'', tfs') = None \( \chi \) FSofTFS tfs' = fs' \( \chi \) h'' = h'
```

Although **destruct** with **eqn** make these facts present in the list of hypotheses anyway, they are scattered throughout the list, interspersed with variable declarations and it is unclear where they came from. Therefore having them together gathered in one place helps us to realise where we are in the proof in one look.

Destruct and discharge. Another aspect of working with multiple large definitions by cases is the strategy used to destruct and discharge goals. If one wants to automate the process, the easiest approach consists in splitting all definitions into simple cases by destructing expressions that block match constructions, and in the end try to discharge the remaining goals. When applying this strategy blindly it may happen that the same expression is destructed many times.

More precisely, if the destructed expression is a variable, all its occurrences are replaced by fresh constructor terms (in all subcases), so the original variable can never reappear as a match blocker. However, if one destructs a more complex expression, even though all its occurrences are also replaced by constructors, the same expression can be re-formed later in the proof process and it needs to be handled again. Consider a proof situation where one premise contains a subexpression match get_sth None with ... and another one match get_sth opt with ... If one destructs first the expression get_sth None and later the variable opt, the expression get_sth None reappears (in one of the cases). If it is re-destructed without care, one can be confronted with a (false) situation where the case from the second destruct of get_sth None is different than the one from the first destruct. Such situations are sometimes unprovable, but one can easily prevent them using destruct with eqn. This guarantees that the first choice for get_sth None remains in the list of hypotheses, so if the second choice is different, the context becomes clearly inconsistent, which can usually be proved by congruence. Nevertheless, avoiding multiple destruction of the same expression can significantly decrease the number of goals that need to be handled.

In order to do that, before destructing an expression one can first check the context for some equations concerning it. If the given expression has already been destructed, the context must contain an equation recording the past choice of constructor and it is enough to reuse that information.

In our development we did precise measurements of the efficiency of this technique for the proof of completeness. It permitted to reduce the number of cases from over a 1000 to 58. Even though in both versions of the proofs most cases are discharged completely automatically, the reduction gives dramatic gain in time (and memory space as well), from minutes to seconds (compare C4 to C5 in Fig. 4). The time obtained with the automatic destruction tactic with early discharge (C5), although significantly larger than the proof made by systematic manual selection of terms to destruct (C3), remain in the same comfortable time segment of a small few seconds, while shortening the proof scrips several times.

It has to be noted that searching context for equations about a particular expression does not come for free, as one has to search through all hypotheses using Ltac goal matching. Therefore doing it systematically in an automatic case analysis is only beneficial if the probability of repeating an expression is high, which is the case in our situation.

5 Related works

There is a significant number of formalisations of semantics for programming languages so we give here only a rough picture of the landscape.

The most notable line of research here is done by teams that work on semantics of the C programming language. One of the most notable projects here is development of C compiler CompCert in Coq [20]. The semantics of current version of CompCert is primarily expressed in the small-step fashion, but it is also augmented with a big-step counterpart. These efforts are complemented by works of other teams (see e.g. [18,31]).

Another take on the C semantics was proposed in the context of LLVM [35]. The basic non-deterministic small-step semantics serves to describe the LLVM behaviour. This semantics can be used to prove the correctness of compilation transformations that operate locally on instructions. This semantics is further refined into a deterministic one, which makes it possible to execute pieces of code and compare results with the actual LLVM platform. There are two more big-step semantics that are used to prove the correctness of transformations in case they operate on bigger pieces of code (functions or program blocks).

Also Maude rewriting system was used to formalise mechanically the semantics of C [9]. An extension, called K-Maude, was used for this purpose, which made it possible to define semantics, which was small-step in spirit, although it heavily used various structural enhancements of K-Maude.

Formalisations of C compilation process rely on some form of formalisation of the target low-level language. Typical formalisations of low-level languages start from small-step semantics (see e.g. [4,5,10,16,27]) and only then build big-step versions. This approach makes it possible to extract [5,23] from the proofs an interpreter of the target language which has a structure compatible with the imagined process of step-wise computation.

There are formalisations of compiling process for languages other than C. CakeML is a dialect of ML for which a verified implementation of compiler was proposed [13,19]. The development is based upon a big-step operational semantics on which the compiler correctness proof is developed. This is very natural here since structural operational semantics is the semantics format of choice in Standard ML [24]. However, the authors use a small-step semantics for expressions to define divergence and to make a type-soundness proof.

In the context of Java-like languages formalisations occurred mainly to prove soundness of program analyses. Interestingly enough, Strniša et al [33,34] presented a formalisation of a Java-like module system in Isabelle/HOL which served to identify issues with the existing system as well as to highlight important design decisions. Even though module semantics are usually given in big-step form, this formalisation is done in small-step fashion. This is due to the fact that the goal of the formalisation was not to give a formal proof of some module system property, but to have a format in which everything should be expressed both formally and precisely to gain the confidence in the understanding of the system.

Recently a number of formalisations appeared [2,3,29] that take as their basis the Kleene algebra with tests introduced by Kozen [17]. These works exploit the

fact that the high level of abstraction present in the algebra makes it possible to focus separately on expression forms that occur in many languages and develop a general framework to deal with them and only then instantiate it in the context of a particular language and property of interest.

An attractive proposal, similar to the Kleene algebra with tests, is the generic formalisation of program execution in small-step fashion by Dinsdale at al [8]. The main motivation for the small-step format there is that it enables the possibility to express the interleaving semantics for multiple threads (this advantage was also observed e.g. by Amani et al [1]). Additionally, the system separates the flow of control from actions that manipulate the state. This facilitates verification of soundness for expressive type systems in Java-like languages [11] even in the context of multithreading. However, the available control flow expressions make it difficult to handle exceptions and method call stack so it is difficult to express properties such as that a method is an extensional function.

In the mentioned above work of Gordon et al [11] the method call stack is actually introduced implicitly. This is necessary to get the soundness argument through since the system, similarly to ours, requires management of heap regions at the entry and exit from methods. In order to make the management possible, the authors introduce an additional non-standard kind of expression Bind that binds formal method parameters to the actual ones and guards the subexpression in which execution of the method body is performed. In this way the authors obtain implicitly the necessary functionality of method call stack.

6 Conclusions

We presented a Coq formalisation of the frame stack based small-step operational semantics of *Jafun*, a small Java-like language with classes and an effectual type system that makes it possible to delineate a notion of a compound value and classify certain methods as extensional functions. The total size of the whole formalisation is currently over 16900 lines of proof scripts and definitions.

For our type system, we defined a notion of typed frame stacks akin to Church-style expressions in λ -calculi and proved the equivalence of the original reduction and the reduction on the typed stacks. Such a proof turned out to be non-trivial in case of our system since it required 589 lines of a proof script.

Subsequently we studied different methods this proof could be done to observe the impact of different approaches on proof construction efforts and on time of their checking. We measured for example that aggressive prevention of repeating destruction of the same expression can lead to a tenfold reduction in the proof-checking time.

References

 S. Amani, J. Andronick, M. Bortin, C. Lewis, Ch. Rizkallah, and J. Tuong. Complx: A verification framework for concurrent imperative programs. In *Proc. of CPP* 2017, pp. 138–150. ACM, 2017.

- A. Armstrong, V. B. F. Gomes, and G. Struth. Lightweight program construction and verification tools in Isabelle/HOL. In D. Giannakopoulou and G. Salaün, eds., Proc. of SEFM 2014, pp. 5–19. Springer, 2014.
- 3. A. Armstrong, V. B. F. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2):265–293, 2016.
- 4. A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. The Matita interactive theorem prover. In N. Bjørner and V. Sofronie-Stokkermans, eds., *Automated Deduction CADE-23*, pp. 64–69. Springer, 2011.
- R. Atkey. CoqJVM: An executable specification of the Java Virtual Machine using dependent types. In M. Miculan, I. Scagnetto, and F. Honsell, eds., *Proc. of TYPES* 2007, vol. 4941 of *LNCS*, pp. 18–32. Springer, 2008.
- B.-M. Chang. Static check analysis for Java stack inspection. SIGPLAN Not., 41(3):40–48, 2006.
- J. Chrząszcz and A. Schubert. Function definitions for compound values in objectoriented languages. In Proc. of the 19th International Symposium on Principles and Practice of Declarative Programming, PPDP '17, pp. 61–72. ACM, 2017.
- Th. Dinsdale-Young, L. Birkedal, Ph. Gardner, M. Parkinson, and H. Yang. Views: Compositional reasoning for concurrent programs. In *Proc. of POPL '13*. ACM, 2013.
- Ch. Ellison and G. Rosu. An executable formal semantics of C with applications. In Proc. of POPL '12. ACM, 2012.
- A. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In Proc. of the First International Conference on Interactive Theorem Proving, ITP'10, pp. 243–258. Springer, 2010.
- C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *Proc. of OOPSLA '12*, pp. 21–40. ACM, 2012.
- C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In R. De Nicola, ed., *Proc. of ESOP*, vol. 4421 of *LNCS*, pp. 347–362. Springer, 2007.
- L. Hupel and T. Nipkow. A verified compiler from Isabelle/HOL to CakeML. In A. Ahmed, ed., Programming Languages and Systems. Springer, 2018.
- 14. A. Igarashi, B. C. Pierce, and Ph. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst., 23(3):396–450, 2001.
- 15. G. Kahn. Natural semantics. In F. J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, eds., STACS 87, pp. 22–39. Springer, 1987.
- G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Transactions on Programming Languages and Systems, 28(4):619–695, 2006.
- D. Kozen. Kleene algebra with tests. ACM Trans. Program. Lang. Syst., 19(3), 1997.
- R. Krebbers and F. Wiedijk. A typed C11 semantics for interactive theorem proving. In X. Leroy and A. Tiu, eds., Proc. of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015, 2015.
- R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In Proc. of POPL '14. ACM, 2014.
- X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In Proc. of POPL '06, pp. 42–54. ACM Press, 2006.
- X. Leroy. Mechanized semantics for compiler verification. In Ch. Hawblitzel and D. Miller, eds., Certified Programs and Proofs, pp. 4–6. Springer, 2012.

- 22. X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.
- A. Lochbihler and L. Bulwahn. Animating the formalised semantics of a Javalike language. In M. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, eds., Interactive Theorem Proving, pp. 216–232. Springer, 2011.
- R. Milner, M. Tofte, R. Harper, and D. MacQueen. The Definition of Standard ML (Revised). The MIT Press, Cambridge, Massachusetts, 1997.
- B. M. Moore, L. Peña, and G. Rosu. Program verification by coinduction. In A. Ahmed, ed., Proc. of ESOP 2018, pp. 589

 –618. Springer, 2018.
- 26. Ch. Okasaki. Purely functional data structures. Cambridge University Press, 1999.
- 27. D. Pichardie. Bicolano Byte Code Language in Coq. http://mobius.inria.fr/bicolano., 2006.
- G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, 1981.
- D. Pous. Kleene algebra with tests and Coq tools for while programs. In Proc. of the 4th International Conference on Interactive Theorem Proving, ITP'13, pp. 180–196. Springer, 2013.
- D. Scott and Ch. Strachey. Toward a mathematical semantics for computer languages. Oxford Programming Research Group Technical Monograph PRG-6, Oxford University, 1971.
- G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In Proc. of POPL '15. ACM, 2015.
- I. Stilkerich, C. Lang, Ch. Erhardt, Ch. Bay, and M. Stilkerich. The Perfect Getaway: Using escape analysis in embedded real-time systems. ACM Trans. Embed. Comput. Syst., 16(4):99:1–99:30, 2017.
- 33. R. Strniša. Formalising, improving, and reusing the Java Module System. PhD thesis, University of Cambridge, St. John's Collage, 2010.
- 34. R. Strniša, P. Sewell, and M. Parkinson. The Java module system: Core design and semantic definition. In *Proc. of OOPSLA '07*. ACM, 2007.
- J. Zhao, S. Nagarakatte, M.M.K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proc.* of POPL '12, pp. 427–440. ACM, 2012.